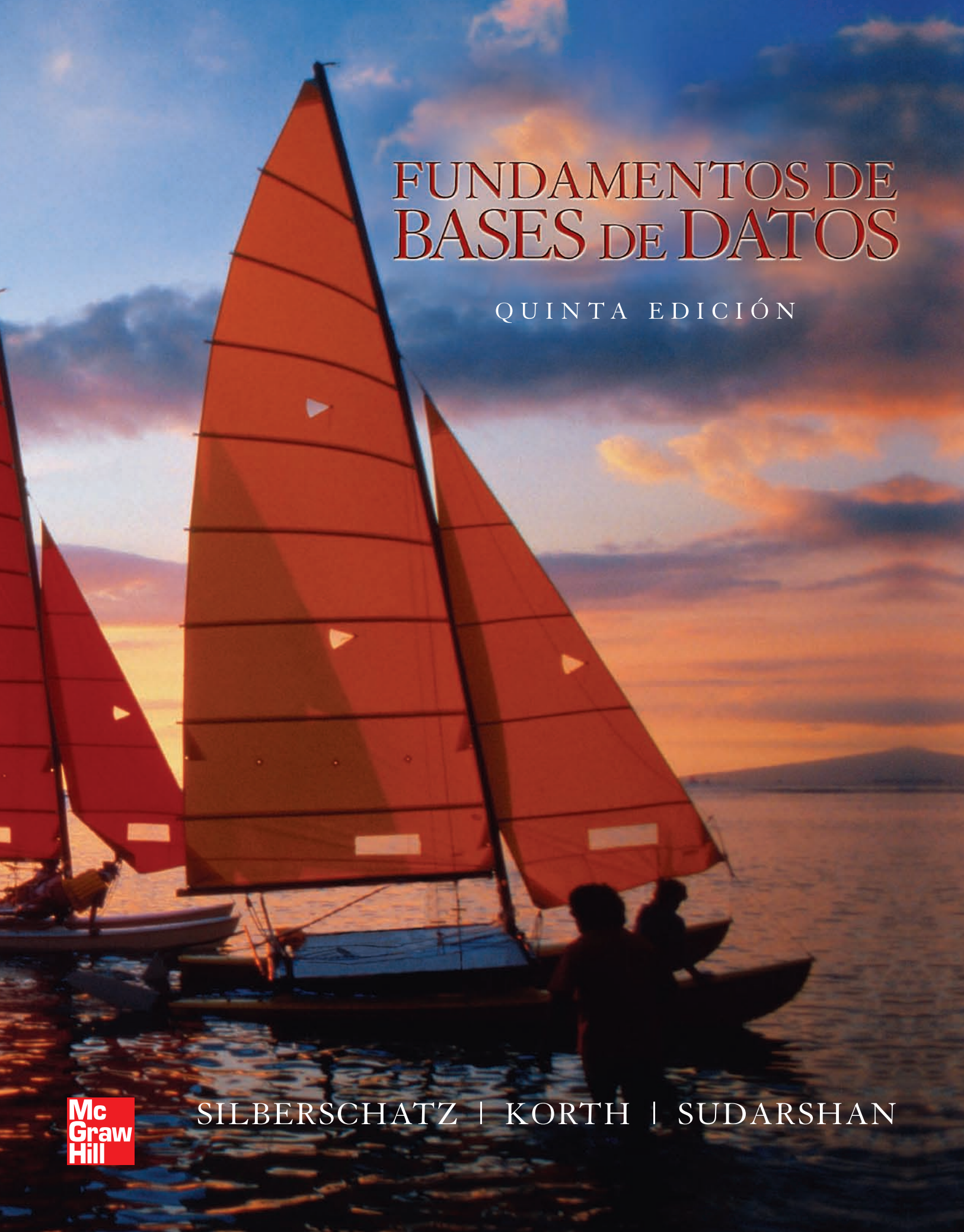

FUNDAMENTOS DE
BASES DE DATOS
QUINTA EDICIÓN
SILBERSCHATZ | KORTH | SUDARSHAN
Mc
Graw
Hill

Fundamentos de bases de datos
Quinta edición

Fundamentos de bases de datos
Quinta edición

ABRAHAM SILBERSCHATZ
Universidad de Yale

HENRY F. KORTH
Universidad de Lehigh

S. SUDARSHAN
Instituto tecnológico indio, Bombay

Traducción

FERNANDO SÁENZ PÉREZ
ANTONIO GARCÍA CORDERO
JESÚS CORREAS FERNÁNDEZ
Universidad Complutense de Madrid

Revisión técnica

LUIS GRAU FERNÁNDEZ
Universidad Nacional de Educación a Distancia

MADRID BOGOTÁ BUENOS AIRES CARACAS GUATEMALA LISBOA
MÉXICO NUEVA YORK PANAMÁ SAN JUAN SANTIAGO SAO PAULO
AUCKLAND HAMBURGO LONDRES MILÁN MONTREAL NUEVA DELHI PARÍS
SAN FRANCISCO SIDNEY SINGAPUR ST. LOUIS TOKIO TORONTO

The McGraw-Hill Companies

La información contenida en este libro procede de una obra original publicada por The McGraw-Hill Companies. No obstante, McGraw-Hill/Interamericana de España no garantiza la exactitud o perfección de la información publicada. Tampoco asume ningún tipo de garantía sobre los contenidos y las opiniones vertidas en dichos textos.

Este trabajo se publica con el reconocimiento expreso de que se está proporcionando una información, pero no tratando de prestar ningún tipo de servicio profesional o técnico. Los procedimientos y la información que se presentan en este libro tienen sólo la intención de servir como guía general.

McGraw-Hill ha solicitado los permisos oportunos para la realización y el desarrollo de esta obra.

Fundamentos de bases de datos, quinta edición

No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión de ninguna forma o por cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro u otros métodos, sin el permiso previo y por escrito de los titulares del Copyright.

McGraw-Hill/Interamericana de España, S.A.U.

DERECHOS RESERVADOS © 2006, respecto a la quinta edición en español, por
McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S.A.U.
Edificio Valrealty, 1ª planta
Basauri, 17
28023 Aravaca (Madrid)

http://www.mcgraw-hill.es
universidad@mcgraw-hill.com

Traducido de la quinta edición en inglés de
Database System Concepts
ISBN: 0-07-295886-3

Copyright © 2006 por The McGraw-Hill Companies, Inc.

ISBN: 84-481-4644-1
Depósito legal: M.

Editor: Carmelo Sánchez González
Compuesto por: Fernando Sáenz Pérez
Impreso en

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

En memoria de mi padre Joseph Silberschatz,
de mi madre Vera Silberschatz
y de mis abuelos Stepha y Aaron Rosenblum.

Avi Silberschatz

A mi esposa, Joan,
mis hijos, Abigail y Joseph,
y mis padres, Henry y Frances.

Hank Korth

A mi esposa, Sita,
mi hijo, Madhur,
y mi madre, Indira.

S. Sudarshan

Contenido

Prefacio xv

Capítulo 1 Introducción

1.1 Aplicaciones de los sistemas de bases de datos 1
1.2 Propósito de los sistemas de bases de datos 2
1.3 Visión de los datos 4
1.4 Lenguajes de bases de datos 7
1.5 Bases de datos relacionales 9
1.6 Diseño de bases de datos 11
1.7 Bases de datos basadas en objetos y semiestructuradas 15
1.8 Almacenamiento de datos y consultas 16
1.9 Gestión de transacciones 17
1.10 Minería y análisis de datos 18
1.11 Arquitectura de las bases de datos 19
1.12 Usuarios y administradores de bases de datos 21
1.13 Historia de los sistemas de bases de datos 22
1.14 Resumen 24
Ejercicios 25
Notas bibliográficas 26

PARTE 1 ■ BASES DE DATOS RELACIONALES

Capítulo 2 El modelo relacional

2.1 La estructura de las bases de datos relacionales 29
2.2 Operaciones fundamentales del álgebra relacional 36
2.3 Otras operaciones del álgebra relacional 44
2.4 Operaciones del álgebra relacional extendida 48
2.5 Valores nulos 53
2.6 Modificación de la base de datos 54
2.7 Resumen 56
Ejercicios 57
Notas bibliográficas 59

Capítulo 3 SQL

3.1 Introducción 61
3.2 Definición de datos 62
3.3 Estructura básica de las consultas SQL 65
3.4 Operaciones sobre conjuntos 71
3.5 Funciones de agregación 73
3.6 Valores nulos 75
3.7 Subconsultas anidadas 76
3.8 Consultas complejas 80
3.9 Vistas 81
3.10 Modificación de la base de datos 84
3.11 Reunión de relaciones** 90
3.12 Resumen 94
Ejercicios 95
Notas bibliográficas 98

Capítulo 4 SQL avanzado

4.1 Tipos de datos y esquemas 101
4.2 Restricciones de integridad 105
4.3 Autorización 111
4.4 SQL incorporado 112
4.5 SQL dinámico 114
4.6 Funciones y procedimientos** 121
4.7 Consultas recursivas** 126
4.8 Características avanzadas de SQL** 129
4.9 Resumen 132
Ejercicios 133
Notas bibliográficas 135

Capítulo 5 Otros lenguajes relacionales

5.1 El cálculo relacional de tuplas 137
5.2 El cálculo relacional de dominios 141
5.3 Query-by-Example 144
5.4 Datalog 151
5.5 Resumen 162
Ejercicios 163
Notas bibliográficas 165

PARTE 2 ■ DISEÑO DE BASES DE DATOS

Capítulo 6 Diseño de bases de datos y el modelo E-R

6.1 Visión general del proceso de diseño 169
6.2 El modelo entidad-relación 171
6.3 Restricciones 176
6.4 Diagramas entidad-relación 180
6.5 Aspectos del diseño entidad-relación 184
6.6 Conjuntos de entidades débiles 189
6.7 Características del modelo E-R extendido 190
6.8 Diseño de una base de datos para un banco 197
6.9 Reducción a esquemas relacionales 200
6.10 Otros aspectos del diseño de bases de datos 207
6.11 El lenguaje de modelado unificado UML** 210
6.12 Resumen 212
Ejercicios 213
Notas bibliográficas 217

Capítulo 7 Diseño de bases de datos relacionales

7.1 Características de los buenos diseños relacionales 219
7.2 Dominios atómicos y la primera forma normal 223
7.3 Descomposición mediante dependencias funcionales 224
7.4 Teoría de las dependencias funcionales 231
7.5 Algoritmos de descomposición 239
7.6 Descomposición mediante dependencias multivaloradas 244
7.7 Más formas normales 248
7.8 Proceso de diseño de las bases de datos 248
7.9 Modelado de datos temporales 251
7.10 Resumen 253
Ejercicios 254
Notas bibliográficas 257

Capítulo 8 Diseño y desarrollo de aplicaciones

8.1 Interfaces de usuario y herramientas 259
8.2 Interfaces Web para bases de datos 262
8.3 Fundamentos de Web 263
8.4 Servlets y JSP 267
8.5 Creación de aplicaciones Web de gran tamaño 271
8.6 Disparadores 273
8.7 Autorización en SQL 278
8.8 Seguridad de las aplicaciones 285
8.9 Resumen 291
Ejercicios 293
Notas bibliográficas 297

PARTE 3 ■ BASES DE DATOS ORIENTADAS A OBJETOS Y XML

Capítulo 9 Bases de datos basadas en objetos

9.1 Visión general 301
9.2 Tipos de datos complejos 302
9.3 Tipos estructurados y herencia en SQL 303
9.4 Herencia de tablas 308
9.5 Tipos array y multiconjunto en SQL 309
9.6 Identidad de los objetos y tipos de referencia en SQL 313
9.7 Implementación de las características O-R 315
9.8 Lenguajes de programación persistentes 316
9.9 Sistemas orientados a objetos y sistemas relacionales orientados a objetos 322
9.10 Resumen 323
Ejercicios 324
Notas bibliográficas 327

Capítulo 10 XML

10.1 Motivación 329
10.2 Estructura de los datos XML 332
10.3 Esquema de los documentos XML 335
10.4 Consulta y transformación 340
10.5 La interfaz de programación de aplicaciones de XML 349
10.6 Almacenamiento de datos XML 350
10.7 Aplicaciones XML 354
10.8 Resumen 358
Ejercicios 360
Notas bibliográficas 362

PARTE 4 ■ ALMACENAMIENTO DE DATOS Y CONSULTAS

Capítulo 11 Almacenamiento y estructura de archivos

11.1 Visión general de los medios físicos de almacenamiento 367
11.2 Discos magnéticos 370
11.3 RAID 375
11.4 Almacenamiento terciario 382
11.5 Acceso al almacenamiento 383
11.6 Organización de los archivos 386
11.7 Organización de los registros en archivos 390
11.8 Almacenamiento con diccionarios de datos 393
11.9 Resumen 395
Ejercicios 396
Notas bibliográficas 398

Capítulo 12 Indexación y asociación

12.1 Conceptos básicos 401
12.2 Índices ordenados 402
12.3 Archivos de índices de árbol B^+ 408
12.4 Archivos de índices de árbol B 417
12.5 Accesos bajo varias claves 418
12.6 Asociación estática 421
12.7 Asociación dinámica 426
12.8 Comparación de la indexación ordenada y la asociación 431
12.9 Índices de mapas de bits 432
12.10 Definición de índices en SQL 435
12.11 Resumen 436
Ejercicios 438
Notas bibliográficas 440

Capítulo 13 Procesamiento de consultas

13.1 Visión general 443
13.2 Medidas del coste de una consulta 445
13.3 Operación selección 446
13.4 Ordenación 450
13.5 Operación reunión 452
13.6 Otras operaciones 463
13.7 Evaluación de expresiones 466
13.8 Resumen 470
Ejercicios 472
Notas bibliográficas 473

Capítulo 14 Optimización de consultas

14.1 Visión general 475
14.2 Transformación de expresiones relacionales 476
14.3 Estimación de las estadísticas de los resultados de las expresiones 482
14.4 Elección de los planes de evaluación 487
14.5 Vistas materializadas** 494
14.6 Resumen 498
Ejercicios 500
Notas bibliográficas 502

PARTE 5 ■ GESTIÓN DE TRANSACCIONES

Capítulo 15 Transacciones

15.1 Concepto de transacción 507
15.2 Estados de una transacción 510
15.3 Implementación de la atomicidad y la durabilidad 512
15.4 Ejecuciones concurrentes 513
15.5 Secuencialidad 516
15.6 Recuperabilidad 520
15.7 Implementación del aislamiento 522
15.8 Comprobación de la secuencialidad 522
15.9 Resumen 523
Ejercicios 525
Notas bibliográficas 527

Capítulo 16 Control de concurrencia

16.1 Protocolos basados en el bloqueo 529
16.2 Protocolos basados en marcas temporales 539
16.3 Protocolos basados en validación 542
16.4 Granularidad múltiple 544
16.5 Esquemas multiversión 546
16.6 Tratamiento de interbloqueos 548
16.7 Operaciones para insertar y borrar 553
16.8 Niveles débiles de consistencia 555
16.9 Concurrencia en los índices** 557
16.10 Resumen 560
Ejercicios 562
Notas bibliográficas 566

Capítulo 17 Sistema de recuperación

17.1 Clasificación de los fallos 567
17.2 Estructura del almacenamiento 568
17.3 Recuperación y atomicidad 571
17.4 Recuperación basada en el registro histórico 572
17.5 Transacciones concurrentes y recuperación 579
17.6 Gestión de la memoria intermedia 581
17.7 Fallo con pérdida de almacenamiento no volátil 584
17.8 Técnicas avanzadas de recuperación** 585
17.9 Sistemas remotos de copias de seguridad 591
17.10 Resumen 593
Ejercicios 596
Notas bibliográficas 597

PARTE 6 ■ MINERíA DE DATOS Y RECUPERACIÓN DE INFORMACIÓN

Capítulo 18 Análisis y minería de datos

18.1 Sistemas de ayuda a la toma de decisiones 601
18.2 Análisis de datos y OLAP 602
18.3 Almacenes de datos 612
18.4 Minería de datos 615
18.5 Resumen 625
Ejercicios 627
Notas bibliográficas 628

Capítulo 19 Recuperación de información

19.1 Visión general 631
19.2 Clasificación por relevancia según los términos 632
19.3 Relevancia según los hipervínculos 635
19.4 Sinónimos, homónimos y ontologías 638
19.5 Creación de índices de documentos 639
19.6 Medida de la efectividad de la recuperación 640
19.7 Motores de búsqueda en Web 641
19.8 Recuperación de información y datos estructurados 642
19.9 Directorios 643
19.10 Resumen 645
Ejercicios 646
Notas bibliográficas 647

PARTE 7 ■ ARQUITECTURA DE SISTEMAS

Capítulo 20 Arquitecturas de los sistemas de bases de datos

20.1 Arquitecturas centralizadas y cliente–servidor 651
20.2 Arquitecturas de sistemas servidores 653
20.3 Sistemas paralelos 657
20.4 Sistemas distribuidos 663
20.5 Tipos de redes 666
20.6 Resumen 668
Ejercicios 669
Notas bibliográficas 671

Capítulo 21 Bases de datos paralelas

21.1 Introducción 673
21.2 Paralelismo de E/S 674
21.3 Paralelismo entre consultas 677
21.4 Paralelismo en consultas 678
21.5 Paralelismo en operaciones 678
21.6 Paralelismo entre operaciones 685
21.7 Diseño de sistemas paralelos 687
21.8 Resumen 688
Ejercicios 689
Notas bibliográficas 691

Capítulo 22 Bases de datos distribuidas

22.1 Bases de datos homogéneas y heterogéneas 693
22.2 Almacenamiento distribuido de datos 694
22.3 Transacciones distribuidas 697
22.4 Protocolos de compromiso 698
22.5 Control de la concurrencia en las bases de datos distribuidas 704
22.6 Disponibilidad 710
22.7 Procesamiento distribuido de consultas 714
22.8 Bases de datos distribuidas heterogéneas 717
22.9 Sistemas de directorio 719
22.10 Resumen 723
Ejercicios 726
Notas bibliográficas 728

PARTE 8 ■ OTROS TEMAS

Capítulo 23 Desarrollo avanzado de aplicaciones

23.1 Ajuste del rendimiento 733
23.2 Pruebas de rendimiento 741
23.3 Normalización 744
23.4 Migración de aplicaciones 748
23.5 Resumen 749
Ejercicios 750
Notas bibliográficas 751

Capítulo 24 Tipos de datos avanzados y nuevas aplicaciones

24.1 Motivación 753
24.2 El tiempo en las bases de datos 754
24.3 Datos espaciales y geográficos 756
24.4 Bases de datos multimedia 765
24.5 Computadoras portátiles y bases de datos personales 767
24.6 Resumen 772
Ejercicios 773
Notas bibliográficas 775

Capítulo 25 Procesamiento avanzado de transacciones

25.1 Monitores de procesamiento de transacciones 777
25.2 Flujos de trabajo de transacciones 781
25.3 Comercio electrónico 786
25.4 Bases de datos en memoria principal 788
25.5 Sistemas de transacciones de tiempo real 790
25.6 Transacciones de larga duración 791
25.7 Gestión de transacciones en varias bases de datos 796
25.8 Resumen 799
Ejercicios 801
Notas bibliográficas 802

PARTE 9 ■ ESTUDIO DE CASOS

Capítulo 26 PostgreSQL

Anastassia Ailamaki, Sailesh Krishnamurthy, Spiros Papadimitriou, Bianca Schroeder

26.1 Introducción 807
26.2 Interfaces de usuario 808
26.3 Variaciones y extensiones de SQL 809
26.4 Gestión de transacciones en PostgreSQL 817
26.5 Almacenamiento e índices 824
26.6 Procesamiento y optimización de consultas 827
26.7 Arquitectura del sistema 830
Notas bibliográficas 831

Capítulo 27 Oracle

Hakan Jakobsson

27.1 Herramientas para el diseño de bases de datos y la consulta 833
27.2 Variaciones y extensiones de SQL 834
27.3 Almacenamiento e índices 836
27.4 Procesamiento y optimización de consultas 844
27.5 Control de concurrencia y recuperación 849
27.6 Arquitectura del sistema 851
27.7 Réplica, distribución y datos externos 854
27.8 Herramientas de gestión de bases de datos 855
27.9 Minería de datos 856
Notas bibliográficas 857

Capítulo 28 DB2 Universal Database de IBM

Sriram Padmanabhan

28.1 Visión general 859
28.2 Herramientas de diseño de bases de datos 860
28.3 Variaciones y extensiones de SQL 861
28.4 Almacenamiento e indexación 864
28.5 Agrupación multidimensional 867
28.6 Procesamiento y optimización de consultas 870
28.7 Tablas de consultas materializadas 874
28.8 Características autónomas de DB2 876
28.9 Herramientas y utilidades 876
28.10 Control de concurrencia y recuperación 878
28.11 Arquitectura del sistema 880
28.12 Réplicas, distribución y datos externos 881
28.13 Características de inteligencia de negocio 882
Notas bibliográficas 882

Capítulo 29 SQL Server de Microsoft

Sameet Agarwal, José A. Blakeley, Thierry D'Hers, Gerald Hinson, Dirk Myers, Vaqar Pirzada, Bill Ramos, Balaji Rathakrishnan, Michael Rys, Florian Waas, Michael Zwilling

29.1 Herramientas para la administración, el diseño y la consulta de las bases de datos 885
29.2 Variaciones y extensiones de SQL 889
29.3 Almacenamiento e índices 892
29.4 Procesamiento y optimización de consultas 895
29.5 Concurrencia y recuperación 899
29.6 Arquitectura del sistema 903
29.7 Acceso a los datos 904
29.8 Procesamiento de consultas heterogéneas distribuidas 905
29.9 Duplicación 906
29.10 Programación de servidores en .NET 908
29.11 Soporte de XML en SQL Server 2005 912
29.12 Service Broker de SQLServer 916
29.13 Almacenes de datos e inteligencia de negocio 918
Notas bibliográficas 921

Bibliografía 923

Índice 943

Prefacio

La gestión de las bases de datos ha evolucionado desde una aplicación informática especializada hasta convertirse en parte esencial de los entornos informáticos modernos. Por tanto, el conocimiento acerca de los sistemas de bases de datos se ha convertido en una parte imprescindible de la formación en informática. En este texto se presentan los conceptos fundamentales de la gestión de las bases de datos. Estos conceptos incluyen aspectos del diseño de bases de datos, de los lenguajes y de la implementación de los sistemas de bases de datos.

Este libro está orientado a un primer curso de bases de datos para los niveles técnicos y superiores. Además del material básico para un primer curso, el texto también contiene material avanzado que se puede usar como complemento del curso o como material introductorio de un curso avanzado.

En este libro se asume que se dispone de conocimientos básicos sobre estructuras de datos básicas, organización de computadoras y un lenguaje de programación de alto nivel (tipo Pascal). Los conceptos se presentan usando descripciones intuitivas, muchas de las cuales están basadas en el ejemplo propuesto de una entidad bancaria. Se tratan los resultados teóricos importantes, pero se omiten las demostraciones formales. En lugar de las demostraciones se usan figuras y ejemplos para sugerir su justificación. Las descripciones formales y las pruebas pueden hallarse en los artículos de investigación y en los textos avanzados a los que se hace referencia en las notas bibliográficas.

Los conceptos y algoritmos fundamentales tratados en este libro suelen basarse en los usados en sistemas de bases de datos comerciales o experimentales ya existentes. El objetivo es presentar esos conceptos y algoritmos en un marco general que no esté vinculado a ningún sistema de bases de datos en concreto. Los detalles de los sistemas de bases de datos concretos se estudian en la Parte 9, "Estudio de casos".

En esta quinta edición de *Fundamentos de bases de datos* se ha mantenido el estilo global de las ediciones anteriores, mientras que su contenido y organización han evolucionado para reflejar las modificaciones que se están produciendo en el modo en que las bases de datos se diseñan, se gestionan y se usan. También se han tenido en cuenta las tendencias en la enseñanza de los conceptos de bases de datos y se han hecho adaptaciones para facilitar esas tendencias donde ha resultado conveniente. Antes de describir el contenido del libro con detalle, se destacarán algunas de las características de la quinta edición.

- **Tratamiento precoz de SQL.** Muchos profesores usan SQL como componente principal de sus proyectos de fin de curso (visítese el sitio Web http://www.mhe.es/universidad/informatica/fundamentos para ver proyectos de ejemplo). Con objeto de proporcionar a los estudiantes tiempo suficiente para realizar los proyectos, especialmente en universidades que trabajen por trimestres, resulta fundamental enseñar SQL tan pronto como sea posible. Con esto en mente se han llevado a cabo varias modificaciones en la organización del libro:
  1. Aplazar la presentación del modelo entidad-relación hasta la Parte 2, titulada "Diseño de bases de datos".

2. Racionalizar la introducción del modelo relacional mediante el aplazamiento del tratamiento del cálculo relacional hasta el Capítulo 5, mientras se mantiene el tratamiento del álgebra relacional en el Capítulo 2.
3. Dedicar dos de los primeros capítulos a SQL. El Capítulo 3 trata las características básicas de SQL, incluidas la definición y la manipulación de los datos. El Capítulo 4 trata características más avanzadas, como las restricciones de integridad, SQL dinámico y los constructores procedimentales. Entre el material nuevo de este capítulo figura un tratamiento más amplio de JDBC, de los constructores procedimentales en SQL, de la recursión en SQL y de las nuevas características de SQL:2003. Este capítulo también incluye una breve visión general de las autorizaciones, aunque su tratamiento detallado se pospone hasta el Capítulo 8.

Estas modificaciones permiten que los estudiantes comiencen a escribir consultas de SQL en una etapa inicial del curso y que se familiaricen con el empleo de los sistemas de bases de datos. Esto también permite a los estudiantes desarrollar una intuición sobre el diseño de bases de datos que facilita la enseñanza de las metodologías de diseño en la Parte 2 del texto. Se ha descubierto que los estudiantes aprecian mejor los aspectos del diseño de bases de datos con esta organización.

- **Una parte nueva (la Parte 2) que está dedicada al diseño de las bases de datos.** La Parte 2 del texto contiene tres capítulos dedicados al diseño de las bases de datos y de las aplicaciones para bases de datos. Aquí se incluye un capítulo (el Capítulo 6) sobre el modelo entidad-relación que contiene todo el material del capítulo correspondiente de la cuarta edición (el Capítulo 2), además de varias actualizaciones significativas. También se presenta en el Capítulo 6 una breve visión general del proceso de diseño de las bases de datos. Los profesores que prefieran comenzar el curso con el modelo E-R pueden empezar por este capítulo sin pérdida de continuidad, ya que se ha hecho todo lo posible para evitar las dependencias con cualquier capítulo anterior salvo con el Capítulo 1.

  El Capítulo 7, sobre el diseño relacional, presenta el material tratado en el Capítulo 7 de la cuarta edición, pero con un estilo nuevo y más legible. Los conceptos de diseño del modelo E-R se usan para crear una visión general intuitiva de los aspectos del diseño relacional, adelantándose a la presentación del enfoque formal al diseño mediante dependencias funcionales y multivaloradas y la normalización algorítmica. Este capítulo también incluye un apartado nuevo sobre los aspectos temporales del diseño de bases de datos.

  La Parte 2 concluye con un nuevo capítulo, el Capítulo 8, que describe el diseño y el desarrollo de las aplicaciones de bases de datos, incluidas las aplicaciones Web, los servlets, JSP, los disparadores y los aspectos de seguridad. Para atender a la creciente necesidad de proteger el software de ataques, el tratamiento de la seguridad ha aumentado significativamente respecto de la cuarta edición.

- **Tratamiento completamente revisado y actualizado de las bases de datos basadas en objetos y de XML.** La Parte 3 incluye un capítulo profundamente revisado sobre las bases de datos basadas en objetos que pone el énfasis en las características relacionales de objetos de SQL y sustituye a los capítulos independientes sobre bases de datos orientadas a objetos y bases de datos relacionales de objetos de la cuarta edición. Se ha eliminado parte del material introductorio sobre la orientación a objetos de cursos anteriores con el que los estudiantes están familiarizados, al igual que los detalles sintácticos del ahora obsoleto estándar ODMG. No obstante, se han conservado conceptos importantes subyacentes a las bases de datos orientadas a objetos, incluyendo nuevo material sobre el estándar JDO para añadir persistencia a Java.

  La Parte 3 incluye también un capítulo sobre el diseño y la consulta de datos XML, que se ha revisado a fondo respecto del capítulo correspondiente de la cuarta edición. Contiene un tratamiento mejorado de XML Schema y de XQuery, tratamiento del estándar SQL/XML y más ejemplos de aplicaciones XML, incluyendo servicios Web.

- **Material reorganizado sobre la minería de datos y la recuperación de la información.** La minería de datos y el procesamiento analítico en conexión son actualmente usos extremadamente importantes de las bases de datos—ya no sólo "temas avanzados". Por tanto, se ha trasladado el

tratamiento de estos temas a una parte nueva, la Parte 6, que contiene un capítulo sobre minería y análisis de datos junto con otro capítulo sobre la recuperación de la información.

- **Nuevo caso de estudio: PostgreSQL.** PostgreSQL es un sistema de bases de datos de código abierto que ha conseguido enorme popularidad en los últimos años. Además de ser una plataforma sobre la que crear aplicaciones de bases de datos, el código fuente puede estudiarse y ampliarse en cursos que pongan énfasis en la implementación de sistemas de bases de datos. Por tanto, se ha añadido un caso de estudio de PostgreSQL a la Parte 9, en la que se suma a los tres casos de estudio que aparecían en la cuarta edición (Oracle, IBM DB2 y Microsoft SQL Server). Estos tres últimos casos de estudio se han actualizado para que reflejen las últimas versiones del software respectivo.

El tratamiento de los temas que no se han mencionado anteriormente, incluidos el procesamiento de transacciones (concurrencia y recuperación), las estructuras de almacenamiento, el procesamiento de consultas y las bases de datos distribuidas y paralelas, se ha actualizado respecto de sus equivalentes de la cuarta edición, aunque su organización general permanezca relativamente intacta. El tratamiento de QBE en el Capítulo 5 se ha revisado, eliminando los detalles sintácticos de la agregación y de las actualizaciones que no corresponden a ninguna implementación real, pero se han conservado los conceptos fundamentales de QBE.

Organización

El texto está organizado en nueve partes principales y tres apéndices.

- **Visión general** (Capítulo 1). En el Capítulo 1 se proporciona una visión general de la naturaleza y propósito de los sistemas de bases de datos. Se explica cómo se ha desarrollado el concepto de sistema de bases de datos, cuáles son las características usuales de los sistemas de bases de datos, lo que proporciona al usuario un sistema de bases de datos y cómo se comunican los sistemas de bases de datos con los sistemas operativos. También se introduce un ejemplo de aplicación de las bases de datos: una entidad bancaria que consta de muchas sucursales. Este ejemplo se usa a lo largo de todo el libro. Este capítulo es de naturaleza justificativa, histórica y explicativa.
- **Parte 1: Bases de datos relacionales** (Capítulos 2 a 5). El Capítulo 2 introduce el modelo relacional de datos y trata de conceptos básicos y del álgebra relacional. Este capítulo también ofrece una breve introducción a las restricciones de integridad. Los Capítulos 3 y 4 se centran en el más influyente de los lenguajes relacionales orientados al usuario: SQL. Mientras que el Capítulo 3 ofrece una introducción básica a SQL, el Capítulo 4 describe características de SQL más avanzadas, incluido el modo de establecer comunicación entre un lenguaje de programación y una base de datos que soporte SQL. El Capítulo 5 trata de otros lenguajes relacionales, incluido el cálculo relacional, QBE y Datalog.

  Los capítulos de esta parte describen la manipulación de los datos: consultas, actualizaciones, inserciones y eliminaciones y dan por supuesto que se ha proporcionado un diseño de esquema. Los aspectos del diseño de esquemas se posponen hasta la Parte 2.
- **Parte 2: Diseño de bases de datos** (Capítulos 6 a 8). El Capítulo 6 ofrece una visión general del proceso de diseño de las bases de datos, con el énfasis puesto en el diseño mediante el modelo de datos entidad-relación. Este modelo ofrece una vista de alto nivel de los aspectos del diseño de las bases de datos y de los problemas que se hallan al capturar la semántica de las aplicaciones realistas en las restricciones de un modelo de datos. La notación de los diagramas de clase UML también se trata en este capítulo.

  El Capítulo 7 introduce la teoría del diseño de las bases de datos relacionales. Se tratan la teoría de las dependencias funcionales y de la normalización, con el énfasis puesto en la motivación y la comprensión intuitiva de cada forma normal. Este capítulo comienza con una visión general del diseño relacional y se basa en la comprensión intuitiva de la implicación lógica de las dependencias funcionales. Esto permite introducir el concepto de normalización antes de haber tratado completamente la teoría de la dependencia funcional, que se presenta más avanzado el capítulo.

Los profesores pueden decidir usar únicamente este tratamiento inicial de los Apartados 7.1 a 7.3 sin pérdida de continuidad. Los profesores que empleen todo el capítulo conseguirán que los estudiantes tengan una buena comprensión de los conceptos de normalización para justificar algunos de los conceptos más difíciles de comprender de la teoría de la dependencia funcional.

El Capítulo 8 trata del diseño y del desarrollo de las aplicaciones. Este capítulo pone énfasis en la creación de aplicaciones de bases de datos con interfaces basadas en Web. Además, el capítulo trata de la seguridad de las aplicaciones.

- **Parte 3: Bases de datos basadas en objetos y XML** (Capítulos 9 y 10). El Capítulo 9 trata de las bases de datos basadas en objetos. Este capítulo describe el modelo de datos objeto-relación, que amplía el modelo de datos relacional para dar soporte a tipos de datos complejos, la herencia de tipos y la identidad de los objetos. Este capítulo también describe el acceso a las bases de datos desde lenguajes de programación orientados a objetos.

  El Capítulo 10 trata del estándar XML para la representación de datos, que está experimentando un uso creciente en el intercambio y el almacenamiento de datos complejos. Este capítulo también describe los lenguajes de consultas para XML.

- **Parte 4: Almacenamiento de datos y consultas** (Capítulos 11 a 14). El Capítulo 11 trata de la estructura de disco, de archivos y del sistema de archivos. En el Capítulo 12 se presenta una gran variedad de técnicas de acceso a los datos, incluidos los índices asociativos y de árbol B^+. Los Capítulos 13 y 14 abordan los algoritmos de evaluación de consultas y su optimización. En estos capítulos se examinan los aspectos internos de los componentes de almacenamiento y de recuperación de las bases de datos.

- **Parte 5: Gestión de transacciones** (Capítulos 15 a 17). El Capítulo 15 se centra en los fundamentos de los sistemas de procesamiento de transacciones, incluidas la atomicidad, la consistencia, el aislamiento y la durabilidad de las transacciones, así como la noción de la secuencialidad.

  El Capítulo 16 se centra en el control de concurrencia y presenta varias técnicas para garantizar la secuencialidad, incluidos el bloqueo, las marcas de tiempo y las técnicas optimistas (de validación). Este capítulo también trata los interbloqueos.

  El Capítulo 17 trata las principales técnicas para garantizar la ejecución correcta de las transacciones pese a las caídas del sistema y los fallos de los discos. Estas técnicas incluyen los registros, los puntos de revisión y los volcados de las bases de datos.

- **Parte 6: Minería de datos y recuperación de la información** (Capítulos 18 y 19). El Capítulo 18 introduce el concepto de almacén de datos y explica la minería de datos y el procesamiento analítico en conexión (*online analytical processing*, OLAP), incluido el soporte de OLAP y del almacenamiento de datos por SQL:1999. El Capítulo 19 describe las técnicas de recuperación de datos para la consulta de datos textuales, incluidas las técnicas basadas en hipervínculos usadas en los motores de búsqueda Web.

  La Parte 6 usa los conceptos de modelado y de lenguaje de las partes 1 y 2, pero no depende de las partes 3, 4 o 5. Por tanto, puede incorporarse fácilmente en cursos que se centren en SQL y en el diseño de bases de datos.

- **Parte 7: Arquitectura de sistemas** (Capítulos 20 a 22). El Capítulo 20 trata de la arquitectura de los sistemas informáticos y describe la influencia de los subyacentes a los sistemas de bases de datos. En este capítulo se estudian los sistemas centralizados, los sistemas cliente–servidor, las arquitecturas paralela y distribuida, y los tipos de red.

  El Capítulo 21, que trata las bases de datos paralelas, explora una gran variedad de técnicas de paralelismo, incluidos el paralelismo de E/S, el paralelismo en consultas y entre consultas, y el paralelismo en operaciones y entre operaciones. Este capítulo también describe el diseño de sistemas paralelos.

  El Capítulo 22 trata de los sistemas distribuidos de bases de datos, revisitando los aspectos del diseño de bases de datos, la gestión de las transacciones y la evaluación y la optimización de las consultas en el contexto de las bases de datos distribuidas. Este capítulo también trata aspectos de la disponibilidad de los sistemas durante los fallos y describe el sistema de directorios LDAP.

- **Parte 8: Otros temas** (Capítulos 23 a 25). El Capítulo 23 trata de las pruebas de rendimiento, el ajuste de rendimiento, la normalización y la migración de aplicaciones desde sistemas heredados.

  El Capítulo 24 trata de los tipos de datos avanzados y de las nuevas aplicaciones, incluidos los datos temporales, los datos espaciales y geográficos, y los aspectos de la gestión de bases de datos móviles y personales.

  Finalmente, el Capítulo 25 trata del procesamiento avanzado de transacciones. Entre los temas tratados están los monitores de procesamiento de transacciones, los flujos de trabajo transaccionales, el comercio electrónico, los sistemas de transacciones de alto rendimiento, los sistemas de transacciones en tiempo real, las transacciones de larga duración y la gestión de transacciones en sistemas con múltiples bases de datos.

- **Parte 9: Estudio de casos** (Capítulos 26 a 29). En esta parte se estudian cuatro de los principales sistemas de bases de datos, como PostgreSQL, Oracle, DB2 de IBM y SQL Server de Microsoft. Estos capítulos destacan las características propias de cada uno de los sistemas y describen su estructura interna. Ofrecen gran abundancia de información interesante sobre los productos respectivos y ayudan al lector a comprender el uso en los sistemas reales de las diferentes técnicas de implementación descritas en partes anteriores. También tratan varios aspectos prácticos interesantes del diseño de sistemas reales.

- **Apéndices en Internet**. Aunque la mayor parte de las nuevas aplicaciones de bases de datos usan el modelo relacional o el modelo relacional orientado a objetos, los modelos de datos de red y jerárquico se siguen usando en algunas aplicaciones heredadas. Para los lectores interesados en estos modelos de datos se ofrecen apéndices que describen los modelos de datos de red y jerárquico, en los Apéndices A y B respectivamente; los apéndices sólo se encuentran disponibles en la dirección http://www.mhe.es/universidad/informatica/fundamentos. Estos apéndices sólo están disponibles en inglés.

  El Apéndice C describe el diseño avanzado de bases de datos relacionales, incluida la teoría de las dependencias multivaloradas, las dependencias de reunión y las formas normales de proyección-reunión y de dominio-clave. Este apéndice está pensado para quienes deseen estudiar la teoría del diseño de bases de datos relacionales con más detalle y para los profesores que deseen hacerlo en sus cursos. De nuevo, este apéndice está disponible únicamente en la página Web del libro.

La quinta edición

La producción de esta quinta edición ha estado guiada por los muchos comentarios y sugerencias que se han recibido relativos a ediciones anteriores, por nuestras propias observaciones al ejercer la docencia en la Universidad de Yale, la Universidad de Lehigh y el IIT de Bombay y por nuestro análisis de las direcciones hacia las que está evolucionando la tecnología de las bases de datos.

El procedimiento básico fue reescribir el material de cada capítulo, actualizando el material antiguo, añadiendo explicaciones sobre desarrollos recientes de la tecnología de las bases de datos y mejorando las descripciones de los temas que los estudiantes hallaban difíciles de comprender. Al igual que en la cuarta edición, cada capítulo tiene una lista de términos de repaso que puede ayudar a los lectores a repasar los temas principales que se han tratado en él. La mayor parte de los capítulos también contiene al final una sección que ofrece información sobre las herramientas de software relacionadas con el tema del capítulo. También se han añadido ejercicios nuevos y referencias actualizadas.

Nota para los profesores

Este libro contiene material tanto básico como avanzado, que puede que no se abarque en un solo semestre. Se han marcado varios apartados como avanzados mediante el símbolo $**$. Estos apartados pueden omitirse, si se desea, sin pérdida de continuidad. Los ejercicios que son difíciles (y pueden omitirse) también están marcados mediante el símbolo "$**$".

Es posible diseñar los cursos usando varios subconjuntos de los capítulos. A continuación se esbozan varias de las posibilidades:

- Los apartados del Capítulo 4 desde el Apartado 4.6 en adelante pueden omitirse en los cursos introductorios.
- El Capítulo 5 puede omitirse si los estudiantes no van a usar el cálculo relacional, QBE ni Datalog como parte del curso.
- Los Capítulos 9 (Bases de datos orientadas a objetos), 10 (XML) y 14 (Optimización de consultas) pueden omitirse en los cursos introductorios.
- Tanto el tratamiento del procesamiento de las transacciones (Capítulos 15 a 17) como el tratamiento de la arquitectura de los sistemas de bases de datos (Capítulos 20 a 22) constan de un capítulo introductorio (Capítulos 15 y 20, respectivamente) seguidos de los capítulos con los detalles. Se puede decidir usar los Capítulos 15 y 20 y omitir los Capítulos 16, 17, 21 y 22, si se posponen esos capítulos para un curso avanzado.
- Los Capítulos 18 y 19, que tratan de la minería de datos y de la recuperación de la información, pueden usarse como material para estudio individual u omitirse en los cursos introductorios.
- Los Capítulos 23 a 25 son adecuados para cursos avanzados o para su estudio individual por parte de los estudiantes.
- Los capítulos de estudio de casos 26 a 29 son adecuados para su estudio individual por los estudiantes.

Se pueden encontrar modelos de programaciones para cursos, basados en este texto, en la página Web del libro:

http://www.mhe.es/universidad/informatica/fundamentos

Cómo entrar en contacto con los autores y otros usuarios

Se ha hecho todo lo posible por eliminar del texto las erratas y errores. Pero, como en los nuevos desarrollos de software, probablemente queden fallos. En la página Web del libro se puede obtener una lista de erratas actualizada. Se agradecerá que se notifique cualquier error u omisión del libro que no se encuentre en la lista de erratas actual.

Nos alegrará recibir sugerencias de mejoras para el libro. También son bienvenidas las contribuciones para la página Web del libro que puedan resultar útiles para otros lectores, como ejercicios de programación, sugerencias de proyectos, laboratorios y tutoriales en línea, y ayudas a la docencia.

El correo electrónico debe dirigirse a db-book@cs.yale.edu. El resto de la correspondencia debe enviarse a Avi Silberschatz, Department of Computer Science, Yale University, 51 Prospect Street, P.O. Box 208285, New Haven, CT 06520-8285, EE.UU.

También se dispone de una lista de correo mediante la cual los usuarios del libro pueden comunicarse entre sí y con los autores, y recibir actualizaciones del libro y otra información relacionada. La lista está moderada, por lo que en ella no se recibe correo basura. Se ruega seguir el enlace a la lista de correo de la página Web del libro para suscribirse.

Agradecimientos

Esta edición se ha beneficiado de los muchos y útiles comentarios que nos han proporcionado los estudiantes que han usado las cuatro ediciones anteriores. Además, muchas personas nos han escrito o hablado acerca del libro, y nos han ofrecido sugerencias y comentarios. Aunque no podemos mencionarlos aquí a todos, damos las gracias especialmente a los siguientes:

- A Hani Abu-Salem, Universidad DePaul; Jamel R. Alsabbagh, Universidad Grand Valley State; Ramzi Bualuan, Universidad Notre Dame; Zhengxin Chen, Universidad de Nebraska en Omaha; Jan Chomick, Universidad SUNY Buffalo; Qin Ding, Universidad Penn State en Harrisburg; Frantisek Franek, Universidad McMaster; Shashi K. Gadia, Universidad Iowa State; William Hankley, Universidad Kansas State; Randy M. Kaplan, Universidad Drexel; Mark Llewellyn, Universidad

de Central Florida; Marty Maskarinec, Universidad Western Illinois; Yiu-Kai Dennis Ng, Universidad Brigham Young; Sunil Prabhakar, Universidad Purdue; Stewart Shen, Universidad Old Dominion; Anita Whitehall, Foothill College; Christopher Wilson, Universidad de Oregón; Weining Zhang, Universidad de Texas en San Antonio; todos ellos revisaron el libro y sus comentarios nos ayudaron mucho a formular esta quinta edición.

- A Anastassia Ailamaki, Sailesh Krishnamurthy, Spiros Papadimitriou y Bianca Schroeder (Universidad Carnegie Mellon) por escribir el apéndice que describe el sistema de bases de datos PostgreSQL.
- A Hakan Jakobsson (Oracle) por el apéndice sobre el sistema de bases de datos Oracle.
- A Sriram Padmanabhan (IBM) por el apéndice que describe el sistema de bases de datos DB2 de IBM.
- A Sameet Agarwal, José A. Blakeley, Thierry D'Hers, Gerald Hinson, Dirk Myers, Vaqar Pirzada, Bill Ramos, Balaji Rathakrishnan, Michael Rys, Florian Waas y Michael Zwilling (todos ellos de Microsoft) por el apéndice sobre el sistema de bases de datos Microsoft SQL Server. A José Blakeley también por coordinar y editar el Capítulo 29, y a César Galindo-Legaria, Goetz Graefe, Kalen Delaney y Thomas Casey (todos ellos de Microsoft) por sus aportaciones a la edición anterior del capítulo sobre SQL Server de Microsoft.
- A Chen Li y a Sharad Mehrotra por ofrecer material sobre JDBC y sobre seguridad que nos ha ayudado a actualizar y ampliar el Capítulo 8.
- A Valentin Dinu, Goetz Graefe, Bruce Hillyer, Chad Hogg, Nahid Rahman, Patrick Schmid, Jeff Storey, Prem Thomas, Liu Zhenming y, especialmente, a N.L. Sarda por su respuesta, que nos ayudó a preparar la quinta edición.
- A Rami Khouri, Nahid Rahman y Michael Rys por su respuesta a las versiones de borrador de los capítulos de la quinta edición.
- A Raj Ashar, Janek Bogucki, Gavin M. Bierman, Christian Breimann, Tom Chappell, Y. C. Chin, Laurens Damen, Prasanna Dhandapani, Arvind Hulgeri, Zheng Jiaping, Graham J. L. Kemp, Hae Choon Lee, Sang-Won Lee, Thanh-Duy Nguyen, D. B. Phatak, Juan Altmayer Pizzorno, Rajarshi Rakshit, Greg Riccardi, N. L. Sarda, Max Smolens, Nikhil Sethi y Tim Wahls por señalar errores en la cuarta edición.
- A Marilyn Turnamian, cuya excelente ayuda como secretaria fue esencial para la finalización a tiempo de esta quinta edición.

La editora fue Betsy Jones. La editora patrocinadora fue Kelly Lowery. La editora de desarrollo fue Melinda D. Bilecki. La directora del proyecto fue Peggy Selle. El director ejecutivo de mercadotecnia fue Michael Weitz. La directora de mercadotecnia fue Dawn Bercier. La ilustradora y diseñadora de la cubierta fue JoAnne Schopler. El editor de copia autónomo fue George Watson. La correctora de pruebas autónoma fue Judy Gantenbein. La diseñadora fue Laurie Janssen. El indexador autónomo fue Tobiah Waldron.

Esta edición se basa en las cuatro ediciones anteriores, por lo que volvemos a dar las gracias a las muchas personas que nos ayudaron con las cuatro primeras ediciones, como R. B. Abhyankar, Don Batory, Phil Bernhard, Haran Boral, Paul Bourgeois, Phil Bohannon, Robert Brazile, Yuri Breitbart, Michael Carey, Soumen Chakrabarti, J. Edwards, Christos Faloutsos, Homma Farian, Alan Fekete, Shashi Gadia, Jim Gray, Le Gruenwald, Eitan M. Gurari, Ron Hitchens, Yannis Ioannidis, Hyoung-Joo Kim, Won Kim, Henry Korth (padre de Henry F.), Carol Kroll, Gary Lindstrom, Irwin Levinstein, Ling Liu, Dave Maier, Keith Marzullo, Fletcher Mattox, Sharad Mehrotra, Jim Melton, Alberto Mendelzon, Hector Garcia-Molina, Ami Motro, Bhagirath Narahari, Anil Nigam, Cyril Orji, Meral Ozsoyoglu, Bruce Porter, Jim Peterson, K.V. Raghavan, Krithi Ramamritham, Mike Reiter, Odinaldo Rodriguez, Mark Roth, Marek Rusinkiewicz, Sunita Sarawagi, N.L. Sarda, S. Seshadri, Shashi Shekhar, Amit Sheth, Nandit Soparkar, Greg Speegle, Dilys Thomas y Marianne Winslett.

Marilyn Turnamian y Nandprasad Joshi ofrecieron ayuda como secretarias para la cuarta edición, y Marilyn también preparó un primer borrador del diseño de la cubierta para la cuarta edición. Lyn Dupré editó la copia de la tercera edición y Sara Strandtman editó el texto de la tercera edición. Nilesh Dalvi, Sumit Sanghai, Gaurav Bhalotia, Arvind Hulgeri K.V. Raghavan, Prateek Kapadia, Sara Strandtman, Greg Speegle y Dawn Bezviner ayudaron a preparar el manual para los profesores de ediciones anteriores. La nueva cubierta es una evolución de las cuatro primeras ediciones. La idea de usar barcos como parte del concepto de la cubierta nos la sugirió inicialmente Bruce Stephan.

Finalmente, Sudarshan quiere agradecer a su esposa, Sita, su amor y su apoyo, y a su hijo Madhur su amor. Hank quiere agradecer a su mujer, Joan, y a sus hijos, Abby y Joe, su amor y su comprensión. Avi quiere agradecer a Valerie su amor, su paciencia y su apoyo durante la revisión de este libro.

A. S.
H. F. K.
S. S.

CAPÍTULO 1

Introducción

Un **sistema gestor de bases de datos** (SGBD) consiste en una colección de datos interrelacionados y un conjunto de programas para acceder a dichos datos. La colección de datos, normalmente denominada **base de datos**, contiene información relevante para una empresa. El objetivo principal de un SGBD es proporcionar una forma de almacenar y recuperar la información de una base de datos de manera que sea tanto *práctica* como *eficiente*.

Los sistemas de bases de datos se diseñan para gestionar grandes cantidades de información. La gestión de los datos implica tanto la definición de estructuras para almacenar la información como la provisión de mecanismos para la manipulación de la información. Además, los sistemas de bases de datos deben garantizar la fiabilidad de la información almacenada, a pesar de las caídas del sistema o de los intentos de acceso no autorizados. Si los datos van a ser compartidos entre diferentes usuarios, el sistema debe evitar posibles resultados anómalos.

Dado que la información es tan importante en la mayoría de las organizaciones, los científicos informáticos han desarrollado una gran cuerpo de conceptos y técnicas para la gestión de los datos. Estos conceptos y técnicas constituyen el objetivo central de este libro. En este capítulo se presenta una breve introducción a los principios de los sistemas de bases de datos.

1.1 Aplicaciones de los sistemas de bases de datos

Las bases de datos se usan ampliamente. Algunas de sus aplicaciones representativas son:

- *Banca*: para información de los clientes, cuentas, préstamos y transacciones bancarias.
- *Líneas aéreas*: para reservas e información de horarios. Las líneas aéreas fueron de las primeras en usar las bases de datos de forma distribuida geográficamente.
- *Universidades*: para información de los estudiantes, matrículas en las asignaturas y cursos.
- *Transacciones de tarjetas de crédito*: para compras con tarjeta de crédito y la generación de los extractos mensuales.
- *Telecomunicaciones*: para guardar un registro de las llamadas realizadas, generar las facturas mensuales, mantener el saldo de las tarjetas telefónicas de prepago y para almacenar información sobre las redes de comunicaciones.
- *Finanzas*: para almacenar información sobre compañías tenedoras, ventas y compras de productos financieros, como acciones y bonos; también para almacenar datos del mercado en tiempo real para permitir a los clientes la compraventa en línea y a la compañía la compraventa automática.
- *Ventas*: para información de clientes, productos y compras.

- *Comercio en línea*: para los datos de ventas ya mencionados y para el seguimiento de los pedidos Web, generación de listas de recomendaciones y mantenimiento de evaluaciones de productos en línea.
- *Producción*: para la gestión de la cadena de proveedores y para el seguimiento de la producción de artículos en las factorías, inventarios en los almacenes y pedidos.
- *Recursos humanos*: para información sobre los empleados, salarios, impuestos sobre los sueldos y prestaciones sociales, y para la generación de las nóminas.

Como muestra esta lista, las bases de datos forman una parte esencial de casi todas las empresas actuales.

Durante las últimas cuatro décadas del siglo veinte, el uso de las bases de datos creció en todas las empresas. En los primeros días, muy pocas personas interactuaban directamente con los sistemas de bases de datos, aunque sin darse cuenta interactuaban indirectamente con bases de datos—con informes impresos como los extractos de las tarjetas de crédito, o mediante agentes como los cajeros de los bancos y los agentes de reservas de las líneas aéreas. Después vinieron los cajeros automáticos y permitieron a los usuarios interactuar directamente con las bases de datos. Las interfaces telefónicas con las computadoras (sistemas de respuesta vocal interactiva) también permitieron a los usuarios tratar directamente con las bases de datos—la persona que llamaba podía marcar un número y pulsar las teclas del teléfono para introducir información o para seleccionar opciones alternativas, para conocer las horas de llegada o salida de los vuelos, por ejemplo, o para matricularse de asignaturas en una universidad.

La revolución de Internet a finales de los años noventa aumentó significativamente el acceso directo del usuario a las bases de datos. Las organizaciones convirtieron muchas de sus interfaces telefónicas a las bases de datos en interfaces Web, y dejaron disponibles en línea muchos servicios. Por ejemplo, cuando se accede a una librería en línea y se busca en una colección de libros o de música, se está accediendo a datos almacenados en una base de datos. Cuando se realiza un pedido en línea, el pedido se almacena en una base de datos. Cuando se accede al sitio Web de un banco y se consultan el estado de la cuenta y los movimientos, la información se recupera del sistema de bases de datos del banco. Cuando se accede a un sitio Web, puede que se recupere información personal de una base de datos para seleccionar los anuncios que se deben mostrar. Más aún, los datos sobre los accesos Web pueden almacenarse en una base de datos

Así, aunque las interfaces de usuario ocultan los detalles del acceso a las bases de datos, y la mayoría de la gente ni siquiera es consciente de que están interactuando con una base de datos, el acceso a las bases de datos forma actualmente una parte esencial de la vida de casi todas las personas.

La importancia de los sistemas de bases de datos se puede juzgar de otra forma—actualmente, los fabricantes de sistemas de bases de datos como Oracle están entre las mayores compañías de software del mundo, y los sistemas de bases de datos forman una parte importante de la línea de productos de compañías más diversificadas como Microsoft e IBM.

1.2 Propósito de los sistemas de bases de datos

Los sistemas de bases de datos surgieron en respuesta a los primeros métodos de gestión informatizada de los datos comerciales. A modo de ejemplo de dichos métodos, típicos de los años sesenta, considérese parte de una entidad bancaria que, entre otros datos, guarda información sobre todos los clientes y todas las cuentas de ahorro. Una manera de guardar la información en la computadora es almacenarla en archivos del sistema operativo. Para permitir que los usuarios manipulen la información, el sistema tiene varios programas de aplicación que gestionan los archivos, incluyendo programas para:

- Efectuar cargos o abonos en las cuentas.
- Añadir cuentas nuevas.
- Calcular el saldo de las cuentas.
- Generar los extractos mensuales.

Estos programas de aplicación los han escrito programadores de sistemas en respuesta a las necesidades del banco.

Se añaden nuevos programas de aplicación al sistema según surgen las necesidades. Por ejemplo, supóngase que una caja de ahorros decide ofrecer cuentas corrientes. En consecuencia, se crean nuevos archivos permanentes que contienen información acerca de todas las cuentas corrientes abiertas en el banco y puede que haya que escribir nuevos programas de aplicación para afrontar situaciones que no se dan en las cuentas de ahorro, como los descubiertos. Así, con el paso del tiempo, se añaden más archivos y programas de aplicación al sistema.

Los sistemas operativos convencionales soportan este **sistema de procesamiento de archivos** típico. El sistema almacena los registros permanentes en varios archivos y necesita diferentes programas de aplicación para extraer y añadir a los archivos correspondientes. Antes de la aparición de los sistemas gestores de bases de datos (SGBDs), las organizaciones normalmente almacenaban la información en sistemas de este tipo.

Guardar la información de la organización en un sistema de procesamiento de archivos tiene una serie de inconvenientes importantes:

- **Redundancia e inconsistencia de los datos**. Debido a que los archivos y programas de aplicación los crean diferentes programadores en el transcurso de un largo período de tiempo, es probable que los diversos archivos tengan estructuras diferentes y que los programas estén escritos en varios lenguajes de programación diferentes. Además, puede que la información esté duplicada en varios lugares (archivos). Por ejemplo, la dirección y el número de teléfono de un cliente dado pueden aparecer en un archivo que contenga registros de cuentas de ahorros y en un archivo que contenga registros de cuentas corrientes. Esta redundancia conduce a costes de almacenamiento y de acceso más elevados. Además, puede dar lugar a la **inconsistencia de los datos**; es decir, puede que las diferentes copias de los mismos datos no coincidan. Por ejemplo, puede que el cambio en la dirección de un cliente esté reflejado en los registros de las cuentas de ahorro pero no en el resto del sistema.

- **Dificultad en el acceso a los datos**. Supóngase que uno de los empleados del banco necesita averiguar los nombres de todos los clientes que viven en un código postal dado. El empleado pide al departamento de procesamiento de datos que genere esa lista. Debido a que esta petición no fue prevista por los diseñadores del sistema original, no hay un programa de aplicación a mano para satisfacerla. Hay, sin embargo, un programa de aplicación que genera la lista de todos los clientes. El empleado del banco tiene ahora dos opciones: bien obtener la lista de *todos* los clientes y extraer manualmente la información que necesita, o bien pedir a un programador de sistemas que escriba el programa de aplicación necesario. Ambas alternativas son obviamente insatisfactorias. Supóngase que se escribe el programa y que, varios días más tarde, el mismo empleado necesita reducir esa lista para que incluya únicamente a aquellos clientes que tengan una cuenta con saldo igual o superior a 10.000 €. Como se puede esperar, no existe ningún programa que genere tal lista. De nuevo, el empleado tiene que elegir entre dos opciones, ninguna de las cuales es satisfactoria.

  La cuestión aquí es que los entornos de procesamiento de archivos convencionales no permiten recuperar los datos necesarios de una forma práctica y eficiente. Hacen falta sistemas de recuperación de datos más adecuados para el uso general.

- **Aislamiento de datos.** Como los datos están dispersos en varios archivos, y los archivos pueden estar en diferentes formatos, es difícil escribir nuevos programas de aplicación para recuperar los datos correspondientes.

- **Problemas de integridad.** Los valores de los datos almacenados en la base de datos deben satisfacer ciertos tipos de **restricciones de consistencia**. Por ejemplo, el saldo de ciertos tipos de cuentas bancarias no puede nunca ser inferior a una cantidad predeterminada (por ejemplo, 25 €). Los desarrolladores hacen cumplir esas restricciones en el sistema añadiendo el código correspondiente en los diversos programas de aplicación. Sin embargo, cuando se añaden nuevas restricciones, es difícil cambiar los programas para hacer que se cumplan. El problema se complica cuando las restricciones implican diferentes elementos de datos de diferentes archivos.

- **Problemas de atomicidad.** Los sistemas informáticos, como cualquier otro dispositivo mecánico o eléctrico, está sujeto a fallos. En muchas aplicaciones es crucial asegurar que, si se produce algún fallo, los datos se restauren al estado consistente que existía antes del fallo. Considérese un programa para transferir 50 € desde la cuenta *A* a la *B*. Si se produce un fallo del sistema durante la ejecución del programa, es posible que los 50 € fueran retirados de la cuenta *A* pero no abonados en la cuenta *B*, dando lugar a un estado inconsistente de la base de datos. Evidentemente, resulta esencial para la consistencia de la base de datos que tengan lugar tanto el abono como el cargo, o que no tenga lugar ninguno. Es decir, la transferencia de fondos debe ser *atómica*—debe ocurrir en su totalidad o no ocurrir en absoluto. Resulta difícil asegurar la atomicidad en los sistemas convencionales de procesamiento de archivos.

- **Anomalías en el acceso concurrente**. Para aumentar el rendimiento global del sistema y obtener una respuesta más rápida, muchos sistemas permiten que varios usuarios actualicen los datos simultáneamente. En realidad, hoy en día, los principales sitios de comercio electrónico de Internet pueden tener millones de accesos diarios de compradores a sus datos. En tales entornos es posible la interacción de actualizaciones concurrentes y puede dar lugar a datos inconsistentes. Considérese una cuenta bancaria *A*, que contenga 500 €. Si dos clientes retiran fondos (por ejemplo, 50 € y 100 €, respectivamente) de la cuenta *A* aproximadamente al mismo tiempo, el resultado de las ejecuciones concurrentes puede dejar la cuenta en un estado incorrecto (o inconsistente). Supóngase que los programas que se ejecutan para cada retirada leen el saldo anterior, reducen su valor en el importe que se retira y luego escriben el resultado. Si los dos programas se ejecutan concurrentemente, pueden leer el valor 500 €, y escribir después 450 € y 400 €, respectivamente. Dependiendo de cuál escriba el valor en último lugar, la cuenta puede contener 450 € o 400 €, en lugar del valor correcto, 350 €. Para protegerse contra esta posibilidad, el sistema debe mantener alguna forma de supervisión. Pero es difícil ofrecer supervisión, ya que muchos programas de aplicación diferentes que no se han coordinado con anterioridad pueden tener acceso a los datos.

- **Problemas de seguridad**. No todos los usuarios de un sistema de bases de datos deben poder acceder a todos los datos. Por ejemplo, en un sistema bancario, el personal de nóminas sólo necesita ver la parte de la base de datos que contiene información acerca de los diferentes empleados del banco. No necesitan tener acceso a la información acerca de las cuentas de clientes. Pero, como los programas de aplicación se añaden al sistema de procesamiento de datos de una forma ad hoc, es difícil hacer cumplir tales restricciones de seguridad.

Estas dificultades, entre otras, motivaron el desarrollo de los sistemas de bases de datos. En el resto del libro se examinarán los conceptos y los algoritmos que permiten que los sistemas de bases de datos resuelvan los problemas de los sistemas de procesamiento de archivos. En general, en este libro se usa una entidad bancaria como ejemplo de aplicación típica de procesamiento de datos que puede encontrarse en una empresa.

1.3 Visión de los datos

Un sistema de bases de datos es una colección de datos interrelacionados y un conjunto de programas que permiten a los usuarios tener acceso a esos datos y modificarlos. Una de las principales finalidades de los sistemas de bases de datos es ofrecer a los usuarios una visión *abstracta* de los datos. Es decir, el sistema oculta ciertos detalles del modo en que se almacenan y mantienen los datos.

1.3.1 Abstracción de datos

Para que el sistema sea útil debe recuperar los datos eficientemente. La necesidad de eficiencia ha llevado a los diseñadores a usar estructuras de datos complejas para la representación de los datos en la base de datos. Dado que muchos de los usuarios de sistemas de bases de datos no tienen formación en informática, los desarrolladores ocultan esa complejidad a los usuarios mediante varios niveles de abstracción para simplificar la interacción de los usuarios con el sistema:

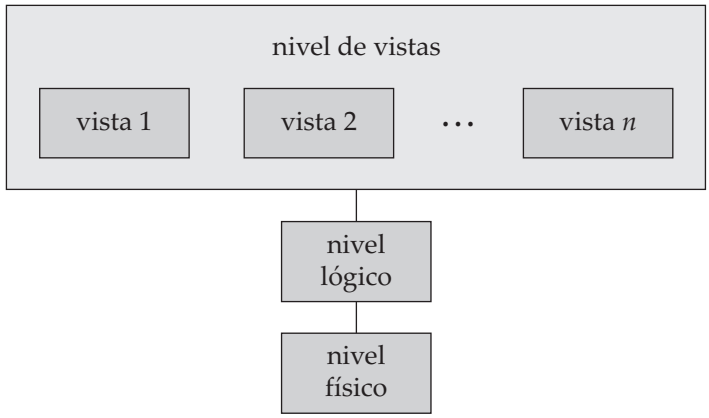


Figura 1.1 Los tres niveles de abstracción de datos.

- **Nivel físico**. El nivel más bajo de abstracción describe *cómo* se almacenan realmente los datos. El nivel físico describe en detalle las estructuras de datos complejas de bajo nivel.
- **Nivel lógico**. El nivel inmediatamente superior de abstracción describe *qué* datos se almacenan en la base de datos y qué relaciones existen entre esos datos. El nivel lógico, por tanto, describe toda la base de datos en términos de un número pequeño de estructuras relativamente simples. Aunque la implementación de esas estructuras simples en el nivel lógico puede involucrar estructuras complejas del nivel físico, los usuarios del nivel lógico no necesitan preocuparse de esta complejidad. Los administradores de bases de datos, que deben decidir la información que se guarda en la base de datos, usan el nivel de abstracción lógico.
- **Nivel de vistas**. El nivel más elevado de abstracción sólo describe parte de la base de datos. Aunque el nivel lógico usa estructuras más simples, queda algo de complejidad debido a la variedad de información almacenada en las grandes bases de datos. Muchos usuarios del sistema de bases de datos no necesitan toda esta información; en su lugar sólo necesitan tener acceso a una parte de la base de datos. El nivel de abstracción de vistas existe para simplificar su interacción con el sistema. El sistema puede proporcionar muchas vistas para la misma base de datos.

La Figura 1.1 muestra la relación entre los tres niveles de abstracción.

Una analogía con el concepto de tipos de datos en lenguajes de programación puede clarificar la diferencia entre los niveles de abstracción. La mayoría de los lenguajes de programación de alto nivel soportan el concepto de tipo estructurado. Por ejemplo, en los lenguajes tipo Pascal se pueden declarar registros de la manera siguiente:

```
type cliente = record
                 id_cliente : string;
                 nombre_cliente : string;
                 calle_cliente : string;
                 ciudad_cliente : string;
               end;
```

Este código define un nuevo tipo de registro denominado *cliente* con cuatro campos. Cada campo tiene un nombre y un tipo asociados. Una entidad bancaria puede tener varios tipos de estos registros, incluidos:

- *cuenta*, con los campos *número_cuenta* y *saldo*.
- *empleado*, con los campos *nombre_empleado* y *sueldo*.

En el nivel físico, los registros *cliente*, *cuenta* o *empleado* se pueden describir como bloques de posiciones consecutivas de almacenamiento (por ejemplo, palabras o bytes). El compilador oculta este nivel

de detalle a los programadores. De manera parecida, el sistema de base de datos oculta muchos de los detalles de almacenamiento de los niveles inferiores a los programadores de bases de datos. Los administradores de bases de datos, por otro lado, pueden ser conscientes de ciertos detalles de la organización física de los datos.

En el nivel lógico cada registro de este tipo se describe mediante una definición de tipo, como en el fragmento de código anterior, y también se define la relación entre estos tipos de registros. Los programadores que usan un lenguaje de programación trabajan en este nivel de abstracción. De manera parecida, los administradores de bases de datos suelen trabajar en este nivel de abstracción.

Finalmente, en el nivel de vistas, los usuarios de computadoras ven un conjunto de programas de aplicación que ocultan los detalles de los tipos de datos. De manera parecida, en el nivel de vistas se definen varias vistas de la base de datos y los usuarios de la base de datos pueden verlas. Además de ocultar los detalles del nivel lógico de la base de datos, las vistas también proporcionan un mecanismo de seguridad para evitar que los usuarios tengan acceso a ciertas partes de la base de datos. Por ejemplo, los cajeros de un banco sólo ven la parte de la base de datos que contiene información de las cuentas de los clientes; no pueden tener acceso a la información referente a los sueldos de los empleados.

1.3.2 Ejemplares y esquemas

Las bases de datos van cambiando a lo largo del tiempo conforme la información se inserta y se elimina. La colección de información almacenada en la base de datos en un momento dado se denomina **ejemplar** de la base de datos. El diseño general de la base de datos se denomina **esquema** de la base de datos. Los esquemas se modifican rara vez, si es que se modifican.

El concepto de esquemas y ejemplares de las bases de datos se puede comprender por analogía con los programas escritos en un lenguaje de programación. El esquema de la base de datos se corresponde con las declaraciones de las variables (junto con las definiciones de tipos asociadas) de los programas. Cada variable tiene un valor concreto en un instante dado. Los valores de las variables de un programa en un instante dado se corresponden con un *ejemplar* del esquema de la base de datos.

Los sistemas de bases de datos tiene varios esquemas divididos según los niveles de abstracción. El **esquema físico** describe el diseño de la base de datos en el nivel físico, mientras que el **esquema lógico** describe su diseño en el nivel lógico. Las bases de datos también pueden tener varios esquemas en el nivel de vistas, a veces denominados **subesquemas**, que describen diferentes vistas de la base de datos.

De éstos, el esquema lógico es con mucho el más importante en términos de su efecto sobre los programas de aplicación, ya que los programadores crean las aplicaciones usando el esquema lógico. El esquema físico está oculto bajo el esquema lógico, y generalmente puede modificarse fácilmente sin afectar a los programas de aplicación. Se dice que los programas de aplicación muestran **independencia!física respecto de los datos** si no dependen del esquema físico y, por tanto, no hace falta volver a escribirlos si se modifica el esquema físico.

Se estudiarán los lenguajes para la descripción de los esquemas, después de introducir el concepto de modelos de datos en el apartado siguiente.

1.3.3 Modelos de datos

Bajo la estructura de las bases de datos se encuentra el **modelo de datos**: una colección de herramientas conceptuales para describir los datos, sus relaciones, su semántica y las restricciones de consistencia. Los modelos de datos ofrecen un modo de describir el diseño de las bases de datos en los niveles físico, lógico y de vistas.

En este texto se van a tratar varios modelos de datos diferentes. Los modelos de datos pueden clasificarse en cuatro categorías diferentes:

- **Modelo relacional**. El modelo relacional usa una colección de tablas para representar tanto los datos como sus relaciones. Cada tabla tiene varias columnas, y cada columna tiene un nombre único. El modelo relacional es un ejemplo de un modelo basado en registros. Los modelos basados en registros se denominan así porque la base de datos se estructura en registros de formato fijo de varios tipos. Cada tabla contiene registros de un tipo dado. Cada tipo de registro define un número fijo de campos, o atributos. Las columnas de la tabla se corresponden con los atributos

del tipo de registro. El modelo de datos relacional es el modelo de datos más ampliamente usado, y una gran mayoría de sistemas de bases de datos actuales se basan en el modelo relacional. Los Capítulos 2 al 7 tratan el modelo relacional en detalle.

- **El modelo entidad-relación**. El modelo de datos entidad-relación (E-R) se basa en una percepción del mundo real que consiste en una colección de objetos básicos, denominados *entidades*, y de las *relaciones* entre ellos. Una entidad es una "cosa" u "objeto" del mundo real que es distinguible de otros objetos. El modelo entidad-relación se usa mucho en el diseño de bases de datos y en el Capítulo 6 se examina detalladamente.
- **Modelo de datos orientado a objetos**. El modelo de datos orientado a objetos es otro modelo de datos que está recibiendo una atención creciente. El modelo orientado a objetos se puede considerar como una extensión del modelo E-R con los conceptos de la encapsulación, los métodos (funciones) y la identidad de los objetos. En el Capítulo 9 se examina este modelo de datos.
- **Modelo de datos semiestructurados**. El modelo de datos semiestructurados permite la especificación de datos donde los elementos de datos individuales del mismo tipo pueden tener diferentes conjuntos de atributos. Esto lo diferencia de los modelos de datos mencionados anteriormente, en los que cada elemento de datos de un tipo particular debe tener el mismo conjunto de atributos. El **lenguaje de marcas extensible** (XML, eXtensible Markup Language) se emplea mucho para representar datos semiestructurados. Se estudia en el Capítulo 10.

El **modelo de datos de red** y el **modelo de datos jerárquico** precedieron cronológicamente al relacional. Estos modelos estuvieron íntimamente ligados a la implementación subyacente y complicaban la tarea del modelado de datos. En consecuencia, se usan muy poco hoy en día, excepto en el código de bases de datos antiguas que sigue estando en servicio en algunos lugares. Se describen brevemente en los Apéndices A y B para los lectores interesados.

1.4 Lenguajes de bases de datos

Los sistemas de bases de datos proporcionan un **lenguaje de definición de datos** para especificar el esquema de la base de datos y un **lenguaje de manipulación de datos** para expresar las consultas y las modificaciones de la base de datos. En la práctica, los lenguajes de definición y manipulación de datos no son dos lenguajes diferentes; en cambio, simplemente forman parte de un único lenguaje de bases de datos, como puede ser el muy usado SQL.

1.4.1 Lenguaje de manipulación de datos

Un **lenguaje de manipulación de datos** (LMD) es un lenguaje que permite a los usuarios tener acceso a los datos organizados mediante el modelo de datos correspondiente o manipularlos. Los tipos de acceso son:

- La recuperación de la información almacenada en la base de datos.
- La inserción de información nueva en la base de datos.
- El borrado de la información de la base de datos.
- La modificación de la información almacenada en la base de datos.

Hay fundamentalmente dos tipos:

- Los **LMDs procedimentales** necesitan que el usuario especifique *qué* datos se necesitan y *cómo* obtener esos datos.
- Los **LMDs declarativos** (también conocidos como **LMDs no procedimentales**) necesitan que el usuario especifique *qué* datos se necesitan *sin* que haga falta que especifique cómo obtener esos datos.

Los LMDs declarativos suelen resultar más fáciles de aprender y de usar que los procedimentales. Sin embargo, como el usuario no tiene que especificar cómo conseguir los datos, el sistema de bases de datos tiene que determinar un medio eficiente de acceso a los datos.

Una **consulta** es una instrucción que solicita que se recupere información. La parte de los LMDs implicada en la recuperación de información se denomina **lenguaje de consultas**. Aunque técnicamente sea incorrecto, resulta habitual usar las expresiones *lenguaje de consultas* y *lenguaje de manipulación de datos* como sinónimas.

Existen varios lenguajes de consultas de bases de datos en uso, tanto comercial como experimentalmente. El lenguaje de consultas más ampliamente usado, SQL, se estudiará en los Capítulos 3 y 4. También se estudiarán otros lenguajes de consultas en el Capítulo 5.

Los niveles de abstracción que se trataron en el Apartado 1.3 no sólo se aplican a la definición o estructuración de datos, sino también a su manipulación. En el nivel físico se deben definir los algoritmos que permitan un acceso eficiente a los datos. En los niveles superiores de abstracción se pone el énfasis en la facilidad de uso. El objetivo es permitir que los seres humanos interactúen de manera eficiente con el sistema. El componente procesador de consultas del sistema de bases de datos (que se estudia en los Capítulos 13 y 14) traduce las consultas LMD en secuencias de acciones en el nivel físico del sistema de bases de datos.

1.4.2 Lenguaje de definición de datos

Los esquemas de las bases de datos se especifican mediante un conjunto de definiciones expresadas mediante un lenguaje especial denominado **lenguaje de definición de datos (LDD)**. El LDD también se usa para especificar más propiedades de los datos.

La estructura de almacenamiento y los métodos de acceso usados por el sistema de bases de datos se especifican mediante un conjunto de instrucciones en un tipo especial de LDD denominado **lenguaje de almacenamiento y definición de datos**. Estas instrucciones definen los detalles de implementación de los esquemas de las bases de datos, que suelen ocultarse a los usuarios.

Los valores de los datos almacenados en la base de datos deben satisfacer ciertas **restricciones de consistencia**. Por ejemplo, supóngase que el saldo de una cuenta no debe caer por debajo de 100 €. El LDD proporciona facilidades para especificar tales restricciones. Los sistemas de bases de datos las comprueban cada vez que se modifica la base de datos. En general, las restricciones pueden ser predicados arbitrarios relativos a la base de datos. No obstante, los predicados arbitrarios pueden resultar costosos de comprobar. Por tanto, los sistemas de bases de datos se concentran en las restricciones de integridad que pueden comprobarse con una sobrecarga mínima:

- **Restricciones de dominio.** Se debe asociar un dominio de valores posibles a cada atributo (por ejemplo, tipos enteros, tipos de carácter, tipos fecha/hora). La declaración de un atributo como parte de un dominio concreto actúa como restricción de los valores que puede adoptar. Las restricciones de dominio son la forma más elemental de restricción de integridad. El sistema las comprueba fácilmente siempre que se introduce un nuevo elemento de datos en la base de datos.
- **Integridad referencial.** Hay casos en los que se desea asegurar que un valor que aparece en una relación para un conjunto de atributos dado aparece también para un determinado conjunto de atributos en otra relación (integridad referencial). Las modificaciones de la base de datos pueden causar violaciones de la integridad referencial. Cuando se viola una restricción de integridad, el procedimiento normal es rechazar la acción que ha causado esa violación.
- **Asertos.** Un aserto es cualquier condición que la base de datos debe satisfacer siempre. Las restricciones de dominio y las restricciones de integridad referencial son formas especiales de asertos. No obstante, hay muchas restricciones que no pueden expresarse empleando únicamente esas formas especiales. Por ejemplo: "Cada préstamo tiene como mínimo un cliente tenedor de una cuenta con un saldo mínimo de 1.000,00 €" debe expresarse en forma de aserto. Cuando se crea un aserto, el sistema comprueba su validez. Si el aserto es válido, cualquier modificación futura de la base de datos se permite únicamente si no hace que se viole ese aserto.
- **Autorización.** Puede que se desee diferenciar entre los usuarios en cuanto al tipo de acceso que se les permite a diferentes valores de los datos de la base de datos. Estas diferenciaciones se

expresan en términos de **autorización**, cuyas modalidades más frecuentes son: **autorización de lectura**, que permite la lectura pero no la modificación de los datos; **autorización de inserción**, que permite la inserción de datos nuevos, pero no la modificación de los datos ya existentes; **autorización de actualización**, que permite la modificación, pero no la eliminación, de los datos; y la **autorización de eliminación**, que permite la eliminación de datos. A cada usuario se le pueden asignar todos, ninguno o una combinación de estos tipos de autorización.

El LDD, al igual que cualquier otro lenguaje de programación, obtiene como entrada algunas instrucciones y genera una salida. La salida del LDD se coloca en el **diccionario de datos**, que contiene **metadatos**—es decir, datos sobre datos. El diccionario de datos se considera un tipo especial de tabla, a la que sólo puede tener acceso y actualizar el propio sistema de bases de datos (no los usuarios normales). El sistema de bases de datos consulta el diccionario de datos antes de leer o modificar los datos reales.

1.5 Bases de datos relacionales

Las bases de datos relacionales se basan en el modelo relacional y usan un conjunto de tablas para representar tanto los datos como las relaciones entre ellos. También incluyen un LMD y un LDD. La mayor parte de los sistemas de bases de datos relacionales comerciales emplean el lenguaje SQL, que se trata en este apartado y que se tratará con gran detalle en los Capítulos 3 y 4. En el Capítulo 5 se estudiarán otros lenguajes influyentes.

1.5.1 Tablas

Cada tabla tiene varias columnas, y cada columna tiene un nombre único. En la Figura 1.2 se presenta un ejemplo de base de datos relacional consistente en tres tablas: una muestra detalles de los clientes de un banco, la segunda muestra las cuentas y la tercera muestra las cuentas que pertenecen a cada cliente.

La primera tabla, la tabla *cliente*, muestra, por ejemplo, que el cliente identificado por *id_cliente* 19.283.746 se llama González y vive en la calle Arenal en La Granja. La segunda tabla, *cuenta*, muestra, por ejemplo, que la cuenta C-101 tiene un saldo de 500 € y la C-201 un saldo de 900 €.

La tercera tabla muestra las cuentas que pertenecen a cada cliente. Por ejemplo, la cuenta C-101 pertenece al cliente cuyo *id_cliente* es 19.283.746 (González), y los clientes 19.283.746 (González) y 01.928.374 (Gómez) comparten el número de cuenta C-201 (pueden compartir un negocio).

El modelo relacional es un ejemplo de modelo basado en registros. Los modelos basados en registros se denominan así porque la base de datos se estructura en registros de formato fijo de varios tipos. Cada tabla contiene registros de un tipo dado. Cada tipo de registro define un número fijo de campos, o atributos. Las columnas de la tabla se corresponden con los atributos del tipo de registro.

No es difícil ver cómo se pueden almacenar las tablas en archivos. Por ejemplo, se puede usar un carácter especial (como la coma) para delimitar los diferentes atributos de un registro, y otro carácter especial (como el carácter de nueva línea) para delimitar los registros. El modelo relacional oculta esos detalles de implementación de bajo nivel a los desarrolladores de bases de datos y a los usuarios.

El modelo de datos relacional es el modelo de datos más ampliamente usado, y una gran mayoría de los sistemas de bases de datos actuales se basan en el modelo relacional. Los Capítulos 2 al 7 tratan el modelo relacional con detalle.

Obsérvese también que en el modelo relacional es posible crear esquemas que tengan problemas tales como información duplicada innecesariamente. Por ejemplo, supóngase que se almacena *número_cuenta* como atributo de un registro *cliente*. Entonces, para representar el hecho de que tanto la cuentas C-101 como la cuenta C-201 pertenece al cliente González (con *id_cliente* 19.283.746) sería necesario almacenar dos filas en la tabla *cliente*. Los valores de *nombre_cliente*, *calle_cliente* y *ciudad_cliente* de González estarían innecesariamente duplicados en las dos filas. En el Capítulo 7 se estudiará el modo de distinguir los buenos diseños de esquema de los malos.

id_cliente	*nombre_cliente*	*calle_cliente*	*ciudad_cliente*
19.283.746	González	Arenal, 12	La Granja
67.789.901	López	Mayor, 3	Peguerinos
18.273.609	Abril	Preciados, 123	Valsaín
32.112.312	Santos	Mayor, 100	Peguerinos
33.666.999	Rupérez	Ramblas, 175	León
01.928.374	Gómez	Carretas, 72	Cerceda

(a) La tabla *cliente*

número_cuenta	*saldo*
C-101	500
C-215	700
C-102	400
C-305	350
C-201	900
C-217	750
C-222	700

(b) La tabla *cuenta*

id_cliente	*número_cuenta*
19.283.746	C-101
19.283.746	C-201
01.928.374	C-215
67.789.901	C-102
18.273.609	C-305
32.112.312	C-217
33.666.999	C-222
01.928.374	C-201

(c) La tabla *impositor*

Figura 1.2 Un ejemplo de base de datos relacional.

1.5.2 Lenguaje de manipulación de datos

El lenguaje de consultas de SQL no es procedimental. Usa como entrada varias tablas (posiblemente sólo una) y devuelve siempre una sola tabla. A continuación se ofrece un ejemplo de consulta SQL que halla el nombre de todos los clientes que residen en Peguerinos:

select *cliente.nombre_cliente*
from *cliente*
where *cliente.ciudad_cliente* = 'Peguerinos'

La consulta especifica que hay que recuperar (*select*) las filas de (*from*) la tabla *cliente* en las que (*where*) la *ciudad_cliente* es Peguerinos, y que sólo debe mostrarse el atributo *nombre_cliente* de esas filas. Más concretamente, el resultado de la ejecución de esta consulta es una tabla con una sola columna denominada *nombre_cliente* y un conjunto de filas, cada una de las cuales contiene el nombre de un cliente cuya *ciudad_cliente* es Peguerinos. Si la consulta se ejecuta sobre la tabla de la Figura 1.2, el resultado constará de dos filas, una con el nombre López y otra con el nombre Santos.

Las consultas pueden involucrar información de más de una tabla. Por ejemplo, la siguiente consulta busca todos los números de cuenta y sus saldos del cliente con *id_cliente* 19.283.746.

```
select cuenta.número_cuenta, cuenta.saldo
from impositor, cuenta
where impositor.id_cliente = '19.283.746' and
      impositor.número_cuenta = cuenta.número_cuenta
```

Si la consulta anterior se ejecutase sobre las tablas de la Figura 1.2, el sistema encontraría que las dos cuentas denominadas C-101 y C-201 pertenecen al cliente 19.283.746 y el resultado consistiría en una tabla con dos columnas (*número_cuenta, saldo*) y dos filas (C-101, 500) y (C-201,900).

1.5.3 Lenguaje de definición de datos

SQL ofrece un LDD elaborado que permite definir tablas, restricciones de integridad, asertos, etc.

Por ejemplo, la siguiente instrucción del lenguaje SQL define la tabla *cuenta*:

```
create table cuenta
      (número_cuenta char(10),
       saldo integer)
```

La ejecución de esta instrucción LDD crea la tabla *cuenta.* Además, actualiza el diccionario de datos, que contiene metadatos (1.4.2). Los esquemas de las tablas son ejemplos de metadatos.

1.5.4 Acceso a las bases de datos desde los programas de aplicación

SQL no es tan potente como la máquina universal de Turing; es decir, hay algunos cálculos que no pueden obtenerse mediante ninguna consulta SQL. Esos cálculos deben escribirse en un lenguaje *anfitrión*, como Cobol, C, C++ o Java. Los **programas de aplicación** son programas que se usan para interactuar de esta manera con las bases de datos. Algunos de los ejemplos de un sistema bancario serían los programas que generan las nóminas, realizan cargos en las cuentas, realizan abonos en las cuentas o transfieren fondos entre las cuentas.

Para tener acceso a la base de datos, las instrucciones LMD deben ejecutarse desde el lenguaje anfitrión. Hay dos maneras de conseguirlo:

- Proporcionando una interfaz de programas de aplicación (conjunto de procedimientos) que se pueda usar para enviar instrucciones LMD y LDD a la base de datos y recuperar los resultados.

  El estándar de conectividad abierta de bases de datos (ODBC, Open Data Base Connectivity) definido por Microsoft para su empleo con el lenguaje C es un estándar de interfaz de programas de aplicación usado habitualmente. El estándar de conectividad de Java con bases de datos (JDBC, Java Data Base Connectivity) ofrece las características correspondientes para el lenguaje Java.

- Extendiendo la sintaxis del lenguaje anfitrión para que incorpore las llamadas LMD dentro del programa del lenguaje anfitrión. Generalmente, un carácter especial precede a las llamadas LMD y un preprocesador, denominado **precompilador LMD**, convierte las instrucciones LMD en llamadas normales a procedimientos en el lenguaje anfitrión.

1.6 Diseño de bases de datos

Los sistemas de bases de datos se diseñan para gestionar grandes cantidades de información. Esas grandes cantidades de información no existen aisladas. Forman parte del funcionamiento de alguna empresa, cuyo producto final puede que sea la información obtenida de la base de datos o algún dispositivo o servicio para el que la base de datos sólo desempeña un papel secundario.

El diseño de bases de datos implica principalmente el diseño del esquema de las bases de datos. El diseño de un entorno completo de aplicaciones para la base de datos que satisfaga las necesidades de la empresa que se está modelando exige prestar atención a un conjunto de aspectos más amplio. Este texto se centrará inicialmente en la escritura de las consultas a la base de datos y en el diseño de los esquemas de las bases de datos. En el Capítulo 8 se estudia el proceso general de diseño de las aplicaciones.

1.6.1 Proceso de diseño

Los modelos de datos de alto nivel resultan útiles a los diseñadores de bases de datos al ofrecerles un marco conceptual en el que especificar, de manera sistemática, los requisitos de datos de los usuarios de las bases de datos y la manera en que se estructurará la base de datos para satisfacer esos requisitos. La fase inicial del diseño de las bases de datos, por tanto, es caracterizar completamente los requisitos de datos de los hipotéticos usuarios de la base de datos. Los diseñadores de bases de datos deben interactuar ampliamente con los expertos y usuarios del dominio para llevar a cabo esta tarea. El resultado de esta fase es la especificación de los requisitos de los usuarios.

A continuación, el diseñador escoge un modelo de datos y, mediante la aplicación de los conceptos del modelo de datos elegido, traduce esos requisitos en un esquema conceptual de la base de datos. El esquema desarrollado en esta fase de **diseño conceptual** ofrece una visión general detallada de la empresa. El diseñador revisa el esquema para confirmar que todos los requisitos de datos se satisfacen realmente y no entran en conflicto entre sí. El diseñador también puede examinar el diseño para eliminar cualquier característica redundante. En este punto, la atención se centra en describir los datos y sus relaciones, más que en especificar los detalles del almacenamiento físico.

En términos del modelo relacional, el proceso de diseño conceptual implica decisiones sobre *qué* atributos se desea capturar en la base de datos y *cómo agruparlos* para formar las diferentes tablas. La parte "qué" es, esencialmente, una decisión conceptual, y no se seguirá estudiando en este texto. La parte del "cómo" es, esencialmente, un problema informático. Hay dos vías principales para afrontar el problema. La primera supone usar el modelo entidad-relación (Apartado 1.6.3); la otra es emplear un conjunto de algoritmos (denominados colectivamente como *normalización*) que toma como entrada el conjunto de todos los atributos y genera un conjunto de tablas (Apartado 1.6.4).

Un esquema conceptual completamente desarrollado también indica los requisitos funcionales de la empresa. En la **especificación de requisitos funcionales** los usuarios describen el tipo de operaciones (o transacciones) que se llevarán a cabo con los datos. Un ejemplo de estas operaciones es modificar o actualizar los datos, buscar y recuperar datos concretos y eliminar datos. En esta etapa del diseño conceptual el diseñador puede revisar el esquema para asegurarse de que satisface los requisitos funcionales.

El proceso de pasar de un modelo de datos abstracto a la implementación de la base de datos continúa con dos fases de diseño finales. En la **fase de diseño lógico** el diseñador relaciona el esquema conceptual de alto nivel con el modelo de implementación de datos del sistema de bases de datos que se va a usar. El diseñador usa el esquema de bases de datos específico para el sistema resultante en la **fase de diseño físico** posterior, en la que se especifican las características físicas de la base de datos. Entre esas características están la forma de organización de los archivos y las estructuras de almacenamiento interno; se estudian en el Capítulo 11.

1.6.2 Diseño de la base de datos para una entidad bancaria

Para ilustrar el proceso de diseño, examinemos el modo en que puede diseñarse una base de datos para una entidad bancaria. La especificación inicial de los requisitos de los usuarios puede basarse en entrevistas con los usuarios de la base de datos y en el análisis de la entidad realizado por el propio diseñador. La descripción que surge de esta fase de diseño sirve como base para especificar la estructura conceptual de la base de datos. Éstas son las principales características de la entidad bancaria:

- El banco está organizado en sucursales. Cada sucursal se ubica en una ciudad determinada y queda identificada por un nombre único. El banco supervisa los activos de cada sucursal.
- Los clientes quedan identificados por el valor de su *id_cliente.* El banco almacena el nombre de cada cliente y la calle y la ciudad en la que vive. Los clientes pueden abrir cuentas y solicitar préstamos. Cada cliente puede asociarse con un empleado del banco en concreto, que puede actuar como prestamista o como asesor personal para ese cliente.
- El banco ofrece dos tipos de cuenta: de ahorro y corriente. Las cuentas pueden tener como titular a más de un cliente, y cada cliente puede abrir más de una cuenta. A cada cuenta se le asigna un número de cuenta único. El banco guarda un registro del saldo de cada cuenta y de la fecha

más reciente en que cada cliente titular de esa cuenta tuvo acceso a ella. Además, cada cuenta de ahorro tiene una tasa de interés y se registran los descubiertos de las cuentas corrientes.

- El banco ofrece préstamos a sus clientes. Cada préstamo se origina en una sucursal concreta y puede tener como titulares a uno o más clientes. Cada préstamo queda identificado por un número de préstamo único. De cada préstamo el banco realiza un seguimiento del importe del préstamo y de sus pagos. Aunque el número de pago del préstamo no identifica de manera única un pago concreto entre todos los del banco, el número de pago identifica cada pago concreto de un préstamo dado. Se registran la fecha y el importe de cada pago.
- Los empleados del banco quedan identificados por el valor de su *id_empleado*. La administración del banco almacena el nombre y número de teléfono de cada empleado, el nombre de las personas dependientes de cada empleado y el número de *id_empleado* del jefe de cada empleado. El banco también realiza un seguimiento de la fecha de contratación y, por tanto, de la antigüedad de cada empleado.

En las entidades bancarias reales, el banco realizaría un seguimiento de los depósitos y de los reintegros de las cuentas de ahorro y corrientes, igual que realiza un seguimiento de los pagos de las cuentas de crédito. Dado que los requisitos de modelado para ese seguimiento son parecidas y nos gustaría que la aplicación de ejemplo no tuviera un tamaño excesivo, en nuestro modelo no se realiza el seguimiento de esos depósitos y reintegros.

1.6.3 El modelo entidad-relación

El modelo de datos entidad-relación (E-R) está basado en una percepción del mundo real que consiste en un conjunto de objetos básicos, denominados *entidades*, y de las *relaciones* entre esos objetos. Una entidad es una "cosa" u "objeto" del mundo real que es distinguible de otros objetos. Por ejemplo, cada persona es una entidad, y las cuentas bancarias pueden considerarse entidades.

Las entidades se describen en las bases de datos mediante un conjunto de **atributos**. Por ejemplo, los atributos *número_cuenta* y *saldo* pueden describir una cuenta concreta de un banco y constituyen atributos del conjunto de entidades *cuenta*. Análogamente, los atributos *nombre_cliente*, *calle_cliente* y *ciudad_cliente* pueden describir una entidad *cliente*.

Se usa un atributo extra, *id_cliente*, para identificar unívocamente a los clientes (dado que es posible que haya dos clientes con el mismo nombre, calle y ciudad). Se debe asignar un identificador de cliente único a cada cliente. En Estados Unidos, muchas empresas usan el número de la seguridad social de cada persona (un número único que el Gobierno de Estados Unidos asigna a cada persona) como identificador de cliente.

Una **relación** es una asociación entre varias entidades. Por ejemplo, la relación *impositor* asocia un cliente con cada cuenta que tiene. El conjunto de todas las entidades del mismo tipo, y el conjunto de todas las relaciones del mismo tipo se denominan, respectivamente, **conjunto de entidades** y **conjunto de relaciones**.

La estructura lógica general (esquema) de la base de datos se puede expresar gráficamente mediante un *diagrama E-R*, que está constituido por los siguientes componentes:

- **Rectángulos**, que representan conjuntos de entidades.
- **Elipses**, que representan atributos.
- **Rombos**, que representan conjuntos de relaciones entre miembros de varios conjuntos de entidades.
- **Líneas**, que unen los atributos con los conjuntos de entidades entre sí, y también los conjuntos de entidades con las relaciones.

Cada componente se etiqueta con la entidad o relación que representa.

Como ilustración, considérese parte de un sistema bancario de bases de datos consistente en los clientes y las cuentas que tienen esos clientes. La Figura 1.3 muestra el diagrama E-R correspondiente. El

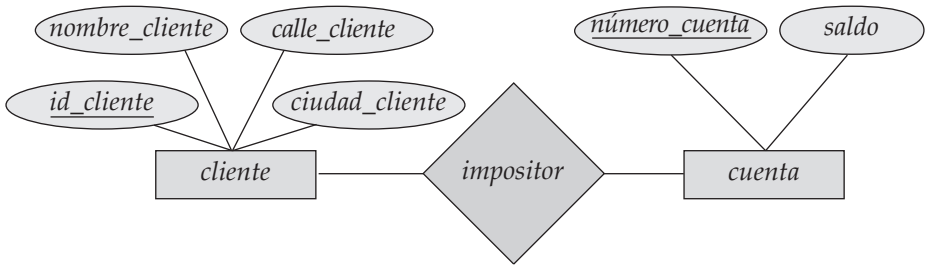


Figura 1.3 Un ejemplo de diagrama E-R.

id_cliente	*número_cuenta*	*saldo*
19.283.746	C-101	500
19.283.746	C-201	900
01.928.374	C-215	700
67.789.901	C-102	400
18.273.609	C-305	350
32.112.312	C-217	750
33.666.999	C-222	700
01.928.374	C-201	900

Figura 1.4 La tabla *impositor'*.

diagrama E-R indica que hay dos conjuntos de entidades, *cliente* y *cuenta*, con los atributos descritos anteriormente. El diagrama también muestra la relación *impositor* entre cliente y cuenta.

Además de entidades y relaciones, el modelo E-R representa ciertas restricciones que los contenidos de la base de datos deben cumplir. Una restricción importante es la **correspondencia de cardinalidades**, que expresa el número de entidades con las que otra entidad se puede asociar a través de un conjunto de relaciones. Por ejemplo, si cada cuenta sólo debe pertenecer a un cliente, el modelo puede expresar esa restricción.

El modelo entidad-relación se usa ampliamente en el diseño de bases de datos, y en el Capítulo 6 se explora en detalle.

1.6.4 Normalización

Otro método de diseño de bases de datos es usar un proceso que suele denominarse normalización. El objetivo es generar un conjunto de esquemas de relaciones que permita almacenar información sin redundancias innecesarias, pero que también permita recuperar la información con facilidad. El enfoque es diseñar esquemas que se hallen en la *forma normal* adecuada. Para determinar si un esquema de relación se halla en una de las formas normales deseadas, hace falta información adicional sobre la empresa real que se está modelando con la base de datos. El enfoque más frecuente es usar **dependencias funcionales**, que se tratan en el Apartado 7.4.

Para comprender la necesidad de la normalización, obsérvese lo que puede fallar en un mal diseño de base de datos. Algunas de las propiedades no deseables de un mal son:

- Repetición de la información.
- Imposibilidad de representar determinada información.

Se examinarán estos problemas con la ayuda de un diseño de bases de datos modificado para el ejemplo bancario.

Supóngase que, en lugar de tener las dos tablas separadas *cuenta* e *impositor*, se tiene una sola tabla, *impositor'*, que combina la información de las dos tablas (como puede verse en la Figura 1.4). Obsérvese que hay dos filas de *impositor'* que contienen información sobre la cuenta C-201. La repetición de información en este diseño alternativo no es deseable. La repetición de información malgasta espacio.

id_cliente	*nombre_cliente*	*calle_cliente*	*ciudad_cliente*	*número_cuenta*
19.283.746	González	Arenal, 12	La Granja	C-101
19.283.746	González	Arenal, 12	La Granja	C-201
67.789.901	López	Mayor, 3	Peguerinos	C-102
18.273.609	Abril	Preciados, 123	Valsaín	C-305
32.112.312	Santos	Mayor, 100	Peguerinos	C-217
33.666.999	Rupérez	Ramblas, 175	León	C-222
01.928.374	Gómez	Carretas, 72	Cerceda	C-201

Figura 1.5 La tabla *cliente'*.

Además, complica las actualizaciones de la base de datos. Supóngase que se desea modificar el saldo de la cuenta C-201 de 900 a 950 €. Esta modificación debe reflejarse en las dos filas; compárese con el diseño original, en el que esto daría lugar a la actualización de una sola fila. Por tanto, las actualizaciones resultan más costosas bajo el diseño alternativo que bajo el diseño original. Cuando se lleva a cabo la actualización de la base de datos alternativa, hay que asegurarse de que *todas* las tuplas que afectan a la cuenta C-201 se actualicen, o la base de datos mostrará dos valores de saldo diferentes para la cuenta C-201.

Examínese ahora el problema de la "imposibilidad de representar determinada información". Supóngase que, en vez de tener las dos tablas separadas *cliente* e *impositor*, se tuviera una sola tabla, *cliente'*, que combinara la información de esas dos tablas (como puede verse en la Figura 1.5). No se puede representar directamente la información relativa a los clientes (*id_cliente*, *nombre_cliente*, *calle_cliente*, *ciudad_cliente*) a menos que el cliente tenga, como mínimo, una cuenta en el banco. Esto se debe a que las filas de *cliente'* necesitan valores de *número_cuenta*.

Una solución de este problema es introducir valores **nulos**. Los valores *nulos* indican que el valor no existe (o es desconocido). Los valores desconocidos pueden ser valores *ausentes* (el valor existe, pero no se tiene la información) o valores *desconocidos* (no se sabe si el valor existe realmente o no). Como se verá más adelante, los valores nulos resultan difíciles de tratar, y es preferible no recurrir a ellos. Si no se desea tratar con valores nulos, se puede crear un elemento concreto de información del cliente sólo si el cliente tiene cuenta en el banco (obsérvese que los clientes pueden tener un préstamo pero no tener ninguna cuenta). Además, habría que eliminar esa información cuando el cliente cerrara la cuenta. Claramente, esta situación no es deseable ya que, bajo el diseño original de la base de datos, la información de los clientes estaría disponible independientemente de si el cliente tiene cuenta en el banco o no, y sin necesidad de recurrir a los valores nulos.

1.7 Bases de datos basadas en objetos y semiestructuradas

Varias áreas de aplicaciones de los sistemas de bases de datos están limitadas por las restricciones del modelo de datos relacional. En consecuencia, los investigadores han desarrollado varios modelos de datos para tratar con estos dominios de aplicación. Los modelos de datos que se tratarán en este texto son el orientado a objetos y el relacional orientado a objetos, representativos de los modelos de datos basados en objetos, y XML, representativo de los modelos de datos semiestructurados.

1.7.1 Modelos de datos basados en objetos

El **modelo de datos orientado a objetos** se basa en el paradigma de los lenguajes de programación orientados a objetos, que actualmente se usa en gran medida. La herencia, la identidad de los objetos y la encapsulación (ocultación de la información), con métodos para ofrecer una interfaz para los objetos, están entre los conceptos principales de la programación orientada a objetos que han encontrado aplicación en el modelado de datos. El modelo de datos orientado a objetos también soporta un sistema elaborado de tipos, incluidos los tipos estructurados y las colecciones. El modelo orientado a objetos puede considerarse una extensión del modelo E-R con los conceptos de encapsulación, métodos (funciones) e identidad de los objetos.

El **modelo de datos relacional orientado a objetos** extiende el modelo relacional tradicional con gran variedad de características como los tipos estructurados y las colecciones, así como la orientación a objetos.

En el Capítulo 9 se examinan las bases de datos relacionales orientadas a objetos (es decir, las bases de datos construidas según el modelo relacional orientado a objetos), así como las bases de datos orientadas a objetos (es decir, las bases de datos construidas según el modelo de datos orientado a objetos).

1.7.2 Modelos de datos semiestructurados

Los modelos de datos semiestructurados permiten la especificación de los datos en los que cada elemento de datos del mismo tipo puede tener conjuntos de atributos diferentes. Esto los diferencia de los modelos de datos mencionados anteriormente, en los que todos los elementos de datos de un tipo dado deben tener el mismo conjunto de atributos.

El lenguaje XML se diseñó inicialmente como un modo de añadir información de marcas a los documentos de texto, pero se ha vuelto importante debido a sus aplicaciones en el intercambio de datos. XML ofrece un modo de representar los datos que tienen una estructura anidada y, además, permite una gran flexibilidad en la estructuración de los datos, lo cual es importante para ciertas clases de datos no tradicionales. En el Capítulo 10 se describe el lenguaje XML, diferentes maneras de expresar las consultas sobre datos representados en XML y la transformación de los datos XML de una forma a otra.

1.8 Almacenamiento de datos y consultas

Los sistemas de bases de datos están divididos en módulos que tratan con cada una de las responsabilidades del sistema general. Los componentes funcionales de los sistemas de bases de datos pueden dividirse grosso modo en los componentes gestor de almacenamiento y procesador de consultas.

El gestor de almacenamiento es importante porque las bases de datos suelen necesitar una gran cantidad de espacio de almacenamiento. Las bases de datos corporativas tienen un tamaño que va de los centenares de gigabytes hasta, para las bases de datos de mayor tamaño, los terabytes de datos. Un gigabyte son aproximadamente 1.000 megabytes (1.000 millones de bytes), y un terabyte es aproximadamente un millón de megabytes (1 billón de bytes). Debido a que la memoria principal de las computadoras no puede almacenar toda esta información, la información se almacena en discos. Los datos se intercambian entre los discos de almacenamiento y la memoria principal cuando sea necesario. Como el intercambio de datos con el disco es lento comparado con la velocidad de la unidad central de procesamiento, es fundamental que el sistema de base de datos estructure los datos para minimizar la necesidad de intercambio de datos entre los discos y la memoria principal.

El procesador de consultas es importante porque ayuda al sistema de bases de datos a simplificar y facilitar el acceso a los datos. Las vistas de alto nivel ayudan a conseguir este objetivo; con ellas, los usuarios del sistema no se ven molestados innecesariamente con los detalles físicos de la implementación del sistema. Sin embargo, el rápido procesamiento de las actualizaciones y de las consultas es importante. Es función del sistema de bases de datos traducir las actualizaciones y las consultas escritas en lenguajes no procedimentales, en el nivel lógico, en una secuencia eficiente de operaciones en el nivel físico.

1.8.1 Gestor de almacenamiento

Un *gestor de almacenamiento* es un módulo de programa que proporciona la interfaz entre los datos de bajo nivel almacenados en la base de datos y los programas de aplicación y las consultas remitidas al sistema. El gestor de almacenamiento es responsable de la interacción con el gestor de archivos. Los datos en bruto se almacenan en el disco mediante el sistema de archivos que suele proporcionar un sistema operativo convencional. El gestor de almacenamiento traduce las diferentes instrucciones LMD a comandos de bajo nivel del sistema de archivos. Así, el gestor de almacenamiento es responsable del almacenamiento, la recuperación y la actualización de los datos de la base de datos.

Entre los componentes del gestor de almacenamiento se encuentran:

- **Gestor de autorizaciones e integridad**, que comprueba que se satisfagan las restricciones de integridad y la autorización de los usuarios para tener acceso a los datos.

- **Gestor de transacciones**, que garantiza que la base de datos quede en un estado consistente (correcto) a pesar de los fallos del sistema, y que la ejecución concurrente de transacciones transcurra si conflictos.
- **Gestor de archivos**, que gestiona la asignación de espacio de almacenamiento de disco y las estructuras de datos usadas para representar la información almacenada en el disco.
- **Gestor de la memoria intermedia**, que es responsable de traer los datos desde el disco de almacenamiento a la memoria principal y decidir los datos a guardar en la memoria caché. El gestor de la memoria intermedia es una parte fundamental de los sistemas de bases de datos, ya que permite que la base de datos maneje tamaños de datos que son mucho mayores que el tamaño de la memoria principal.

El gestor de almacenamiento implementa varias estructuras de datos como parte de la implementación física del sistema:

- **Archivos de datos**, que almacenan la base de datos en sí misma.
- **Diccionario de datos**, que almacena metadatos acerca de la estructura de la base de datos; en particular, su esquema.
- **Índices**, que pueden proporcionar un acceso rápido a los elementos de datos. Como el índice de este libro de texto, los índices de las bases de datos facilitan punteros a los elementos de datos que tienen un valor concreto. Por ejemplo, se puede usar un índice para buscar todos los registros *cuenta* con un *número_cuenta* determinado. La asociación es una alternativa a la indexación que es más rápida en algunos casos, pero no en todos.

Se estudiarán los medios de almacenamiento, las estructuras de archivos y la gestión de la memoria intermedia en el Capítulo 11. Los métodos de acceso eficiente a los datos mediante indexación o asociación se explican en el Capítulo 12.

1.8.2 El procesador de consultas

Entre los componentes del procesador de consultas se encuentran:

- **Intérprete del LDD**, que interpreta las instrucciones del LDD y registra las definiciones en el diccionario de datos.
- **Compilador del LMD**, que traduce las instrucciones del LMD en un lenguaje de consultas a un plan de evaluación que consiste en instrucciones de bajo nivel que entienda el motor de evaluación de consultas.

  Las consultas se suelen poder traducir en varios planes de ejecución alternativos, todos los cuales proporcionan el mismo resultado. El compilador del LMD también realiza **optimización de consultas**, es decir, elige el plan de evaluación de menor coste de entre todas las opciones posibles.
- **Motor de evaluación de consultas**, que ejecuta las instrucciones de bajo nivel generadas por el compilador del LMD.

La evaluación de las consultas se trata en el Capítulo 13, mientras que los métodos por los que el optimizador de consultas elige entre las estrategias de evaluación posibles se tratan en el Capítulo 14.

1.9 Gestión de transacciones

A menudo, varias operaciones sobre la base de datos forman una única unidad lógica de trabajo. Un ejemplo son las transferencia de fondos, como se vio en el Apartado 1.2, en las que se realiza un cargo en una cuenta (llámese *A*) y un abono en otra cuenta (llámese *B*). Evidentemente, resulta fundamental que, o bien tengan lugar tanto el cargo como el abono, o bien que no se produzca ninguno. Es decir, la transferencia de fondos debe tener lugar por completo o no producirse en absoluto. Este requisito

de todo o nada se denomina **atomicidad**. Además, resulta esencial que la ejecución de la transferencia de fondos preserve la consistencia de la base de datos. Es decir, el valor de la suma $A + B$ se debe preservar. Este requisito de corrección se denomina **consistencia**. Finalmente, tras la ejecución correcta de la transferencia de fondos, los nuevos valores de las cuentas A y B deben persistir, a pesar de la posibilidad de fallo del sistema. Este requisito de persistencia se denomina **durabilidad**.

Una **transacción** es un conjunto de operaciones que lleva a cabo una única función lógica en una aplicación de bases de datos. Cada transacción es una unidad de atomicidad y consistencia. Por tanto, se exige que las transacciones no violen ninguna restricción de consistencia de la base de datos. Es decir, si la base de datos era consistente cuando la transacción comenzó, debe ser consistente cuando la transacción termine con éxito. Sin embargo, durante la ejecución de una transacción, puede ser necesario permitir inconsistencias temporalmente, ya que el cargo a A o el abono a B se debe realizar en primer lugar. Esta inconsistencia temporal, aunque necesaria, puede conducir a dificultades si ocurre un fallo.

Es responsabilidad del programador definir adecuadamente las diferentes transacciones, de tal manera que cada una preserve la consistencia de la base de datos. Por ejemplo, la transacción para transferir fondos de la cuenta A a la cuenta B puede definirse como si estuviera compuesta de dos programas diferentes: uno que realiza el cargo en la cuenta A y otro que realiza el abono en la cuenta B. La ejecución de estos dos programas uno después del otro preservará realmente la consistencia. Sin embargo, cada programa en sí mismo no transforma la base de datos de un estado consistente a otro nuevo. Por tanto, estos programas no son transacciones.

Garantizar las propiedades de atomicidad y de durabilidad es responsabilidad del propio sistema de bases de datos —concretamente del **componente de gestión de transacciones**. A falta de fallos, todas las transacciones se completan con éxito y la atomicidad se consigue fácilmente. Sin embargo, debido a diversos tipos de fallos, puede que las transacciones no siempre completen su ejecución con éxito. Si se va a asegurar la propiedad de atomicidad, las transacciones fallidas no deben tener ningún efecto sobre el estado de la base de datos. Por tanto, la base de datos debe restaurarse al estado en que estaba antes de que la transacción en cuestión comience a ejecutarse. El sistema de bases de datos, por tanto, debe realizar la **recuperación de fallos**, es decir, detectar los fallos del sistema y restaurar la base de datos al estado que tenía antes de que ocurriera el fallo.

Finalmente, cuando varias transacciones actualizan la base de datos de manera concurrente, puede que no se preserve la consistencia de los datos, aunque cada una de las transacciones sea correcta. Es responsabilidad del **gestor de control de concurrencia** controlar la interacción entre las transacciones concurrentes para garantizar la consistencia de la base de datos.

Los conceptos básicos del procesamiento de transacciones se tratan en el Capítulo 15. La gestión de las transacciones concurrentes se trata en el Capítulo 16. En el Capítulo 17 se trata con detalle la recuperación de fallos.

Puede que los sistemas de bases de datos diseñados para su empleo en computadoras personales pequeños no tengan todas estas características. Por ejemplo, muchos sistemas pequeños sólo permiten que un usuario tenga acceso a la base de datos en cada momento. Otros no ofrecen copias de seguridad ni recuperación, y dejan esas tareas a los usuarios. Estas restricciones permiten un gestor de datos de menor tamaño, con menos requisitos de recursos físicos —especialmente de memoria principal. Aunque tales enfoques de bajo coste y bajas prestaciones son adecuados para bases de datos personales pequeñas, resultan inadecuados para empresas medianas y grandes.

El concepto de transacción se ha aplicado ampliamente en los sistemas y en las aplicaciones de bases de datos. Aunque el empleo inicial de las transacciones se produjo en las aplicaciones financieras, el concepto se usa ahora en aplicaciones de tiempo real de telecomunicaciones, así como en la gestión de las actividades de larga duración como el diseño de productos o los flujos de trabajo administrativos. Estas aplicaciones más amplias del concepto de transacción se estudian en el Capítulo 25.

1.10 Minería y análisis de datos

El término **minería de datos** se refiere en líneas generales al proceso de análisis semiautomático de grandes bases de datos para descubrir patrones útiles. Al igual que el descubrimiento de conocimiento en inteligencia artificial (también denominado **aprendizaje de la máquina**) o el análisis estadístico, la minería de datos intenta descubrir reglas y patrones en los datos. Sin embargo, la minería de datos

se diferencia del aprendizaje de la máquina y de la estadística en que maneja grandes volúmenes de datos, almacenados principalmente en disco. Es decir, la minería de datos trata del "descubrimiento de conocimiento en las bases de datos".

Algunos tipos de conocimiento descubiertos en las bases de datos pueden representarse mediante un conjunto de **reglas**. Lo que sigue es un ejemplo de regla, definida informalmente: "las mujeres jóvenes con ingresos anuales superiores a 50.000 € son las personas con más probabilidades de comprar coches deportivos pequeños". Por supuesto, esas reglas no son universalmente ciertas, sino que tienen grados de "apoyo" y de "confianza". Otros tipos de conocimiento se representan mediante ecuaciones que relacionan diferentes variables, o mediante otros mecanismos para la predicción de los resultados cuando se conocen los valores de algunas variables.

Hay gran variedad de tipos posibles de patrones que pueden resultar útiles y se emplean diferentes técnicas para descubrir tipos de patrones diferentes. En el Capítulo 18 se estudian unos cuantos ejemplos de patrones y se ve la manera en que pueden obtenerse de las bases de datos de forma automática.

Generalmente hay un componente manual en la minería de datos, que consiste en el preprocesamiento de los datos de una manera aceptable para los algoritmos, y el postprocesamiento de los patrones descubiertos para descubrir otros nuevos que puedan resultar útiles. Puede haber también más de un tipo de patrón que pueda descubrirse en una base de datos dada, y puede ser necesaria la interacción manual para escoger los tipos de patrones útiles. Por este motivo, la minería de datos es, en realidad, un proceso semiautomático en la vida real. No obstante, en nuestra descripción se concentra la atención en el aspecto automático de la minería.

Las empresas han comenzado a explotar la creciente cantidad de datos en línea para tomar mejores decisiones sobre sus actividades, como los artículos de los que hay que tener existencias y la mejor manera de llegar a los clientes para incrementar las ventas. Muchas de sus consultas son bastante complicadas, sin embargo, y ciertos tipos de información no pueden extraerse ni siquiera usando SQL.

Se dispone de varias técnicas y herramientas para ayudar a la toma de decisiones. Varias herramientas para el análisis de datos permiten a los analistas ver los datos de diferentes maneras. Otras herramientas de análisis realizan cálculos previos de resúmenes de grandes cantidades de datos, con objeto de dar respuestas rápidas a las preguntas. El estándar SQL:1999 contiene actualmente constructores adicionales para soportar el análisis de datos.

Los datos textuales también han crecido de manera explosiva. Estos datos carecen de estructura, a diferencia de los datos rígidamente estructurados de las bases de datos relacionales. La consulta de datos textuales no estructurados se denomina *recuperación de la información*. Los sistemas de recuperación de la información tienen mucho en común con los sistemas de bases de datos —en especial, el almacenamiento y recuperación de datos en medios de almacenamiento secundarios. Sin embargo, el énfasis en el campo de los sistemas de información es diferente del de los sistemas de bases de datos, y se concentra en aspectos como las consultas basadas en palabras clave; la relevancia de los documentos para la consulta, y el análisis, clasificación e indexación de los documentos. En los Capítulos 18 y 19 se trata la ayuda a la toma de decisiones, incluyendo el procesamiento analítico en línea, la minería de datos y la recuperación de la información.

1.11 Arquitectura de las bases de datos

Ahora es posible ofrecer una visión única (Figura 1.6) de los diversos componentes de los sistemas de bases de datos y de las conexiones existentes entre ellos.

La arquitectura de los sistemas de bases de datos se ve muy influida por el sistema informático subyacente sobre el que se ejecuta el sistema de bases de datos. Los sistemas de bases de datos pueden estar centralizados o ser del tipo cliente-servidor, en los que una máquina servidora ejecuta el trabajo en nombre de multitud de máquinas clientes. Los sistemas de bases de datos pueden diseñarse también para aprovechar las arquitecturas de computadoras paralelas. Las bases de datos distribuidas se extienden por varias máquinas geográficamente separadas.

En el Capítulo 20 se trata la arquitectura general de los sistemas informáticos modernos. El Capítulo 21 describe el modo en que diversas acciones de las bases de datos, en especial el procesamiento de las consultas, pueden implementarse para aprovechar el procesamiento paralelo. El Capítulo 22 presenta varios problemas que surgen en las bases de datos distribuidas y describe el modo de afrontarlos. Entre

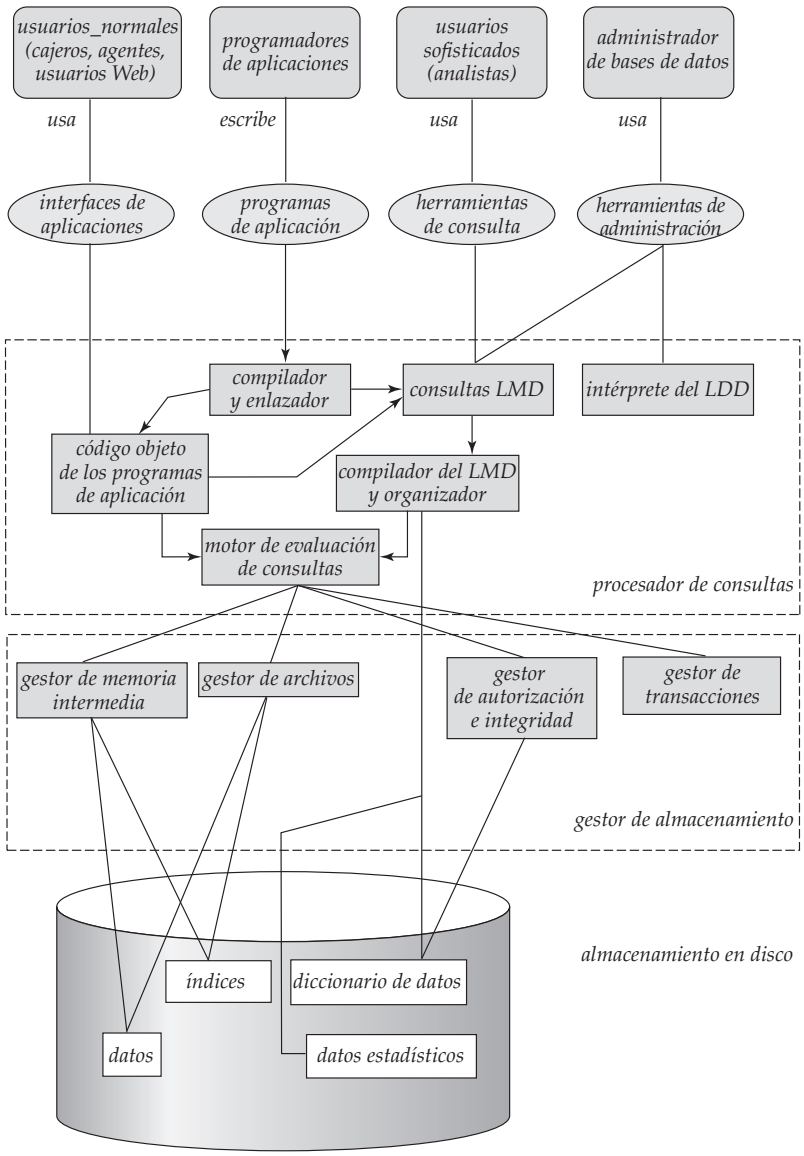


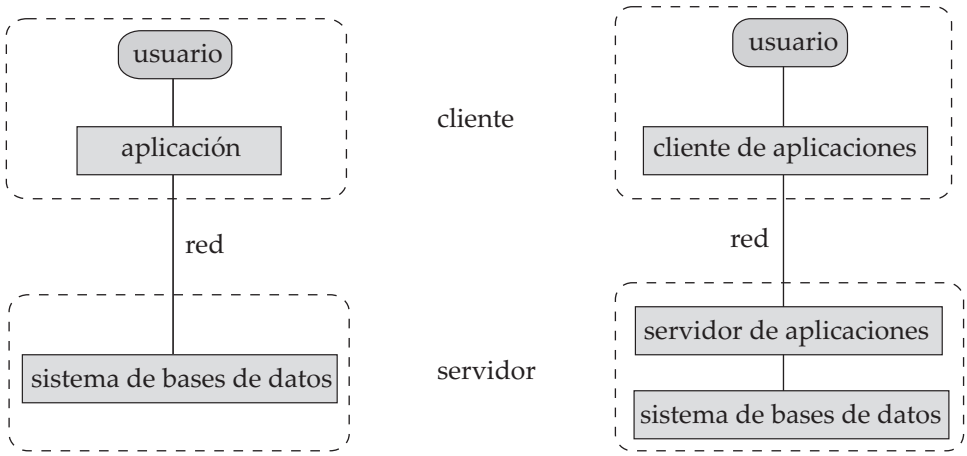
Figura 1.6 Arquitectura del sistema.

los problemas se encuentran el modo de almacenar los datos, la manera de asegurar la atomicidad de las transacciones que se ejecutan en varios sitios, cómo llevar a cabo controles de concurrencia y el modo de ofrecer alta disponibilidad en presencia de fallos. El procesamiento distribuido de las consultas y los sistemas de directorio también se describen en ese capítulo.

Hoy en día la mayor parte de los usuarios de los sistemas de bases de datos no está presente en el lugar físico en que se encuentra el sistema de bases de datos, sino que se conectan a él a través de una red. Por tanto, se puede diferenciar entre los sistemas **clientes**, en los que trabajan los usuarios remotos de la base de datos, y los sistemas **servidores**, en los que se ejecutan los sistemas de bases de datos.

Las aplicaciones de bases de datos suelen dividirse en dos o tres partes, como puede verse en la Figura 1.7. En una **arquitectura de dos capas**, la aplicación se divide en un componente que reside en la máquina cliente, que llama a la funcionalidad del sistema de bases de datos en la máquina servidora mediante instrucciones del lenguaje de consultas. Los estándares de interfaces de programas de aplicación como ODBC y JDBC se usan para la interacción entre el cliente y el servidor.

En cambio, en una **arquitectura de tres capas**, la máquina cliente actúa simplemente como una parte visible al usuario y no contiene ninguna llamada directa a la base de datos. En vez de eso, el extremo



(a) Arquitectura de dos capas (b) Arquitectura de tres capas

Figura 1.7 Arquitecturas de dos y tres capas.

cliente se comunica con un **servidor de aplicaciones**, generalmente mediante una interfaz de formularios. El servidor de aplicaciones, a su vez, se comunica con el sistema de bases de datos para tener acceso a los datos. La **lógica de negocio** de la aplicación, que establece las acciones que se deben realizar según las condiciones reinantes, se incorpora en el servidor de aplicaciones, en lugar de estar distribuida entre múltiples clientes. Las aplicaciones de tres capas resultan más adecuadas para aplicaciones de gran tamaño y para las aplicaciones que se ejecutan en World Wide Web.

1.12 Usuarios y administradores de bases de datos

Uno de los objetivos principales de los sistemas de bases de datos es recuperar información de la base de datos y almacenar en ella información nueva. Las personas que trabajan con una base de datos se pueden clasificar como usuarios o administradores de bases de datos.

1.12.1 Usuarios de bases de datos e interfaces de usuario

Hay cuatro tipos diferentes de usuarios de los sistemas de bases de datos, diferenciados por la forma en que esperan interactuar con el sistema. Se han diseñado diferentes tipos de interfaces de usuario para los diferentes tipos de usuarios.

- Los **usuarios normales** son usuarios no sofisticados que interactúan con el sistema invocando alguno de los programas de aplicación que se han escrito previamente. Por ejemplo, un cajero bancario que necesita transferir 50 € de la cuenta *A* a la cuenta *B* invoca un programa llamado *transferencia*. Ese programa le pide al cajero el importe de dinero que se va a transferir, la cuenta desde la que se va a transferir el dinero y la cuenta a la que se va a transferir el dinero.

  Como ejemplo adicional, considérese un usuario que desea averiguar el saldo de su cuenta en World Wide Web. Ese usuario puede acceder a un formulario en el que introduce su número de cuenta. Un programa de aplicación en el servidor Web recupera entonces el saldo de la cuenta, usando el número de cuenta proporcionado, y devuelve la información al usuario.

  La interfaz de usuario habitual para los usuarios normales es una interfaz de formularios, donde el usuario puede rellenar los campos correspondientes del formulario. Los usuarios normales también pueden limitarse a leer *informes* generados por la base de datos.

- Los **programadores de aplicaciones** son profesionales informáticos que escriben programas de aplicación. Los programadores de aplicaciones pueden elegir entre muchas herramientas para desarrollar las interfaces de usuario. Las herramientas de **desarrollo rápido de aplicaciones**

(DRA) son herramientas que permiten al programador de aplicaciones crear formularios e informes con un mínimo esfuerzo de programación.

- Los **usuarios sofisticados** interactúan con el sistema sin escribir programas. En su lugar, formulan sus consultas en un lenguaje de consultas de bases de datos. Remiten cada una de las consultas al **procesador de consultas**, cuya función es dividir las instrucciones LMD en instrucciones que el gestor de almacenamiento entienda. Los analistas que remiten las consultas para explorar los datos de la base de datos entran en esta categoría.
- Los **usuarios especializados** son usuarios sofisticados que escriben aplicaciones de bases de datos especializadas que no encajan en el marco tradicional del procesamiento de datos. Entre estas aplicaciones están los sistemas de diseño asistido por computadora, los sistemas de bases de conocimientos y los sistemas expertos, los sistemas que almacenan datos con tipos de datos complejos (por ejemplo, los datos gráficos y los datos de sonido) y los sistemas de modelado del entorno. En el Capítulo 9 se estudian varias de estas aplicaciones.

1.12.2 Administrador de bases de datos

Una de las principales razones de usar SGBDs es tener un control centralizado tanto de los datos como de los programas que tienen acceso a esos datos. La persona que tiene ese control central sobre el sistema se denomina **administrador de bases de datos** (**ABD**). Las funciones del ABD incluyen:

- La **definición del esquema**. El ABD crea el esquema original de la base de datos mediante la ejecución de un conjunto de instrucciones de definición de datos en el LDD.
- La **definición de la estructura y del método de acceso**.
- La **modificación del esquema y de la organización física**. El ABD realiza modificaciones en el esquema y en la organización física para reflejar las necesidades cambiantes de la organización, o para alterar la organización física a fin de mejorar el rendimiento.
- La **concesión de autorización para el acceso a los datos**. Mediante la concesión de diferentes tipos de autorización, el administrador de bases de datos puede regular las partes de la base de datos a las que puede tener acceso cada usuario. La información de autorización se guarda en una estructura especial del sistema que el SGBD consulta siempre que alguien intenta tener acceso a los datos del sistema.
- El **mantenimiento rutinario**. Algunos ejemplos de las actividades de mantenimiento rutinario del administrador de la base de datos son:
  - Copia de seguridad periódica de la base de datos, bien sobre cinta o sobre servidores remotos, para impedir la pérdida de datos en caso de desastres como las inundaciones.
  - Asegurarse de que se dispone de suficiente espacio libre en disco para las operaciones normales y aumentar el espacio en disco según sea necesario.
  - Supervisar los trabajos que se ejecuten en la base de datos y asegurarse de que el rendimiento no se degrade debido a que algún usuario haya remitido tareas muy costosas.

1.13 Historia de los sistemas de bases de datos

El procesamiento de datos impulsa el crecimiento de las computadoras, como lo ha hecho desde los primeros días de las computadoras comerciales. De hecho, la automatización de las tareas de procesamiento de datos precede a las computadoras. Las tarjetas perforadas, inventadas por Herman Hollerith, se emplearon a principios del siglo XX para registrar los datos del censo de Estados Unidos, y se usaron sistemas mecánicos para procesar las tarjetas y para tabular los resultados. Las tarjetas perforadas se usaron posteriormente con profusión como medio para introducir datos en las computadoras.

Las técnicas de almacenamiento y de procesamiento de datos han evolucionado a lo largo de los años:

- **Años cincuenta y primeros años sesenta**: se desarrollaron las cintas magnéticas para el almacenamiento de datos. Las tareas de procesamiento de datos como la elaboración de nóminas se

automatizaron, con los datos almacenados en cintas. El procesamiento de datos consistía en leer datos de una o varias cintas y escribir datos en una nueva cinta. Los datos también se podían introducir desde paquetes de tarjetas perforadas e imprimirse en impresoras. Por ejemplo, los aumentos de sueldo se procesaban introduciendo los aumentos en las tarjetas perforadas y leyendo el paquete de cintas perforadas de manera sincronizada con una cinta que contenía los detalles principales de los salarios. Los registros debían estar en el mismo orden. Los aumentos de sueldo se añadían a los sueldos leídos de la cinta maestra y se escribían en una nueva cinta; esa nueva cinta se convertía en la nueva cinta maestra.

Las cintas (y los paquetes de tarjetas perforadas) sólo se podían leer secuencialmente, y el tamaño de datos era mucho mayor que la memoria principal; por tanto, los programas de procesamiento de datos se veían obligados a procesar los datos en un orden determinado, leyendo y mezclando datos de las cintas y de los paquetes de tarjetas perforadas.

- **Finales de los años sesenta y años setenta**: el empleo generalizado de los discos duros a finales de los años sesenta modificó en gran medida la situación del procesamiento de datos, ya que permitieron el acceso directo a los datos. La ubicación de los datos en disco no era importante, ya que se podía tener acceso a cualquier posición del disco en sólo unas decenas de milisegundos. Los datos se liberaron así de la tiranía de la secuencialidad. Con los discos pudieron crearse las bases de datos de red y las bases de datos jerárquicas, que permitieron que las estructuras de datos como las listas y los árboles pudieran almacenarse en disco. Los programadores pudieron crear y manipular estas estructuras de datos.

  El artículo histórico de Codd [1970] definió el modelo relacional y las formas no procedimentales de consultar los datos en el modelo relacional, y así nacieron las bases de datos relacionales. La simplicidad del modelo relacional y la posibilidad de ocultar completamente los detalles de implementación a los programadores resultaron realmente atractivas. Codd obtuvo posteriormente el prestigioso premio Turing de la ACM (Association of Computing Machinery, asociación de maquinaria informática) por su trabajo.

- **Años ochenta**: aunque académicamente interesante, el modelo relacional no se usó inicialmente en la práctica debido a sus inconvenientes en cuanto a rendimiento; las bases de datos relacionales no podían igualar el rendimiento de las bases de datos de red y jerárquicas existentes. Esta situación cambió con System R, un proyecto innovador del centro de investigación IBM Research que desarrolló técnicas para la construcción de un sistema de bases de datos relacionales eficiente. En Astrahan et al. [1976] y Chamberlin et al. [1981] se pueden encontrar excelentes visiones generales de System R. El prototipo de System R completamente funcional condujo al primer producto de bases de datos relacionales de IBM: SQL/DS. Los primeros sistemas comerciales de bases de datos relacionales, como DB2 de IBM, Oracle, Ingres y Rdb de DEC, desempeñaron un importante papel en el desarrollo de técnicas para el procesamiento eficiente de las consultas declarativas. En los primeros años ochenta las bases de datos relacionales habían llegado a ser competitivas frente a los sistemas de bases de datos jerárquicas y de red incluso en cuanto a rendimiento. Las bases de datos relacionales eran tan sencillas de usar que finalmente reemplazaron a las bases de datos jerárquicas y de red; los programadores que usaban esas bases de datos se veían obligados a tratar muchos detalles de implementación de bajo nivel y tenían que codificar sus consultas de forma procedimental. Lo que era aún más importante, tenían que tener presente el rendimiento durante el diseño de los programas, lo que suponía un gran esfuerzo. En cambio, en las bases de datos relacionales, casi todas estas tareas de bajo nivel las realiza de manera automática el sistema de bases de datos, lo que libera al programador para que se centre en el nivel lógico. Desde su obtención de liderazgo en los años ochenta, el modelo relacional ha reinado sin discusión entre todos los modelos de datos.

  Los años ochenta también fueron testigos de una gran investigación en las bases de datos paralelas y distribuidas, así como del trabajo inicial en las bases de datos orientadas a objetos.

- **Primeros años noventa**: el lenguaje SQL se diseñó fundamentalmente para las aplicaciones de ayuda a la toma de decisiones, que son intensivas en consultas, mientras que el objetivo principal de las bases de datos en los años ochenta eran las aplicaciones de procesamiento de tran-

sacciones, que son intensivas en actualizaciones. La ayuda a la toma de decisiones y las consultas volvieron a emerger como una importante área de aplicación para las bases de datos. El uso de las herramientas para analizar grandes cantidades de datos experimentó un gran crecimiento.

En esta época muchas marcas de bases de datos introdujeron productos de bases de datos paralelas. Las diferentes marcas de bases de datos también comenzaron a añadir soporte relacional orientado a objetos a sus bases de datos.

- **Finales de los años noventa**: el principal acontecimiento fue el crecimiento explosivo de World Wide Web. Las bases de datos se implantaron mucho más ampliamente que nunca. Los sistemas de bases de datos tenían que soportar tasas de procesamiento de transacciones muy elevadas, así como una fiabilidad muy alta y tener disponibilidad 24 $\times$ 7 (disponibilidad 24 horas al día y 7 días a la semana, lo que significa que no hay momentos de inactividad debidos a actividades de mantenimiento planificadas). Los sistemas de bases de datos también tenían que soportar interfaces Web para los datos.
- **Principios del siglo XXI**: los principios del siglo XXI han sido testigos de la emergencia de XML y de su lenguaje de consultas asociado, XQuery, como nueva tecnología de las bases de datos. Todavía es pronto para decir el papel que XML desempeñará en las bases de datos futuras. En este periodo también se ha podido presenciar el crecimiento de las técnicas de "informática autónoma/administración automática" para la minimización del esfuerzo de administración.

1.14 Resumen

- Un **sistema gestor de bases de datos (SGBD)** consiste en un conjunto de datos interrelacionados y en un conjunto de programas para tener acceso a esos datos. Los datos describen una empresa concreta.
- El objetivo principal de un SGBD es proporcionar un entorno que sea tanto conveniente como eficiente para las personas que lo usan para la recuperación y almacenamiento de información.
- Los sistemas de bases de datos resultan ubicuos hoy en día, y la mayor parte de la gente interactúa, directa o indirectamente, con bases de datos muchas veces al día.
- Los sistemas de bases de datos se diseñan para almacenar grandes cantidades de información. La gestión de los datos implica tanto la definición de estructuras para el almacenamiento de la información como la provisión de mecanismos para la manipulación de la información. Además, los sistemas de bases de datos deben preocuparse de la seguridad de la información almacenada, en caso de caídas del sistema o de intentos de acceso sin autorización. Si los datos deben compartirse entre varios usuarios, el sistema debe evitar posibles resultados anómalos.
- Uno de los propósitos principales de los sistemas de bases de datos es ofrecer a los usuarios una visión abstracta de los datos. Es decir, el sistema oculta ciertos detalles de la manera en que los datos se almacenan y mantienen.
- Por debajo de la estructura de la base de datos se halla el **modelo de datos**: un conjunto de herramientas conceptuales para describir los datos, las relaciones entre los datos, la semántica de los datos y las restricciones de los datos.
- Un **lenguaje de manipulación de datos (LMD)** es un lenguaje que permite a los usuarios tener acceso a los datos o manipularlos. Los LMDs no procedimentales, que sólo necesitan que el usuario especifique los datos que necesita, sin especificar exactamente la manera de obtenerlos, se usan mucho hoy en día.
- Un **lenguaje de definición de datos (LDD)** es un lenguaje para la especificación del esquema de la base de datos y otras propiedades de los datos.
- El modelo de datos relacional es el más implantado para el almacenamiento de datos en las bases de datos. Otros modelos de datos son el modelo de datos orientado a objetos, el modelo relacional orientado a objetos y los modelos de datos semiestructurados.

- El diseño de bases de datos supone sobre todo el diseño del esquema de la base de datos. El modelo de datos entidad-relación (E-R) es un modelo de datos muy usado para el diseño de bases de datos. Proporciona una representación gráfica conveniente para ver los datos, las relaciones y las restricciones.
- Cada sistema de bases de datos tiene varios subsistemas:
  - El subsistema **gestor de almacenamiento** proporciona la interfaz entre los datos de bajo nivel almacenados en la base de datos y los programas de aplicación y las consultas remitidas al sistema.
  - El subsistema **procesador de consultas** compila y ejecuta instrucciones LDD y LMD.
- El **gestor de transacciones** garantiza que la base de datos permanezca en un estado consistente (correcto) a pesar de los fallos del sistema. El gestor de transacciones garantiza que la ejecución de las transacciones concurrentes se produzca sin conflictos.
- Las aplicaciones de bases de datos suelen dividirse en una fachada que se ejecuta en las máquinas clientes y una parte que se ejecuta en segundo plano. En las arquitecturas de dos capas la fachada se comunica directamente con una base de datos que se ejecuta en segundo plano. En las arquitecturas de tres capas la parte en segundo plano se divide a su vez en un servidor de aplicaciones y un servidor de bases de datos.
- Los usuarios de bases de datos se pueden dividir en varias clases, y cada clase de usuario suele usar un tipo diferente de interfaz para la base de datos.

Términos de repaso

- Sistema gestor de bases de datos (SGBD).
- Aplicaciones de sistemas de bases de datos.
- Sistemas de archivos.
- Inconsistencia de datos.
- Restricciones de consistencia.
- Vistas de datos.
- Abstracción de datos.
- Ejemplar de la base de datos.
- Esquema.
  - Esquema de la base de datos.
  - Esquema físico.
  - Esquema lógico.
- Independencia física de los datos.
- Modelos de datos.
  - Modelo entidad-relación.
  - Modelo de datos relacional.
  - Modelo de datos orientado a objetos.
  - Modelo de datos relacional orientado a objetos.
- Lenguajes de bases de datos.
  - Lenguaje de definición de datos.
  - Lenguaje de manipulación de datos.
  - Lenguaje de consultas.
- Diccionario de datos.
- Metadatos.
- Transacciones.
- Concurrencia.
- Programa de aplicación.
- Administrador de bases de datos (ABD).
- Máquinas cliente y servidor.

Ejercicios prácticos

1.1 En este capítulo se han descrito varias ventajas importantes de los sistemas gestores de bases de datos. ¿Cuáles son sus dos inconvenientes?

1.2 Indíquense siete lenguajes de programación que sean procedimentales y dos que no lo sean. ¿Qué grupo es más fácil de aprender a usar? Explíquese la respuesta.

1.3 Indíquense seis pasos importantes que se deben dar para configurar una base de datos para una empresa dada.

1.4 Considérese un *array* de enteros bidimensional de tamaño $n \times m$ que se va a usar en el lenguaje de programación preferido del lector. Usando el *array* como ejemplo, ilústrese la diferencia (a) entre los tres niveles de abstracción de datos y (b) entre el esquema y los ejemplares.

Ejercicios

1.5 Indíquense cuatro aplicaciones que se hayan usado que sea muy posible que utilicen un sistema de bases de datos para almacenar datos persistentes.

1.6 Indíquense cuatro diferencias significativas entre un sistema de procesamiento de archivos y un SGBD.

1.7 Explíquese la diferencia entre independencia de datos física y lógica.

1.8 Indíquense cinco responsabilidades del sistema gestor de bases de datos. Para cada responsabilidad, explíquense los problemas que surgirían si no se asumiera esa responsabilidad.

1.9 Indíquense al menos dos razones para que los sistemas de bases de datos soporten la manipulación de datos mediante un lenguaje de consultas declarativo como SQL, en vez de limitarse a ofrecer una biblioteca de funciones de C o de C++ para llevar a cabo la manipulación de los datos.

1.10 Explíquense los problemas que causa el diseño de la tabla de la Figura 1.5.

1.11 ¿Cuáles son las cinco funciones principales del administrador de bases de datos?

Notas bibliográficas

A continuación se ofrece una relación de libros de propósito general, colecciones de artículos de investigación y sitios Web sobre bases de datos. Los capítulos siguientes ofrecen referencias a material sobre cada tema descrito en ese capítulo.

Codd [1970] es el artículo histórico que introdujo el modelo relacional.

Entre los libros de texto que tratan los sistemas de bases de datos están Abiteboul et al. [1995], Date [2003], Elmasri y Navathe [2003], O'Neil y O'Neil [2000], Ramakrishnan y Gehrke [2002], Garcia-Molina et al. [2001] y Ullman [1988]. El tratamiento del procesamiento de transacciones en libros de texto se puede encontrar en Bernstein y Newcomer [1997] y Gray y Reuter [1993].

Varios libros incluyen colecciones de artículos de investigación sobre la gestión de las bases de datos. Entre éstos se encuentran Bancilhon y Buneman [1990], Date [1986], Date [1990], Kim [1995], Zaniolo et al. [1997] y Hellerstein y Stonebraker [2005].

Un repaso de los logros en la gestión de bases de datos y una valoración de los desafíos en la investigación futura aparece en Silberschatz et al. [1990], Silberschatz et al. [1996], Bernstein et al. [1998] y Abiteboul et al. [2003]. La página inicial del grupo de interés especial de la ACM en gestión de datos (www.acm.org/sigmod) ofrece gran cantidad de información sobre la investigación en bases de datos. Los sitios Web de los fabricantes de bases de datos (véase a continuación el apartado *Herramientas*) proporciona detalles acerca de sus respectivos productos.

Herramientas

Hay gran número de sistemas de bases de datos comerciales actualmente en uso. Entre los principales están: DB2 de IBM (www.IBM.com/software/data), Oracle (www.oracle.com), SQL Server de Microsoft (www.microsoft.com/SQL), Informix (www.informix.com) (ahora propiedad de IBM) y Sybase (www.sybase.com). Algunos de estos sistemas están disponibles gratuitamente para uso personal o no comercial, o para desarrollo, pero no para su implantación real.

También hay una serie de sistemas de bases de datos gratuitos o de dominio público; los más usados son MySQL (www.mySQL.com) y PostgreSQL (www.postgreSQL.org).

Una lista más completa de enlaces a sitios Web de fabricantes y a otras informaciones se encuentra disponible en la página inicial de este libro, en http://www.mhe.es/universidad/informatica/fundamentos.

PARTE 1

Bases de datos relacionales

Un modelo de datos es un conjunto de herramientas conceptuales para la descripción de los datos, las relaciones entre ellos, su semántica y las restricciones de consistencia. Esta parte centra la atención en el modelo relacional.

El modelo relacional utiliza un conjunto de tablas para representar tanto los datos como las relaciones entre ellos. Su simplicidad conceptual ha conducido a su adopción generalizada; actualmente, una amplia mayoría de los productos de bases de datos se basan en el modelo relacional. En la Parte 9 se expone una visión general de cuatro sistemas de bases de datos relacionales muy utilizados.

Aunque el modelo relacional, que se trata en el Capítulo 2, describe los datos en los niveles lógico y de vistas, es un modelo de datos de nivel inferior comparado con el modelo entidad-relación, que se explica en la Parte 2.

Para poner a disposición de los usuarios los datos de una base de datos relacional hay que abordar varios aspectos. Uno de ellos es determinar uno de los distintos lenguajes de consultas para expresar las peticiones de datos. Los Capítulos 3 y 4 tratan el lenguaje SQL, que es el lenguaje de consultas más utilizado hoy en día. El Capítulo 5 trata en primer lugar dos lenguajes de consultas formales, el cálculo relacional de tuplas y el cálculo relacional de dominios, que son lenguajes declarativos basados en la lógica matemática. Estos dos lenguajes formales constituyen la base para dos lenguajes más amigables para los usuarios: QBE y Datalog, que se estudian en ese capítulo.

Otro aspecto es la integridad y la protección de los datos; las bases de datos deben proteger sus datos del daño que puedan causar las acciones de los usuarios, sean involuntarias o intencionadas. El componente de mantenimiento de la integridad de la base de datos garantiza que las actualizaciones no violen las restricciones de integridad que se hayan especificado para los datos. El componente de protección de la base de datos incluye el control de acceso para restringir las acciones permitidas a cada usuario. El Capítulo 4 trata los problemas de integridad y de protección. Los problemas de protección y de integridad están presentes independientemente del modelo de datos pero, en aras de la concreción, se estudiarán en el contexto del modelo relacional.

CAPÍTULO 2

El modelo relacional

El modelo relacional es hoy en día el principal modelo de datos para las aplicaciones comerciales de procesamiento de datos. Ha conseguido esa posición destacada debido a su simplicidad, lo cual facilita el trabajo del programador en comparación con modelos anteriores, como el de red y el jerárquico.

En este capítulo se estudian en primer lugar los fundamentos del modelo relacional. A continuación se describe el álgebra relacional, que se usa para especificar las solicitudes de información. El álgebra relacional no es cómoda de usar, pero sirve de base formal para lenguajes de consultas que sí lo son y que se estudiarán más adelante, incluido el ampliamente usado lenguaje de consultas SQL, el cual se trata con detalle en los Capítulos 3 y 4.

Existe una amplia base teórica para las bases de datos relacionales. En este capítulo se estudia la parte de esa base teórica referida a las consultas. En los Capítulos 6 y 7 se examinarán aspectos de la teoría de las bases de datos relacionales que ayudan en el diseño de esquemas de bases de datos relacionales, mientras que en los Capítulos 13 y 14 se estudian aspectos de la teoría que se refieren al procesamiento eficiente de consultas.

2.1 La estructura de las bases de datos relacionales

Una base de datos relacional consiste en un conjunto de **tablas**, a cada una de las cuales se le asigna un nombre exclusivo. Cada fila de la tabla representa una *relación* entre un conjunto de valores. De manera informal, cada tabla es un conjunto de entidades, y cada fila es una entidad, tal y como se estudió en el Capítulo 1. Dado que cada tabla es un conjunto de tales relaciones, hay una fuerte correspondencia entre el concepto de *tabla* y el concepto matemático de *relación*, del que toma su nombre el modelo de datos relacional. A continuación se introduce el concepto de relación.

En este capítulo se usarán varias relaciones diferentes para ilustrar los diversos conceptos subyacentes al modelo de datos relacional. Estas relaciones representan parte de una entidad bancaria. Puede que no se correspondan con el modo en que se pueda estructurar realmente una base de datos bancaria, pero así se simplificará la presentación. Los criterios sobre la adecuación de las estructuras relacionales se estudian con gran detalle en los Capítulos 6 y 7.

2.1.1 Estructura básica

Considérese la tabla *cuenta* de la Figura 2.1. Tiene tres cabeceras de columna: *número_cuenta*, *nombre_sucursal* y *saldo*. Siguiendo la terminología del modelo relacional, se puede hacer referencia a estas cabeceras como **atributos**. Para cada atributo hay un conjunto de valores permitidos, denominado **dominio** de ese atributo. Para el atributo *nombre_sucursal*, por ejemplo, el dominio es el conjunto de todos los nombres de sucursal. Supóngase que D_1 denota el conjunto de todos los números de cuenta, D_2 el conjunto de todos los nombres de sucursal y D_3 el conjunto de todos los saldos. Todas las filas de *cuenta* deben consistir en una tupla $(v_1,\ v_2,\ v_3)$, donde v_1 es un número de cuenta (es decir, v_1 está en

número_cuenta	*nombre_sucursal*	*saldo*
C-101	Centro	500
C-102	Navacerrada	400
C-201	Galapagar	900
C-215	Becerril	700
C-217	Galapagar	750
C-222	Moralzarzal	700
C-305	Collado Mediano	350

Figura 2.1 La relación *cuenta*.

el dominio D_1), v_2 es un nombre de sucursal (v_2 en D_2) y v_3 es un saldo (v_3 en D_3). En general, *cuenta* sólo contendrá un subconjunto del conjunto de todas las filas posibles. Por tanto, *cuenta* será un subconjunto de

$$D_1 \times D_2 \times D_3$$

En general, una **tabla** de n atributos debe ser un subconjunto de

$$D_1 \times D_2 \times \cdots \times D_{n-1} \times D_n$$

Los matemáticos definen las **relaciones** como subconjuntos del producto cartesiano de la lista de dominios. Esta definición se corresponde de manera casi exacta con la definición de *tabla* dada anteriormente. La única diferencia es que aquí se han asignado nombres a los atributos, mientras que los matemáticos sólo usan "nombres" numéricos, usando el entero 1 para denotar el atributo cuyo dominio aparece en primer lugar de la lista de dominios, 2 para el atributo cuyo dominio aparece en segundo lugar, etc. Como las tablas son, esencialmente, relaciones, se usarán los términos matemáticos **relación** y **tupla** en lugar de los términos **tabla** y **fila**. Una **variable tupla** es una variable que representa una tupla; en otras palabras, una variable tupla es una variable cuyo dominio es el conjunto de todas las tuplas.

En la relación *cuenta* de la Figura 2.1 hay siete tuplas. Supóngase que la variable tupla t hace referencia a la primera tupla de la relación. Se usa la notación t[*número_cuenta*] para denotar el valor de t en el atributo *número_cuenta*. Por tanto, t[*número_cuenta*] = "C-101" y t[*nombre_sucursal*] = "Centro". De manera alternativa, se puede escribir t[1] para denotar el valor de la tupla t en el primer atributo (*número_cuenta*), t[2] para denotar *nombre_sucursal*, etc. Dado que las relaciones son conjuntos de tuplas, se usa la notación matemática $t \in r$ para denotar que la tupla t está en la relación r.

El orden en que aparecen las tuplas en cada relación es irrelevante, dado que una relación es un *conjunto* de tuplas. Por tanto, no importa si las tuplas de una relación aparecen ordenadas, como en la Figura 2.1, o desordenadas, como en la Figura 2.2; las relaciones de las dos figuras son la misma, ya que las dos contienen el mismo conjunto de tuplas.

Se exige que, para todas las relaciones r, los dominios de todos los atributos de r sean atómicos. Un dominio es **atómico** si los elementos del dominio se consideran unidades indivisibles. Por ejemplo, el conjunto de los enteros es un dominio atómico, pero el conjunto de todos los conjuntos de enteros

número_cuenta	*nombre_sucursal*	*saldo*
C-101	Centro	500
C-215	Becerril	700
C-102	Navacerrada	400
C-305	Collado Mediano	350
C-201	Galapagar	900
C-222	Moralzarzal	700
C-217	Galapagar	750

Figura 2.2 La relación *cuenta* con las tuplas desordenadas.

es un dominio no atómico. La diferencia es que no se suele considerar que los enteros tengan partes constituyentes, pero sí se considera que los conjuntos de enteros las tienen; por ejemplo, los enteros que forman cada conjunto. Lo importante no es lo que sea el propio dominio, sino la manera en que se usan los elementos del dominio en la base de datos. El dominio de todos los enteros sería no atómico si se considerara que cada entero es una lista ordenada de cifras. En todos los ejemplos se supondrá que los dominios son atómicos. En el Capítulo 9 se estudiarán extensiones al modelo de datos relacional para permitir dominios no atómicos.

Es posible que varios atributos tengan el mismo dominio. Por ejemplo, supóngase la relación *cliente* con los tres atributos *nombre_cliente*, *calle_cliente* y *ciudad_cliente* y una relación *empleado* con el atributo *nombre_empleado*. Es posible que los atributos *nombre_cliente* y *nombre_empleado* tengan el mismo dominio, el conjunto de todos los nombres de persona, que en el nivel físico es el conjunto de todas las cadenas de caracteres. Los dominios de *saldo* y *nombre_sucursal*, por otra parte, deberían ser distintos. Quizás sea menos evidente si *nombre_cliente* y *nombre_sucursal* deberían tener el mismo dominio. En el nivel físico, tanto los nombres de los clientes como los nombres de las sucursales son cadenas de caracteres. Sin embargo, en el nivel lógico puede que se desee que *nombre_cliente* y *nombre_sucursal* tengan dominios diferentes.

Un valor de dominio que es miembro de todos los dominios posibles es el valor **nulo**, que indica que el valor es desconocido o no existe. Por ejemplo, supóngase que se incluye el atributo *número_teléfono* en la relación *cliente*. Puede ocurrir que algún cliente no tenga número de teléfono, o que su número de teléfono no figure en la guía. Entonces habrá que recurrir a los valores nulos para indicar que el valor es desconocido o que no existe. Más adelante se verá que los valores nulos crean algunas dificultades cuando se tiene acceso a la base de datos o se la actualiza y que, por tanto, deben eliminarse si es posible. Se supondrá inicialmente que no hay valores nulos y en el Apartado 2.5 se describirá el efecto de los valores nulos en las diferentes operaciones.

2.1.2 Esquema de la base de datos

Cuando se habla de bases de datos se debe diferenciar entre el **esquema de la base de datos**, que es el diseño lógico de la misma, y el **ejemplar de la base de datos**, que es una instantánea de los datos de la misma en un momento dado.

El concepto de relación se corresponde con el concepto de variable de los lenguajes de programación. El concepto de **esquema de la relación** se corresponde con el concepto de definición de tipos de los lenguajes de programación.

Resulta conveniente dar nombre a los esquemas de las relaciones, igual que se dan nombres a las definiciones de los tipos en los lenguajes de programación. Se adopta el convenio de usar nombres en minúsculas para las relaciones y nombres que comiencen por una letra mayúscula para los esquemas de las relaciones. Siguiendo esta notación se usará *Esquema_cuenta* para denotar el esquema de la relación *cuenta*. Por tanto,

$$Esquema_cuenta = (\textit{número_cuenta}, \textit{nombre_sucursal}, \textit{saldo})$$

Se denota el hecho de que *cuenta* es una relación de *Esquema_cuenta* mediante

$$cuenta(Esquema_cuenta)$$

En general, los esquemas de las relaciones consisten en una lista de los atributos y de sus dominios correspondientes. La definición exacta del dominio de cada atributo no será relevante hasta que se estudie el lenguaje SQL en los Capítulos 3 y 4.

El concepto de **ejemplar de la relación** se corresponde con el concepto de valor de una variable en los lenguajes de programación. El valor de una variable dada puede cambiar con el tiempo; de manera parecida, el contenido del ejemplar de una relación puede cambiar con el tiempo cuando la relación se actualiza. Sin embargo, se suele decir simplemente "relación" cuando realmente se quiere decir "ejemplar de la relación".

Como ejemplo de ejemplar de una relación, considérese la relación *sucursal* de la Figura 2.3. El esquema de esa relación es

nombre_sucursal	*ciudad_sucursal*	*activos*
Becerril	Aluche	400.000
Centro	Arganzuela	9.000.000
Collado Mediano	Aluche	8.000.000
Galapagar	Arganzuela	7.100.000
Moralzarzal	La Granja	2.100.000
Navacerrada	Aluche	1.700.000
Navas de la Asunción	Alcalá de Henares	300.000
Segovia	Cerceda	3.700.000

Figura 2.3 La relación *sucursal*.

$$Esquema_sucursal = (\textit{nombre_sucursal}, \textit{ciudad_sucursal}, \textit{activos})$$

Obsérvese que el atributo *nombre_sucursal* aparece tanto en *Esquema_sucursal* como en *Esquema_cuenta*. Esta duplicidad no es una coincidencia. Más bien, usar atributos comunes en los esquemas de las relaciones es una manera de relacionar las tuplas de relaciones diferentes. Por ejemplo, supóngase que se desea obtener información sobre todas las cuentas abiertas en sucursales ubicadas en Arganzuela. Primero se busca en la relación *sucursal* para encontrar los nombres de todas las sucursales situadas en Arganzuela. A continuación y para cada una de ellas, se examina la relación *cuenta* para encontrar la información sobre las cuentas abiertas en esa sucursal.

Siguiendo con el ejemplo bancario, se necesita una relación que describa información sobre los clientes. El esquema de la relación es:

$$Esquema_cliente = (\textit{nombre_cliente}, \textit{calle_cliente}, \textit{ciudad_cliente})$$

La Figura 2.4 muestra un ejemplo de la relación *cliente* (*Esquema_cliente*). Obsérvese que se ha omitido el atributo *id_cliente*, que se usó en el Capítulo 1, ya que se considerarán esquemas de relación más pequeños en nuestro ejemplo de una base de datos bancaria. Se da por supuesto que el nombre de cliente identifica unívocamente a cada cliente—obviamente, puede que esto no sea cierto en el mundo real, pero la suposición hace los ejemplos más sencillos de entender. En una base de datos del mundo real, *id_cliente* (que podría ser *número_seguridad_social* o un identificador generado por el banco) serviría para identificar unívocamente a los clientes.

También se necesita una relación que describa la asociación entre los clientes y las cuentas. El esquema de la relación que describe esta asociación es:

$$Esquema_impositor = (\textit{nombre_cliente}, \textit{número_cuenta})$$

nombre_cliente	*calle_cliente*	*ciudad_cliente*
Abril	Preciados	Valsaín
Amo	Embajadores	Arganzuela
Badorrey	Delicias	Valsaín
Fernández	Jazmín	León
Gómez	Carretas	Cerceda
González	Arenal	La Granja
López	Mayor	Peguerinos
Pérez	Carretas	Cerceda
Rodríguez	Yeserías	Cádiz
Rupérez	Ramblas	León
Santos	Mayor	Peguerinos
Valdivieso	Goya	Vigo

Figura 2.4 La relación *cliente*.

nombre_cliente	*número_cuenta*
Abril	C-305
Gómez	C-215
González	C-101
González	C-201
López	C-102
Rupérez	C-222
Santos	C-217

Figura 2.5 La relación *impositor*.

La Figura 2.5 muestra un ejemplo de la relación *impositor* (*Esquema_impositor*).

Puede parecer que para este ejemplo bancario se podría tener sólo un esquema de relación, en vez de tener varios. Es decir, puede resultar más sencillo para el usuario pensar en términos de un único esquema de relación, en lugar de en varios esquemas. Supóngase que sólo se usara una relación para el ejemplo, con el esquema

(*nombre_sucursal*, *ciudad_sucursal*, *activos*, *nombre_cliente*, *calle_cliente*
ciudad_cliente, *número_cuenta*, *saldo*)

Obsérvese que, si un cliente tiene varias cuentas, hay que repetir su dirección una vez por cada cuenta. Es decir, hay que repetir varias veces parte de la información. Esta repetición malgasta espacio, pero se evita mediante el empleo de varias relaciones mostradas anteriormente.

Además, si una sucursal no tiene ninguna cuenta (por ejemplo, una sucursal recién abierta que todavía no tenga clientes), no se puede construir una tupla completa en la relación única anterior, dado que no hay todavía ningún dato disponible referente a *cliente* ni a *cuenta*. Para representar las tuplas incompletas hay que usar valores *nulos* que indiquen que ese valor es desconocido o no existe. Por tanto, en el ejemplo presente, los valores de *nombre_cliente*, *calle_cliente*, etc., deben ser nulos. Al emplear varias relaciones se puede representar la información de las sucursales del banco sin clientes sin necesidad de valores nulos. Se usa simplemente una tupla en *Esquema_sucursal* para representar la información de la sucursal, y sólo se crean tuplas en los otros esquemas cuando esté disponible la información correspondiente.

En el Capítulo 7 se estudiarán los criterios para decidir cuándo un conjunto de esquemas de relaciones es más adecuado que otro en términos de repetición de la información y de la existencia de valores nulos. Por ahora se supondrá que los esquemas de las relaciones vienen dados de antemano.

Se incluyen dos relaciones más para describir los datos de los préstamos concedidos en las diferentes sucursales del banco:

Esquema_préstamo = (*número_préstamo*, *nombre_sucursal*, *importe*)
Esquema_prestatario = (*nombre_cliente*, *número_préstamo*)

Las Figuras 2.6 y 2.7, respectivamente, muestran las relaciones de ejemplo *préstamo* (*Esquema_préstamo*) y *prestatario* (*Esquema_prestatario*).

Los esquemas de relación se corresponden con el conjunto de tablas que podrían generarse con el método descrito en el Apartado 1.6. Obsérvese que la relación *cliente* puede contener información sobre clientes que no tengan ni cuenta ni préstamo en el banco. La entidad bancaria aquí descrita servirá como ejemplo principal en este capítulo. Cuando sea necesario, habrá que introducir más esquemas de relaciones para ilustrar casos concretos.

2.1.3 Claves

Es necesario disponer de un modo de especificar la manera en que las tuplas de una relación dada se distingan entre sí. Esto se expresa en términos de sus atributos. Es decir, los valores de los valores de los atributos de una tupla deben ser tales que puedan *identificarla unívocamente*. En otras palabras, no

número_préstamo	*nombre_sucursal*	*importe*
P-11	Collado Mediano	900
P-14	Centro	1.500
P-15	Navacerrada	1.500
P-16	Navacerrada	1.300
P-17	Centro	1.000
P-23	Moralzarzal	2.000
P-93	Becerril	500

Figura 2.6 La relación *préstamo*.

se permite que dos tuplas de una misma relación tengan exactamente los mismos valores en todos sus atributos.

Una **superclave** es un conjunto de uno o varios atributos que, considerados conjuntamente, permiten identificar de manera unívoca una tupla de la relación. Por ejemplo, el atributo *id_cliente* de la relación *cliente* es suficiente para distinguir una tupla *cliente* de otra. Por tanto, *id_cliente* es una superclave. De manera parecida, la combinación de *nombre_cliente* e *id_cliente* constituye una superclave para la relación *cliente*. El atributo *nombre_cliente* de *cliente* no es una superclave, ya que es posible que varias personas se llamen igual.

El concepto de superclave no es suficiente para nuestros propósitos, ya que, como se ha podido ver, las superclaves pueden contener atributos innecesarios. Si C es una superclave, entonces también lo es cualquier superconjunto de C. A menudo resultan interesantes superclaves para las que ninguno de sus subconjuntos constituya una superclave. Esas superclaves mínimas se denominan **claves candidatas**.

Es posible que varios conjuntos diferentes de atributos puedan ejercer como claves candidatas. Supóngase que una combinación de *nombre_cliente* y de *calle_cliente* sea suficiente para distinguir entre los miembros de la relación *cliente*. Entonces, tanto {*id_cliente*} como {*nombre_cliente, calle_cliente*} son claves candidatas. Aunque los atributos *id_cliente* y *nombre_cliente* en conjunto pueden diferenciar las tuplas *cliente*, su combinación no forma una clave candidata, ya que el atributo *id_cliente* por sí solo ya lo es.

Se usará el término **clave primaria** para denotar una clave candidata que ha elegido el diseñador de la base de datos como medio principal para la identificación de las tuplas de una relación. Las claves (sean primarias, candidatas o superclaves) son propiedades de toda la relación, no de cada una de las tuplas. Ninguna pareja de tuplas de la relación puede tener simultáneamente el mismo valor de los atributos de la clave. La selección de una clave representa una restricción de la empresa del mundo real que se está modelando.

Las claves candidatas deben escogerse con cuidado. Como se ha indicado, el nombre de una persona evidentemente no es suficiente, ya que puede haber mucha gente con el mismo nombre. En Estados Unidos el atributo número de la seguridad social de cada persona sería clave candidata. Dado que los residentes extranjeros no suelen tener número de la seguridad social, las empresas internacionales deben generar sus propios identificadores unívocos. Una alternativa es usar como clave alguna combinación exclusiva de otros atributos.

nombre_cliente	*número_préstamo*
Fernández	P-16
Gómez	P-11
Gómez	P-23
López	P-15
Pérez	P-93
Santos	P-17
Sotoca	P-14
Valdivieso	P-17

Figura 2.7 La relación *prestatario*.

La clave primaria debe escogerse de manera que los valores de sus atributos no se modifiquen nunca, o muy rara vez. Por ejemplo, el campo domicilio de una persona no debe formar parte de la clave primaria, ya que es probable que se modifique. Por otra parte, está garantizado que los números de la seguridad social no cambian nunca. Los identificadores exclusivos generados por las empresas no suelen cambiar, salvo si se produce una fusión entre dos de ellas; en ese caso, puede que el mismo identificador haya sido emitido por ambas empresas, y puede ser necesaria una reasignación de identificadores para garantizar que sean únicos.

Formalmente, sea R el esquema de una relación. Si se dice que un subconjunto C de R es una *superclave* de R, se restringe la consideración a las relaciones $r(R)$ en las que no hay dos tuplas diferentes que tengan los mismos valores en todos los atributos de C. Es decir, si t_1 y t_2 están en r y $t_1 \neq t_2$, entonces $t_1[C] \neq t_2[C]$.

El esquema de una relación, por ejemplo r_1, puede incluir entre sus atributos la clave primaria de otro esquema de relación, por ejemplo r_2. Este atributo se denomina **clave externa** de r_1, que hace referencia a r_2. La relación r_1 también se denomina **relación referenciante** de la dependencia de clave externa, y r_2 se denomina **relación referenciada** de la clave externa. Por ejemplo, el atributo *nombre_sucursal* de *Esquema_cuenta* es una clave externa de *Esquema_cuenta* que hace referencia a *Esquema_sucursal*, ya que *nombre_sucursal* es la clave primaria de *Esquema_sucursal*. En cualquier ejemplar de la base de datos, dada cualquier tupla, por ejemplo t_a, de la relación *cuenta*, debe haber alguna tupla, por ejemplo t_b, en la relación *sucursal* tal que el valor del atributo *nombre_sucursal* de t_a sea el mismo que el valor de la clave primaria de t_b *nombre_sucursal*.

Es costumbre relacionar los atributos de la clave primaria de un esquema de relación antes que el resto de los atributos; por ejemplo, el atributo *nombre_sucursal* de *Esquema_sucursal* se relaciona en primer lugar, ya que es la clave primaria.

El esquema de la base de datos, junto con las dependencias de clave primaria y externa, se puede mostrar gráficamente mediante **diagramas de esquema**. La Figura 2.8 muestra el diagrama de esquema del ejemplo bancario. Cada relación aparece como un cuadro con los atributos relacionados en su interior y el nombre de la relación sobre él. Si hay atributos de clave primaria, una línea horizontal cruza el cuadro con los atributos de clave primaria por encima de ella y sobre fondo gris. Las dependencias de clave externa aparecen como flechas desde los atributos de clave externa de la relación referenciante a la clave primaria de la relación referenciada.

Muchos sistemas de bases de datos proporcionan herramientas de diseño con una interfaz gráfica de usuario para la creación de los diagramas de esquema.

2.1.4 Lenguajes de consultas

Un **lenguaje de consultas** es un lenguaje en el que los usuarios solicitan información de la base de datos. Estos lenguajes suelen ser de un nivel superior que el de los lenguajes de programación habituales. Los lenguajes de consultas pueden clasificarse como procedimentales o no procedimentales. En los **lenguajes procedimentales** el usuario indica al sistema que lleve a cabo una serie de operaciones en la base

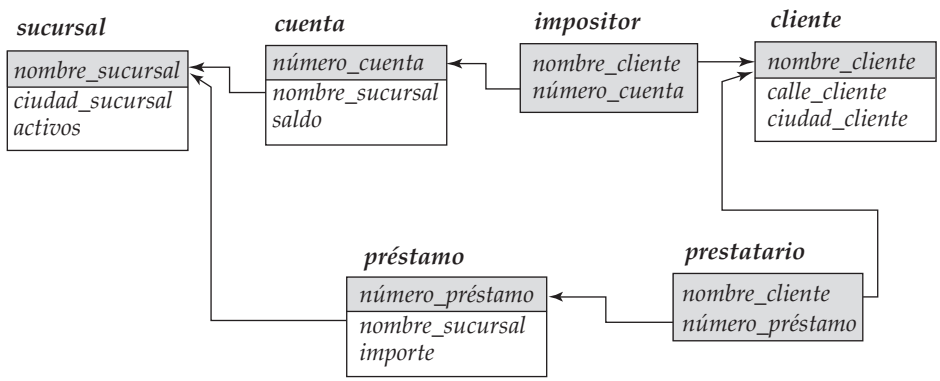


Figura 2.8 Diagrama del esquema de la entidad bancaria.

de datos para calcular el resultado deseado. En los **lenguajes no procedimentales** el usuario describe la información deseada sin dar un procedimiento concreto para obtener esa información.

La mayor parte de los sistemas comerciales de bases de datos relacionales ofrecen un lenguaje de consultas que incluye elementos de los enfoques procedimental y no procedimental. Se estudiará el muy usado lenguaje de consultas SQL en los Capítulos 3 y 4. El Capítulo 5 trata los lenguajes de consultas QBE y Datalog; este último es un lenguaje de consultas parecido al lenguaje de programación Prolog.

Existen varios lenguajes de consultas "puros": el álgebra relacional es procedimental, mientras que el cálculo relacional de tuplas y el cálculo relacional de dominios no lo son. Estos lenguajes de consultas son rígidos y formales, y carecen del "azúcar sintáctico" de los lenguajes comerciales, pero ilustran las técnicas fundamentales para la extracción de datos de las bases de datos.

En este capítulo se examina con gran detalle el lenguaje del álgebra relacional (en el Capítulo 5 se tratan los lenguajes del cálculo relacional de tuplas y del cálculo relacional de dominios). El álgebra relacional consiste en un conjunto de operaciones que toman una o dos relaciones como entrada y generan otra relación nueva como resultado.

Las operaciones fundamentales del álgebra relacional son *selección, proyección*, *unión*, *diferencia de conjuntos*, *producto cartesiano* y *renombramiento*. Además de las operaciones fundamentales hay otras operaciones—por ejemplo, intersección de conjuntos, reunión natural, división y asignación. Estas operaciones se definirán en términos de las operaciones fundamentales.

Inicialmente sólo se estudiarán las consultas. Sin embargo, un lenguaje de manipulación de datos completo no sólo incluye un lenguaje de consultas, sino también un lenguaje para la modificación de las bases de datos. Este tipo de lenguajes incluye comandos para insertar y borrar tuplas, así como para modificar partes de las tuplas existentes. Las modificaciones de las bases de datos se examinarán después de completar la discusión sobre las consultas.

2.2 Operaciones fundamentales del álgebra relacional

Las operaciones selección, proyección y renombramiento se denominan operaciones *unarias* porque operan sobre una sola relación. Las otras tres operaciones operan sobre pares de relaciones y se denominan, por tanto, operaciones *binarias*.

2.2.1 Operación selección

La operación **selección** selecciona tuplas que satisfacen un predicado dado. Se usa la letra griega sigma minúscula (σ) para denotar la selección. El predicado aparece como subíndice de σ. La relación de argumentos se da entre paréntesis a continuación de σ. Por tanto, para seleccionar las tuplas de la relación *préstamo* en que la sucursal es "Navacerrada" se escribe

$$\sigma_{nombre_sucursal = \text{"Navacerrada"}} (préstamo)$$

Si la relación *préstamo* es como se muestra en la Figura 2.6, la relación que resulta de la consulta anterior es como aparece en la Figura 2.9.

número_préstamo	*nombre_sucursal*	*importe*
P-15	Navacerrada	1.500
P-16	Navacerrada	1.300

Figura 2.9 Resultado de $\sigma_{nombre_sucursal = \text{"Navacerrada"}} (préstamo)$.

Se pueden buscar todas las tuplas en las que el importe prestado sea mayor que 1.200 € escribiendo la siguiente consulta:

$$\sigma_{importe>1200}\ (préstamo)$$

En general, se permiten las comparaciones que usan $=$, $\neq$, $<$, $\leq$, $>$ o $\geq$ en el predicado de selección. Además, se pueden combinar varios predicados en uno mayor usando las conectivas *y* ($\wedge$), *o* ($\vee$) y *no* ($\neg$). Por tanto, para encontrar las tuplas correspondientes a préstamos de más de 1.200 € concedidos por la sucursal de Navacerrada, se escribe

$$\sigma_{nombre_sucursal = \text{“Navacerrada”} \wedge importe>1200}\ (préstamo)$$

El predicado de selección puede incluir comparaciones entre dos atributos. Para ilustrarlo, considérese la relación *responsable_préstamo*, que consta de tres atributos: *nombre_cliente*, *nombre_responsable* y *número_préstamo*, que especifica que un empleado concreto es el responsable del préstamo concedido a un cliente. Para hallar todos los clientes que se llaman igual que su responsable de préstamos se puede escribir

$$\sigma_{nombre_cliente = nombre_responsable}(responsable_préstamo)$$

2.2.2 Operación proyección

Supóngase que se desea obtener una relación de todos los números e importes de los préstamos, pero sin los nombres de las sucursales. La operación **proyección** permite obtener esa relación. La operación proyección es una operación unaria que devuelve su relación de argumentos, excluyendo algunos argumentos. Dado que las relaciones son conjuntos, se eliminan todas las filas duplicadas. La proyección se denota por la letra griega mayúscula pi (Π). Se crea una lista de los atributos que se desea que aparezcan en el resultado como subíndice de Π. Su único argumento, una relación, se escribe a continuación entre paréntesis. La consulta para crear una lista de todos los números e importes de los préstamos puede escribirse como

$$\Pi_{número_préstamo, importe}(préstamo)$$

La Figura 2.10 muestra la relación que resulta de esta consulta.

2.2.3 Composición de operaciones relacionales

Es importante el hecho de que el resultado de una operación relacional sea también una relación. Considérese la consulta más compleja “Buscar los clientes que viven en Peguerinos”. Hay que escribir:

$$\Pi_{nombre_cliente}\ (\sigma_{ciudad_cliente = \text{“Peguerinos”}}\ (cliente))$$

Téngase en cuenta que, en vez de dar el nombre de una relación como argumento de la operación proyección, se da una expresión cuya evaluación es una relación.

En general, dado que el resultado de las operaciones del álgebra relacional es del mismo tipo (relación) que los datos de entrada, las operaciones del álgebra relacional pueden componerse para formar

número_préstamo	*importe*
P-11	900
P-14	1.500
P-15	1.500
P-16	1.300
P-17	1.000
P-23	2.000
P-93	500

Figura 2.10 Números e importes de los préstamos.

una **expresión del álgebra relacional**. Componer operaciones del álgebra relacional para formar expresiones del álgebra relacional es igual que componer operaciones aritméticas (como $+$, $-$, $*$ y $\div$) para formar expresiones aritméticas. La definición formal de las expresiones de álgebra relacional se estudia en el Apartado 2.2.8.

2.2.4 Operación unión

Considérese una consulta para determinar el nombre de todos los clientes del banco que tienen una cuenta, un préstamo o ambas cosas. Obsérvese que la relación *cliente* no contiene esa información, dado que los clientes no necesitan tener ni cuenta ni préstamo en el banco. Para contestar a esta consulta hace falta la información de las relaciones *impositor* (Figura 2.5) y *prestatario* (Figura 2.7). Para determinar los nombres de todos los clientes con préstamos en el banco con las operaciones estudiadas se escribe:

$$\Pi_{nombre_cliente}\ (prestatario)$$

Igualmente, para determinar el nombre de todos los clientes con cuenta en el banco se escribe:

$$\Pi_{nombre_cliente}\ (impositor)$$

Para contestar a la consulta es necesaria la **unión** de estos dos conjuntos; es decir, hacen falta todos los nombres de clientes que aparecen en alguna de las dos relaciones o en ambas. Estos datos se pueden averiguar mediante la operación binaria unión, denotada, como en la teoría de conjuntos, por $\cup$. Por tanto, la expresión buscada es:

$$\Pi_{nombre_cliente}\ (prestatario)\ \cup\ \Pi_{nombre_cliente}\ (impositor)$$

La relación resultante de esta consulta aparece en la Figura 2.11. Téngase en cuenta que en el resultado hay diez tuplas, aunque haya siete prestatarios y seis impositores distintos. Esta discrepancia aparente se debe a que Gómez, López y Santos son a la vez prestatarios e impositores. Dado que las relaciones son conjuntos, se eliminan los valores duplicados.

Obsérvese que en este ejemplo se toma la unión de dos conjuntos, ambos consistentes en valores de *nombre_cliente*. En general, se debe asegurar que las uniones se realicen entre relaciones *compatibles*. Por ejemplo, no tendría sentido realizar la unión de las relaciones *préstamo* y *prestatario*. La primera es una relación con tres atributos, la segunda sólo tiene dos. Más aún, considérese la unión de un conjunto de nombres de clientes y de un conjunto de ciudades. Una unión así no tendría sentido en la mayor parte de los casos. Por tanto, para que la operación unión $r \cup s$ sea válida hay que exigir que se cumplan dos condiciones:

1. Las relaciones r y s deben ser de la misma aridad. Es decir, deben tener el mismo número de atributos.
2. Los dominios de los atributos i-ésimos de r y de s deben ser iguales para todo i.

nombre_cliente
Abril
Fernández
Gómez
González
López
Pérez
Rupérez
Santos
Sotoca
Valdivieso

Figura 2.11 Nombre de todos los clientes que tienen un préstamo o una cuenta.

nombre_cliente
Abril
González
Rupérez

Figura 2.12 Clientes con cuenta abierta pero sin préstamo concedido.

Téngase en cuenta que *r* y *s* pueden ser, en general, relaciones de la base de datos o relaciones temporales resultado de expresiones del álgebra relacional.

2.2.5 Operación diferencia de conjuntos

La operación **diferencia de conjuntos**, denotada por $-$, permite hallar las tuplas que están en una relación pero no en la otra. La expresión $r - s$ da como resultado una relación que contiene las tuplas que están en *r* pero no en *s*.

Se pueden buscar todos los clientes del banco que tengan abierta una cuenta pero no tengan concedido ningún préstamo escribiendo

$$\Pi_{nombre_cliente}\ (impositor) - \Pi_{nombre_cliente}\ (prestatario)$$

La relación resultante de esta consulta aparece en la Figura 2.12.

Como en el caso de la operación unión, hay que asegurarse de que las diferencias de conjuntos se realicen entre relaciones *compatibles*. Por tanto, para que una operación diferencia de conjuntos $r - s$ sea válida se exige que las relaciones *r* y *s* sean de la misma aridad y que los dominios de los atributos *i*-ésimos de *r* y de *s* sean iguales.

2.2.6 Operación producto cartesiano

La operación **producto cartesiano**, denotada por un aspa ($\times$), permite combinar información de cualesquiera dos relaciones. El producto cartesiano de las relaciones r_1 y r_2 se escribe $r_1 \times r_2$.

Recuérdese que las relaciones se definen como subconjuntos del producto cartesiano de un conjunto de dominios. A partir de esa definición ya se debe tener una intuición sobre la definición de la operación producto cartesiano. Sin embargo, dado que el mismo nombre de atributo puede aparecer tanto en r_1 como en r_2, es necesario crear un convenio de denominación para distinguir unos atributos de otros. En este caso se realiza adjuntando al atributo el nombre de la relación de la que proviene originalmente. Por ejemplo, el esquema de relación de $r = prestatario \times préstamo$ es

(*prestatario.nombre_cliente*, *prestatario.número_préstamo*,
préstamo.número_préstamo, *préstamo.nombre_sucursal*, *préstamo.importe*)

Con este esquema se puede distinguir entre *prestatario.número_préstamo* y *préstamo.número_préstamo*. Para los atributos que sólo aparecen en uno de los dos esquemas se suele omitir el prefijo con el nombre de la relación. Esta simplificación no genera ambigüedad alguna. Por tanto, se puede escribir el esquema de la relación *r* como

(*nombre_cliente*, *prestatario.número_préstamo*,
préstamo.número_préstamo, *nombre_sucursal*, *importe*)

Este convenio de denominaciones *exige* que las relaciones que sean argumentos de la operación producto cartesiano tengan nombres diferentes. Esta exigencia causa problemas en algunos casos, como cuando se desea calcular el producto cartesiano de una relación consigo misma. Se produce un problema parecido si se usa el resultado de una expresión del álgebra relacional en un producto cartesiano, dado que hará falta un nombre de relación para poder hacer referencia a sus atributos. En el Apartado 2.2.7 se verá la manera de evitar estos problemas mediante la operación renombramiento.

Ahora que se conoce el esquema de relación de *r* = *prestatario* $\times$ *préstamo* es necesario hallar las tuplas que aparecerán en *r*. Como es posible imaginar, se crea una tupla de *r* a partir de cada par de tuplas

nombre_cliente	*prestatario. número_préstamo*	*préstamo. número_préstamo*	*nombre_sucursal*	*importe*
Fernández	P-16	P-11	Collado Mediano	900
Fernández	P-16	P-14	Centro	1.500
Fernández	P-16	P-15	Navacerrada	1.500
Fernández	P-16	P-16	Navacerrada	1.300
Fernández	P-16	P-17	Centro	1.000
Fernández	P-16	P-23	Moralzarzal	2.000
Fernández	P-16	P-93	Becerril	500
Gómez	P-11	P-11	Collado Mediano	900
Gómez	P-11	P-14	Centro	1.500
Gómez	P-11	P-15	Navacerrada	1.500
Gómez	P-11	P-16	Navacerrada	1.300
Gómez	P-11	P-17	Centro	1.000
Gómez	P-11	P-23	Moralzarzal	2.000
Gómez	P-11	P-93	Becerril	500
Gómez	P-23	P-11	Collado Mediano	900
Gómez	P-23	P-14	Centro	1.500
Gómez	P-23	P-15	Navacerrada	1.500
Gómez	P-23	P-16	Navacerrada	1.300
Gómez	P-23	P-17	Centro	1.000
Gómez	P-23	P-23	Moralzarzal	2.000
Gómez	P-23	P-93	Becerril	500
...	...	...	...	...
...	...	...	...	...
...	...	...	...	...
Sotoca	P-14	P-11	Collado Mediano	900
Sotoca	P-14	P-14	Centro	1.500
Sotoca	P-14	P-15	Navacerrada	1.500
Sotoca	P-14	P-16	Navacerrada	1.300
Sotoca	P-14	P-17	Centro	1.000
Sotoca	P-14	P-23	Moralzarzal	2.000
Sotoca	P-14	P-93	Becerril	500
Valdivieso	P-17	P-11	Collado Mediano	900
Valdivieso	P-17	P-14	Centro	1.500
Valdivieso	P-17	P-15	Navacerrada	1.500
Valdivieso	P-17	P-16	Navacerrada	1.300
Valdivieso	P-17	P-17	Centro	1.000
Valdivieso	P-17	P-23	Moralzarzal	2.000
Valdivieso	P-17	P-93	Becerril	500

Figura 2.13 Resultado de $prestatario \times préstamo$.

posible: una de la relación *prestatario* y otra de *préstamo*. Por tanto, *r* es una relación de gran tamaño, como se puede ver en la Figura 2.13, donde sólo se ha incluido una parte de las tuplas que constituyen *r*.

Supóngase que se tienen n_1 tuplas de *prestatario* y n_2 tuplas de *préstamo*. Por tanto, hay $n_1 * n_2$ maneras de escoger un par de tuplas—una tupla de cada relación; por lo que hay $n_1 * n_2$ tuplas en *r*. En concreto, obsérvese que para algunas tuplas *t* de *r* puede ocurrir que *t*[*prestatario.número_préstamo*] $\neq$ *t*[*préstamo.número_préstamo*].

En general, si se tienen las relaciones $r_1(R_1)$ y $r_2(R_2)$, $r_1 \times r_2$ es una relación cuyo esquema es la concatenación de R_1 y de R_2. La relación *R* contiene todas las tuplas *t* para las que hay unas tuplas t_1 en r_1 y t_2 en r_2 para las que $t[R_1] = t_1[R_1]$ y $t[R_2] = t_2[R_2]$.

nombre_cliente	*prestatario. número_préstamo*	*préstamo. número_préstamo*	*nombre_sucursal*	*importe*
Fernández	P-16	P-15	Navacerrada	1.500
Fernández	P-16	P-16	Navacerrada	1.300
Gómez	P-11	P-15	Navacerrada	1.500
Gómez	P-11	P-16	Navacerrada	1.300
Gómez	P-23	P-15	Navacerrada	1.500
Gómez	P-23	P-16	Navacerrada	1.300
López	P-15	P-15	Navacerrada	1.500
López	P-15	P-16	Navacerrada	1.300
Pérez	P-93	P-15	Navacerrada	1.500
Pérez	P-93	P-16	Navacerrada	1.300
Santos	P-17	P-15	Navacerrada	1.500
Santos	P-17	P-16	Navacerrada	1.300
Sotoca	P-14	P-15	Navacerrada	1.500
Sotoca	P-14	P-16	Navacerrada	1.300
Valdivieso	P-17	P-15	Navacerrada	1.500
Valdivieso	P-17	P-16	Navacerrada	1.300

Figura 2.14 Resultado de $\sigma_{nombre_sucursal = \text{“Navacerrada”}} (prestatario \times préstamo)$.

Supóngase que se desea determinar el nombre de todos los clientes que tienen concedido un préstamo en la sucursal de Navacerrada. Se necesita para ello información de las relaciones *préstamo* y *prestatario*. Si se escribe

$$\sigma_{nombre_sucursal = \text{“Navacerrada”}} (prestatario \times préstamo)$$

entonces el resultado es la relación mostrada en la Figura 2.14. Se tiene una relación que sólo atañe a la sucursal de Navacerrada. Sin embargo, la columna *nombre_cliente* puede contener clientes que no tengan concedido ningún préstamo en la sucursal de Navacerrada. (Si no se ve el motivo por el que esto es cierto, recuérdese que el producto cartesiano toma todos los emparejamientos posibles de cada tupla de *prestatario* con cada tupla de *préstamo*).

Dado que la operación producto cartesiano asocia *todas* las tuplas de *préstamo* con todas las tuplas de *prestatario*, se sabe que, si un cliente tiene concedido un préstamo en la sucursal de Navacerrada, hay alguna tupla de *prestatario* $\times$ *préstamo* que contiene su nombre y que *prestatario.número_préstamo* = *préstamo.número_préstamo*. Por tanto, si se escribe

$$\sigma_{prestatario.número_préstamo = préstamo.número_préstamo} (\sigma_{nombre_sucursal = \text{“Navacerrada”}} (prestatario \times préstamo))$$

sólo se obtienen las tuplas de $prestatario \times préstamo$ que corresponden a los clientes que tienen concedido un préstamo en la sucursal de Navacerrada.

Finalmente, dado que sólo se desea obtener *nombre_cliente*, se realiza una proyección:

$$\Pi_{nombre_cliente} (\sigma_{prestatario.número_préstamo = préstamo.número_préstamo} (\sigma_{nombre_sucursal = \text{“Navacerrada”}} (prestatario \times préstamo)))$$

El resultado de esta expresión, mostrada en la Figura 2.15, es la respuesta correcta a la consulta formulada.

2.2.7 Operación renombramiento

A diferencia de las relaciones de la base de datos, los resultados de las expresiones de álgebra relacional no tienen un nombre que se pueda usar para referirse a ellas. Resulta útil poder ponerles nombre; la operación **renombramiento**, denotada por la letra griega ro minúscula (ρ), permite hacerlo. Dada una

nombre_cliente
Fernández
López

Figura 2.15 Resultado de $\Pi_{nombre_cliente}$ $(\sigma_{prestatario.número_préstamo = préstamo.número_préstamo}$ $(\sigma_{nombre_sucursal = \text{“Navacerrada”}} (prestatario \times préstamo)))$.

expresión E del álgebra relacional, la expresión

$$\rho_x (E)$$

devuelve el resultado de la expresión *E* con el nombre *x*.

Las relaciones r por sí mismas se consideran expresiones (triviales) del álgebra relacional. Por tanto, también se puede aplicar la operación renombramiento a una relación r para obtener la misma relación con un nombre nuevo.

Otra forma de la operación renombramiento es la siguiente. Supóngase que una expresión del álgebra relacional E tiene aridad n. Por tanto, la expresión

$$\rho_{x(A_1,A_2,\ldots,A_n)} (E)$$

devuelve el resultado de la expresión *E* con el nombre *x* y con los atributos con el nombre cambiado a $A_1, A_2, \ldots, A_n$.

Para ilustrar el renombramiento de relaciones, considérese la consulta “Buscar el saldo máximo de las cuentas del banco”. La estrategia empleada para obtener el resultado es (1) calcular en primer lugar una relación temporal consistente en los saldos que *no* son el máximo y (2) realizar la diferencia entre la relación $\Pi_{saldo} (cuenta)$ y la relación temporal recién calculada para obtener el resultado.

Paso 1: para calcular la relación intermedia hay que comparar los valores de los saldos de todas las cuentas. Esta comparación se hará calculando el producto cartesiano $cuenta \times cuenta$ y formando una selección para comparar el valor de cualesquiera dos saldos que aparezcan en una tupla. En primer lugar hay que crear un mecanismo para distinguir entre los dos atributos *saldo*. Se usará la operación renombramiento para cambiar el nombre de una referencia a la relación *cuenta*; de este modo se puede hacer referencia dos veces a la relación sin ambigüedad alguna.

La relación temporal que se compone de los saldos que no son el máximo puede escribirse ahora como:

$$\Pi_{cuenta.saldo} (\sigma_{cuenta.saldo < d.saldo} (cuenta \times \rho_d (cuenta)))$$

Esta expresión proporciona los saldos de la relación *cuenta* para los que aparece un saldo mayor en alguna parte de la relación *cuenta* (cuyo nombre se ha cambiado a *d*). El resultado contiene todos los saldos *salvo* el máximo. La Figura 2.16 muestra esta relación.

Paso 2: la consulta para determinar el saldo máximo de las cuentas del banco puede escribirse de la manera siguiente:

$$\Pi_{saldo} (cuenta) - \Pi_{cuenta.saldo} (\sigma_{cuenta.saldo < d.saldo} (cuenta \times \rho_d (cuenta)))$$

saldo
350
400
500
700
750

Figura 2.16 Resultado de la subexpresión $\Pi_{cuenta.saldo} (\sigma_{cuenta.saldo < d.saldo} (cuenta \times \rho_d (cuenta)))$.

saldo
900

Figura 2.17 Saldo máximo de las cuentas del banco.

La Figura 2.17 muestra el resultado de esta consulta.

Considérese la consulta "Determinar el nombre de todos los clientes que viven en la misma ciudad y en la misma calle que Gómez" como un nuevo ejemplo de la operación renombramiento. Se puede obtener la calle y la ciudad en las que vive Gómez escribiendo

$$\Pi_{calle_cliente,\ ciudad_cliente}\ (\sigma_{nombre_cliente\ =\ \text{"Gómez"}}\ (cliente))$$

Sin embargo, para hallar a otros clientes que vivan en esa ciudad y en esa calle hay que hacer referencia por segunda vez a la relación *cliente*. En la consulta siguiente se usa la operación renombramiento sobre la expresión anterior para darle al resultado el nombre *dirección_gómez* y para cambiar el nombre de los atributos *calle_cliente* y *ciudad_cliente* a *calle* y *ciudad*, respectivamente:

$$\begin{array}{l}\Pi_{cliente.nombre_cliente} \\ \quad (\sigma_{cliente.calle_cliente=dirección_gómez.calle\ \wedge\ cliente.ciudad_cliente=dirección_gómez.ciudad} \\ \qquad (cliente\ \times\ \rho_{dirección_gómez(calle,ciudad)} \\ \qquad\quad (\Pi_{calle_cliente,\ ciudad_cliente}(\sigma_{nombre_cliente\ =\ \text{"Gómez"}}(cliente)))))\end{array}$$

El resultado de esta consulta, cuando se aplica a la relación *cliente* de la Figura 2.4, se muestra en la Figura 2.18.

La operación renombramiento no es estrictamente necesaria, dado que es posible usar una notación posicional para los atributos. Se pueden nombrar los atributos de una relación de manera implícita donde $1, $2, ... hagan referencia respectivamente al primer atributo, al segundo, etc. La notación posicional también se aplica a los resultados de las operaciones del álgebra relacional. La siguiente expresión del álgebra relacional ilustra el empleo de esta notación con el operador unario σ:

$$\sigma_{\$2=\$3}(R \times R)$$

Si una operación binaria necesita distinguir entre las dos relaciones que son sus operandos, se puede usar una notación posicional parecida para los nombres de las relaciones. Por ejemplo, $\$R1$ puede hacer referencia al primer operando y $\$R2$, al segundo. Sin embargo, la notación posicional no resulta conveniente para las personas, dado que la posición del atributo es un número en vez de un nombre de atributo fácil de recordar. Por tanto, en este libro no se usa la notación posicional.

2.2.8 Definición formal del álgebra relacional

Las operaciones del Apartado 2.2 permiten dar una definición completa de las expresiones del álgebra relacional. Las expresiones fundamentales del álgebra relacional se componen de alguno de las siguientes elementos:

- Una relación de la base de datos
- Una relación constante

Las relaciones constantes se escriben poniendo una relación de sus tuplas entre llaves ({ }), por ejemplo {(C-101,Centro,500) (C-215, Becerril, 700)}.

nombre_cliente
Gómez
Pérez

Figura 2.18 Clientes que viven en la misma ciudad y en la misma calle que Gómez.

nombre_cliente
Gómez
López
Santos

Figura 2.19 Clientes con una cuenta abierta y un préstamo en el banco.

Las expresiones generales del álgebra relacional se construyen a partir de subexpresiones más pequeñas. Sean E_1 y E_2 expresiones de álgebra relacional. Todas las expresiones siguientes son también expresiones del álgebra relacional:

- $E_1 \cup E_2$
- $E_1 - E_2$
- $E_1 \times E_2$
- $\sigma_P(E_1)$, donde P es un predicado de atributos de E_1
- $\Pi_S(E_1)$, donde S es una lista que se compone de algunos de los atributos de E_1
- $\rho_x(E_1)$, donde x es el nuevo nombre del resultado de E_1

2.3 Otras operaciones del álgebra relacional

Las operaciones fundamentales del álgebra relacional son suficientes para expresar cualquier consulta del álgebra relacional[1]. Sin embargo, limitándose exclusivamente a las operaciones fundamentales, algunas consultas habituales resultan complicadas de expresar. Por tanto, se definen otras operaciones que no añaden potencia al álgebra, pero que simplifican las consultas habituales. Para cada operación nueva se facilita una expresión equivalente usando sólo las operaciones fundamentales.

2.3.1 Operación intersección de conjuntos

La primera operación adicional del álgebra relacional que se va a definir es la **intersección de conjuntos**($\cap$). Supóngase que se desea conocer todos los clientes con un préstamo concedido y una cuenta abierta. Empleando la intersección de conjuntos se puede escribir

$$\Pi_{nombre_cliente}(prestatario) \cap \Pi_{nombre_cliente}(impositor)$$

La relación resultante de esta consulta aparece en la Figura 2.19.

Obsérvese que se puede volver a escribir cualquier expresión del álgebra relacional que utilice la intersección de conjuntos sustituyendo la operación intersección por un par de operaciones de diferencia de conjuntos, de la manera siguiente:

$$r \cap s = r - (r - s)$$

Por tanto, la intersección de conjuntos no es una operación fundamental y no añade potencia al álgebra relacional. Sencillamente, es más conveniente escribir $r \cap s$ que $r - (r - s)$.

2.3.2 Operación reunión natural

Suele resultar deseable simplificar ciertas consultas que exijan un producto cartesiano. Generalmente, las consultas que implican un producto cartesiano incluyen un operador selección sobre el resultado del producto cartesiano. Considérese la consulta "Hallar los nombres de todos los clientes que tienen concedido un préstamo en el banco y averiguar su número e importe". En primer lugar se calcula el

1. En el Apartado 2.4 se introducen operaciones que extienden la potencia del álgebra relacional al tratamiento de los valores nulos y de los valores de agregación.

nombre_cliente	*número_préstamo*	*importe*
Fernández	P-16	1.300
Gómez	P-23	2.000
Gómez	P-11	900
López	P-15	1.500
Pérez	P-93	500
Santos	P-17	1.000
Sotoca	P-14	1.500
Valdivieso	P-17	1.000

Figura 2.20 Resultado de $\Pi_{nombre_cliente, número_préstamo, importe}\ (prestatario \bowtie préstamo)$.

producto cartesiano de las relaciones *prestatario* y *préstamo*. Después se seleccionan las tuplas que sólo atañen al mismo *número_préstamo*, seguidas por la proyección de *nombre_cliente*, *número_préstamo* e *importe* resultantes:

$$\Pi_{nombre_cliente, préstamo.número_préstamo, importe} \left(\sigma_{prestatario.número_préstamo = préstamo.número_préstamo}\ (prestatario\ \times\ préstamo)\right)$$

La *reunión natural* es una operación binaria que permite combinar ciertas selecciones y un producto cartesiano en una sola operación. Se denota por el símbolo de **reunión** $\bowtie$. La operación reunión natural forma un producto cartesiano de sus dos argumentos, realiza una selección forzando la igualdad de los atributos que aparecen en ambos esquemas de relación y, finalmente, elimina los atributos duplicados.

Aunque la definición de la reunión natural es compleja, la operación es sencilla de aplicar. A modo de ilustración, considérese nuevamente el ejemplo "Determinar el nombre de todos los clientes que tienen concedido un préstamo en el banco y averiguar su importe". Esta consulta puede expresarse usando la reunión natural de la manera siguiente:

$$\Pi_{nombre_cliente, número_préstamo, importe}\ (prestatario \bowtie préstamo)$$

Dado que los esquemas de *prestatario* y de *préstamo* (es decir, *Esquema_prestatario* y *Esquema_préstamo*) tienen en común el atributo *número_préstamo*, la operación reunión natural sólo considera los pares de tuplas que tienen el mismo valor de *número_préstamo*. Esta operación combina cada uno de estos pares en una sola tupla en la unión de los dos esquemas (es decir, *nombre_cliente, nombre_sucursal, número _préstamo, importe*). Después de realizar la proyección, se obtiene la relación mostrada en la Figura 2.20.

Considérense dos esquemas de relación R y S—que son, por supuesto, listas de nombres de atributos. Si se consideran los esquemas como *conjuntos*, en vez de como listas, se pueden denotar los nombres de los atributos que aparecen tanto en R como en S mediante $R \cap S$, y los nombres de los atributos que aparecen en R, en S o en ambos mediante $R \cup S$. De manera parecida, los nombres de los atributos que aparecen en R pero no en S se denotan por $R - S$, mientras que $S - R$ denota los nombres de los atributos que aparecen en S pero no en R. Obsérvese que las operaciones unión, intersección y diferencia aquí operan sobre conjuntos de atributos, y no sobre relaciones.

Ahora es posible dar una definición formal de la reunión natural. Considérense dos relaciones $r(R)$ y $s(S)$. La **reunión natural** de r y de s, denotada mediante $r \bowtie s$ es una relación del esquema $R \cup S$ definida formalmente de la manera siguiente:

$$r \bowtie s\ =\ \Pi_{R \cup S}\ (\sigma_{r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \ldots \wedge r.A_n = s.A_n}\ (r\ \times\ s))$$

donde $R \cap S = \{A_1,\ A_2, \ldots, A_n\}$.

Como la reunión natural es fundamental para gran parte de la teoría y de la práctica de las bases de datos relacionales, se ofrecen varios ejemplos de su uso.

- Hallar el nombre de todas las sucursales con clientes que tienen una cuenta abierta en el banco y viven en Peguerinos.

nombre_sucursal
Galapagar
Navacerrada

Figura 2.21 Resultado de $\Pi_{nombre_sucursal}(\sigma_{ciudad_cliente = \text{“Peguerinos”}} (cliente \bowtie cuenta \bowtie impositor))$.

$$\Pi_{nombre_sucursal} (\sigma_{ciudad_cliente = \text{“Peguerinos”}} (cliente \bowtie cuenta \bowtie impositor))$$

La relación resultante de esta consulta aparece en la Figura 2.21.

Obsérvese que se ha escrito *cliente* ⋈ *cuenta* ⋈ *impositor* sin añadir paréntesis para especificar el orden en que se deben ejecutar las operaciones reunión natural sobre las tres relaciones. En el caso anterior hay dos posibilidades:

$$(cliente \bowtie cuenta) \bowtie impositor$$
$$cliente \bowtie (cuenta \bowtie impositor)$$

No se ha especificado la expresión deseada, ya que las dos son equivalentes. Es decir, la reunión natural es **asociativa**.

- Hallar todos los clientes que tienen una cuenta abierta *y* un préstamo concedido en el banco.

$$\Pi_{nombre_cliente} (prestatario \bowtie impositor)$$

Obsérvese que en el Apartado 2.3.1 se escribió una expresión para esta consulta usando la intersección de conjuntos. A continuación se repite esa expresión.

$$\Pi_{nombre_cliente} (prestatario) \cap \Pi_{nombre_cliente} (impositor)$$

La relación resultante de esta consulta se mostró anteriormente en la Figura 2.19. Este ejemplo ilustra una característica del álgebra relacional: se pueden escribir varias expresiones del álgebra relacional equivalentes que sean bastante diferentes entre sí.

- Sean $r(R)$ y $s(S)$ relaciones sin atributos en común; es decir, $R \cap S = \emptyset$. ($\emptyset$ denota el conjunto vacío). Por tanto, $r \bowtie s = r \times s$.

La operación *reunión zeta* es una extensión de la operación reunión natural que permite combinar una selección y un producto cartesiano en una sola operación. Considérense las relaciones $r(R)$ y $s(S)$, y sea θ un predicado de los atributos del esquema $R \cup S$. La operación **reunión zeta** $r \bowtie_\theta s$ se define de la manera siguiente:

$$r \bowtie_\theta s = \sigma_\theta(r \times s)$$

2.3.3 Operación división

La operación **división**, denotada por ÷, resulta adecuada para las consultas que incluyen la expresión “para todos”. Supóngase que se desea hallar a todos los clientes que tengan abierta una cuenta en *todas* las sucursales ubicadas en Arganzuela. Se pueden obtener todas las sucursales de Arganzuela mediante la expresión

$$r_1 = \Pi_{nombre_sucursal} (\sigma_{ciudad_sucursal = \text{“Arganzuela”}} (sucursal))$$

La relación resultante de esta expresión aparece en la Figura 2.22.

nombre_sucursal
Centro
Galapagar

Figura 2.22 Resultado de $\Pi_{nombre_sucursal}(\sigma_{ciudad_sucursal = \text{“Arganzuela”}} (sucursal))$.

Se pueden encontrar todos los pares (*nombre_cliente, nombre_sucursal*) para los que el cliente tiene una cuenta en una sucursal escribiendo

$$r_2 = \Pi_{nombre_cliente, nombre_sucursal} (impositor \bowtie cuenta)$$

La Figura 2.23 muestra la relación resultante de esta expresión.

Ahora hay que hallar los clientes que aparecen en r_2 con los nombres de *todas* las sucursales de r_1. La operación que proporciona exactamente esos clientes es la operación división. La consulta se formula escribiendo

$$\Pi_{nombre_cliente, nombre_sucursal} (impositor \bowtie cuenta)$$
$$\div \Pi_{nombre_sucursal} (\sigma_{ciudad_sucursal = \text{“Arganzuela”}} (sucursal))$$

El resultado de esta expresión es una relación que tiene el esquema (*nombre_cliente*) y que contiene la tupla (González).

Formalmente, sean $r(R)$ y $s(S)$ relaciones y $S \subseteq R$; es decir, todos los atributos del esquema S están también en el esquema R. La relación $r \div s$ es una relación del esquema $R - S$ (es decir, del esquema que contiene todos los atributos del esquema R que no están en el esquema S). Una tupla *t* está en $r \div s$ si y sólo si se cumplen estas dos condiciones:

1. *t* está en $\Pi_{R-S}(r)$
2. Para cada tupla t_s de *s* hay una tupla t_r de *r* que cumple las dos condiciones siguientes:
   a. $t_r[S] = t_s[S]$
   b. $t_r[R - S] = t$

Puede resultar sorprendente descubrir que, dados una operación división y los esquemas de las relaciones, se pueda definir la operación división en términos de las operaciones fundamentales. Sean $r(R)$ y $s(S)$ dadas, con $S \subseteq R$:

$$r \div s = \Pi_{R-S} (r) - \Pi_{R-S} ((\Pi_{R-S} (r) \times s) - \Pi_{R-S,S}(r))$$

Para comprobar que esta expresión es cierta, obsérvese que $\Pi_{R-S} (r)$ proporciona todas las tuplas *t* que cumplen la primera condición de la definición de la división. La expresión del lado derecho del operador diferencia de conjuntos,

$$\Pi_{R-S} ((\Pi_{R-S} (r) \times s) - \Pi_{R-S,S}(r))$$

elimina las tuplas que no cumplen la segunda condición de la definición de la división. Esto se logra de la manera siguiente. Considérese $\Pi_{R-S} (r) \times s$. Esta relación está en el esquema *R* y empareja cada tupla de $\Pi_{R-S} (r)$ con cada tupla de *s*. La expresión $\Pi_{R-S,S}(r)$ sólo reordena los atributos de r.

Por tanto, $(\Pi_{R-S} (r) \times s) - \Pi_{R-S,S}(r)$ genera los pares de tuplas de $\Pi_{R-S} (r)$ y de *s* que no aparecen en *r*. Si una tupla t_j está en

$$\Pi_{R-S} ((\Pi_{R-S} (r) \times s) - \Pi_{R-S,S}(r))$$

nombre_cliente	*nombre_sucursal*
Abril	Collado Mediano
Gómez	Becerril
González	Centro
González	Galapagar
López	Navacerrada
Rupérez	Moralzarzal
Santos	Galapagar

Figura 2.23 Resultado de $\Pi_{nombre_cliente, nombre_sucursal} (impositor \bowtie cuenta)$.

nombre_cliente	*límite*	*saldo_crédito*
Gómez	2.000	400
López	1.500	1.500
Pérez	2.000	1.750
Santos	6.000	700

Figura 2.24 La relación *información_crédito*.

existe alguna tupla t_s de s que no se combina con la tupla t_j para formar una tupla de r. Por tanto, t_j guarda un valor de los atributos $R - S$ que no aparece en $r \div s$. Estos valores son los que se eliminan de $\Pi_{R-S}(r)$.

2.3.4 Operación asignación

En ocasiones resulta conveniente escribir una expresión del álgebra relacional mediante la asignación de partes de esa expresión a variables de relación temporal. La operación **asignación**, denotada por $\leftarrow$, actúa de manera parecida a la asignación de los lenguajes de programación. Para ilustrar esta operación, considérese la definición de la división dada en el Apartado 2.3.3. Se puede escribir $r \div s$ como

$$
\begin{aligned}
temp1 &\leftarrow \Pi_{R-S}(r) \\
temp2 &\leftarrow \Pi_{R-S}((temp1 \times s) - \Pi_{R-S,S}(r)) \\
resultado &= temp1 - temp2
\end{aligned}
$$

La evaluación de una asignación no hace que se muestre ninguna relación al usuario. Por el contrario, el resultado de la expresión situada a la derecha de $\leftarrow$ se asigna a la variable relación situada a la izquierda de $\leftarrow$. Esta variable relación puede usarse en expresiones posteriores.

Con la operación asignación se pueden escribir las consultas como programas secuenciales que constan de en una serie de asignaciones seguida de una expresión cuyo valor se muestra como resultado de la consulta. En las consultas del álgebra relacional la asignación siempre debe hacerse a una variable de relación temporal. Las asignaciones a relaciones permanentes constituyen una modificación de la base de datos. Este asunto se estudiará en el Apartado 2.6. Obsérvese que la operación asignación no añade potencia alguna al álgebra. Resulta, sin embargo, una manera conveniente de expresar las consultas complejas.

2.4 Operaciones del álgebra relacional extendida

Las operaciones básicas del álgebra relacional se han extendido de varias formas. Una sencilla es permitir operaciones aritméticas como parte de las proyecciones. Una importante es permitir *operaciones de agregación*, como el cálculo de la suma de los elementos de un conjunto, o su media. Otra extensión importante es la operación *reunión externa*, que permite a las expresiones del álgebra relacional trabajar con los valores nulos, que modelan la información ausente.

2.4.1 Proyección generalizada

La operación **proyección generalizada** extiende la proyección permitiendo que se utilicen funciones aritméticas en la lista de proyección. La operación proyección generalizada es de la forma

$$\Pi_{F_1, F_2, \ldots, F_n}(E)$$

donde E es cualquier expresión del álgebra relacional y $F_1, F_2, \ldots, F_n$ son expresiones aritméticas que incluyen constantes y atributos del esquema de E. Como caso especial, la expresión aritmética puede ser simplemente un atributo o una constante.

Por ejemplo, supóngase que se dispone de una relación *información_crédito*, como se muestra en la Figura 2.24, que proporciona el límite de crédito y el importe consumido actualmente (*saldo_crédito*). Si se desea determinar el importe disponible por cada persona, se puede escribir la expresión siguiente:

nombre_cliente	*crédito_disponible*
Gómez	1.600
López	0
Pérez	250
Santos	5.300

Figura 2.25 Resultado de $\Pi_{nombre_cliente, (límite\ -\ saldo_crédito)\ \textbf{as}\ crédito_disponible}\ (información_crédito)$.

$$\Pi_{nombre_cliente, límite\ -\ saldo_crédito}\ (\textit{información_crédito})$$

El atributo resultante de la expresión $límite\ -\ saldo_crédito$ no tiene nombre asignado. Se puede aplicar la operación renombramiento al resultado de la proyección generalizada para darle nombre. Como conveniencia notacional, el renombramiento de atributos se puede combinar con la proyección generalizada como se ilustra a continuación:

$$\Pi_{nombre_cliente, (límite\ -\ saldo_crédito)\ \textbf{as}\ crédito_disponible}\ (\textit{información_crédito})$$

Al segundo atributo de esta proyección generalizada se le ha asignado el nombre *crédito_disponible*. La Figura 2.25 muestra el resultado de aplicar esta expresión a la relación de la Figura 2.24.

2.4.2 Funciones de agregación

Las **funciones de agregación** toman un conjunto de valores y devuelven como resultado un único valor. Por ejemplo, la función de agregación **sum** toma un conjunto de valores y devuelve su suma. Por tanto, la función **sum** aplicada al conjunto

$$\{1, 1, 3, 4, 4, 11\}$$

devuelve el valor 24. La función de agregación **avg** devuelve la media de los valores. Cuando se aplica al conjunto anterior devuelve el valor 4. La función de agregación **count** devuelve el número de elementos del conjunto, y devolvería 6 en el caso anterior. Otras funciones de agregación habituales son **min** y **max**, que devuelven los valores mínimo y máximo del conjunto; en el ejemplo anterior devuelven 1 y 11, respectivamente.

Los conjuntos sobre los que operan las funciones de agregación pueden contener valores repetidos; el orden en el que aparezcan los valores no tiene importancia. Estos conjuntos se denominan **multiconjuntos**. Los conjuntos son un caso especial de los multiconjuntos, en los que sólo hay una copia de cada elemento.

Para ilustrar el concepto de agregación se usará la relación *trabajo_por_horas* descrita en la Figura 2.26, que muestra los empleados a tiempo parcial. Supóngase que se desea determinar la suma total de los sueldos de los empleados del banco a tiempo parcial. La expresión del álgebra relacional para esta consulta es la siguiente:

$$\mathcal{G}_{\textbf{sum}(sueldo)}(trabajo_por_horas)$$

nombre_empleado	*nombre_sucursal*	*sueldo*
Cana	Leganés	1.500
Cascallar	Navacerrada	5.300
Catalán	Leganés	1.600
Díaz	Centro	1.300
Fernández	Navacerrada	1.500
González	Centro	1.500
Jiménez	Centro	2.500
Ribera	Navacerrada	1.300

Figura 2.26 La relación *trabajo_por_horas*.

nombre_empleado	*nombre_sucursal*	*sueldo*
Cana	Leganés	1.500
Catalán	Leganés	1.600
Díaz	Centro	1.300
González	Centro	1.500
Jiménez	Centro	2.500
Cascallar	Navacerrada	5.300
Fernández	Navacerrada	1.500
Ribera	Navacerrada	1.300

Figura 2.27 La relación *trabajo_por_horas* después de la agrupación.

El símbolo $\mathcal{G}$ es la letra G en el tipo de letra caligráfico; se lee "G caligráfica". La operación del álgebra relacional $\mathcal{G}$ significa que se debe aplicar la agregación; y su subíndice, la operación de agregación que se aplica. El resultado de la expresión anterior es una relación con un único atributo, que contiene una sola fila con un valor numérico correspondiente a la suma de los sueldos de todos los trabajadores que trabajan en el banco a tiempo parcial.

Supóngase ahora que se desea determinar la suma total de sueldos de los empleados a tiempo parcial de cada sucursal bancaria, en lugar de determinar la suma total. Para ello hay que dividir la relación *trabajo_por_horas* en **grupos** basados en la sucursal y aplicar la función de agregación a cada grupo. La expresión siguiente obtiene el resultado deseado usando el operador de agregación $\mathcal{G}$:

$$_{nombre_sucursal}\mathcal{G}_{\mathbf{sum}(sueldo)}(trabajo_por_horas)$$

Existen casos en los que es necesario borrar los valores repetidos antes de calcular una función de agregación. Si se desea eliminar los valores repetidos hay que usar los mismos nombres de funciones que antes, con la cadena de texto "**distinct**" precedida de un guión añadida al final del nombre de la función (por ejemplo, **count-distinct**). Un ejemplo se da en la consulta "Determinar el número de sucursales que aparecen en la relación *trabajo_por_horas*". En este caso, el nombre de cada sucursal sólo se cuenta una vez, independientemente del número de empleados que trabajen en la misma. Esta consulta se escribe de la manera siguiente:

$$\mathcal{G}_{\mathbf{count-distinct}(nombre_sucursal)}(trabajo_por_horas)$$

Al aplicar esta consulta a la relación de la Figura 2.26 el resultado está compuesto por una única fila con valor 3. En esta consulta el atributo *nombre_sucursal* en el subíndice a la izquierda de $\mathcal{G}$ indica que la relación de entrada *trabajo_por_horas* debe dividirse en grupos de acuerdo con el valor de *nombre_sucursal*. La Figura 2.27 muestra los grupos resultantes. La expresión $\mathbf{sum}(sueldo)$ en el subíndice a la derecha de $\mathcal{G}$ indica que, para cada grupo de tuplas (es decir, para cada sucursal) hay que aplicar la función de agregación **sum** al conjunto de valores del atributo *sueldo*. La relación resultante consiste en tuplas con el nombre de la sucursal y la suma de los sueldos de la sucursal, como se muestra en la Figura 2.28.

La forma general de la **operación de agregación** $\mathcal{G}$ es la siguiente:

$$_{G_1,G_2,\ldots,G_n}\mathcal{G}_{F_1(A_1),F_2(A_2),\ldots,F_m(A_m)}(E)$$

nombre_sucursal	*sum(sueldo)*
Centro	5.300
Leganés	3.100
Navacerrada	8.100

Figura 2.28 Resultado de $_{nombre_sucursal}\mathcal{G}_{\mathbf{sum}(sueldo)}(trabajo_por_horas)$.

nombre_sucursal	*suma_sueldo*	*sueldo_máximo*
Centro	5.300	2.500
Leganés	3.100	1.600
Navacerrada	8.100	5.300

Figura 2.29 Resultado de $_{nombre_sucursal}\mathcal{G}_{\mathbf{sum}(sueldo)\ \mathbf{as}\ suma_sueldo,\mathbf{max}(sueldo)\ \mathbf{as}\ sueldo_máximo}(trabajo_por_horas)$.

donde E es cualquier expresión del álgebra relacional; $G_1, G_2, \ldots, G_n$ constituye una lista de atributos sobre los que se realiza la agrupación, cada F_i es una función de agregación y cada A_i es el nombre de un atributo. El significado de la operación es el siguiente: las tuplas del resultado de la expresión E se dividen en grupos de manera que

1. Todas las tuplas de cada grupo tienen los mismos valores para $G_1, G_2, \ldots, G_n$.
2. Las tuplas de grupos diferentes tienen valores diferentes para $G_1, G_2, \ldots, G_n$.

Por tanto, los grupos pueden identificarse por el valor de los atributos $G_1, G_2, \ldots, G_n$. Para cada grupo $(g_1, g_2, \ldots, g_n)$ el resultado tiene una tupla $(g_1, g_2, \ldots, g_n, a_1, a_2, \ldots, a_m)$ donde, para cada i, a_i es el resultado de aplicar la función de agregación F_i al multiconjunto de valores del atributo A_i en el grupo.

Como caso especial de la operación de agregación, la lista de atributos $G_1, G_2, \ldots, G_n$ puede estar vacía, en cuyo caso sólo hay un grupo que contiene todas las tuplas de la relación. Esto se corresponde con la agregación sin agrupación.

Volviendo al ejemplo anterior, si se deseara determinar el sueldo máximo de los empleados a tiempo parcial de cada sucursal, además de la suma de los sueldos, habría que escribir la expresión

$$_{nombre_sucursal}\mathcal{G}_{\mathbf{sum}(sueldo),\mathbf{max}(sueldo)}(trabajo_por_horas)$$

Como en la proyección generalizada, el resultado de las operaciones de agregación no tiene nombre. Se puede aplicar la operación renombramiento al resultado para darle un nombre. Como conveniencia notacional, los atributos de las operaciones de agregación se pueden renombrar como se indica a continuación:

$$_{nombre_sucursal}\mathcal{G}_{\mathbf{sum}(sueldo)\ \mathbf{as}\ suma_sueldo,\mathbf{max}(sueldo)\ \mathbf{as}\ sueldo_máximo}(trabajo_por_horas)$$

La Figura 2.29 muestra el resultado de la expresión.

2.4.3 Reunión externa

La operación **reunión externa** es una ampliación de la operación reunión para trabajar con información ausente. Supóngase que se dispone de relaciones con los siguientes esquemas y que contienen datos de empleados a tiempo completo:

empleado (*nombre_empleado, calle, ciudad*)
trabajo_a_tiempo_completo (*nombre_empleado, nombre_sucursal, sueldo*)

Considérense las relaciones *empleado* y *trabajo_a_tiempo_completo* mostradas en la Figura 2.30. Supóngase que se desea generar una única relación con toda la información (calle, ciudad, nombre de la sucursal y sueldo) de los empleados a tiempo completo. Un posible enfoque sería usar la operación reunión natural de la manera siguiente:

$$empleado \bowtie trabajo_a_tiempo_completo$$

El resultado de esta expresión se muestra en la Figura 2.31. Obsérvese que se ha perdido la información sobre la calle y la ciudad de residencia de Gómez, dado que la tupla que describe a Gómez no está presente en la relación *trabajo_a_tiempo_completo*; de manera parecida, se ha perdido la información sobre el nombre de la sucursal y sobre el sueldo de Barea, dado que la tupla que describe a Barea no está presente en la relación *empleado*.

nombre_empleado	*calle*	*ciudad*
Segura	Tebeo	La Loma
Domínguez	Viaducto	Villaconejos
Gómez	Bailén	Alcorcón
Valdivieso	Fuencarral	Móstoles

nombre_empleado	*nombre_sucursal*	*sueldo*
Segura	Majadahonda	1.500
Domínguez	Majadahonda	1.300
Barea	Fuenlabrada	5.300
Valdivieso	Fuenlabrada	1.500

Figura 2.30 Las relaciones *empleado* y *trabajo_a_tiempo_completo*.

nombre_empleado	*calle*	*ciudad*	*nombre_sucursal*	*sueldo*
Domínguez	Viaducto	Villaconejos	Majadahonda	1.300
Segura	Tebeo	La Loma	Majadahonda	1.500
Valdivieso	Fuencarral	Móstoles	Fuenlabrada	1.500

Figura 2.31 El resultado de $empleado \bowtie trabajo_a_tiempo_completo$.

Se puede usar la operación reunión externa para evitar esta pérdida de información. En realidad, esta operación tiene tres formas diferentes: *reunión externa por la izquierda*, denotada por $\supset\!\bowtie$; *reunión externa por la derecha*, denotada por $\bowtie\!\subset$ y *reunión externa completa*, denotada por $\supset\!\bowtie\!\subset$. Las tres formas de la reunión externa calculan la reunión y añaden tuplas adicionales al resultado de la misma. El resultado de las expresiones $empleado \supset\!\bowtie trabajo_a_tiempo_completo$, $empleado \bowtie\!\subset trabajo_a_tiempo_completo$ y $empleado \supset\!\bowtie\!\subset trabajo_a_tiempo_completo$ se muestra en las Figuras 2.32, 2.33 y 2.34, respectivamente.

La **reunión externa por la izquierda** ($\supset\!\bowtie$) toma todas las tuplas de la relación de la izquierda que no coinciden con ninguna tupla de la relación de la derecha, las rellena con valores nulos en todos los demás atributos de la relación de la derecha y las añade al resultado de la reunión natural. En la Figura 2.32 la tupla (Gómez, Bailén, Alcorcón, *nulo*, *nulo*) es una tupla de este tipo. Toda la información de la relación de la izquierda se halla presente en el resultado de la reunión externa por la izquierda.

La **reunión externa por la derecha** ($\bowtie\!\subset$) es simétrica de la reunión externa por la izquierda. Rellena con valores nulos las tuplas de la relación de la derecha que no coinciden con ninguna tupla de la relación de la izquierda y las añade al resultado de la reunión natural. En la Figura 2.33 la tupla (Barea, *nulo*,

nombre_empleado	*calle*	*ciudad*	*nombre_sucursal*	*sueldo*
Domínguez	Viaducto	Villaconejos	Majadahonda	1.300
Gómez	Bailén	Alcorcón	*nulo*	*nulo*
Segura	Tebeo	La Loma	Majadahonda	1.500
Valdivieso	Fuencarral	Móstoles	Fuenlabrada	1.500

Figura 2.32 Resultado de $empleado \supset\!\bowtie trabajo_a_tiempo_completo$.

nombre_empleado	*calle*	*ciudad*	*nombre_sucursal*	*sueldo*
Barea	*nulo*	*nulo*	Fuenlabrada	5.300
Domínguez	Viaducto	Villaconejos	Majadahonda	1.300
Segura	Tebeo	La Loma	Majadahonda	1.500
Valdivieso	Fuencarral	Móstoles	Fuenlabrada	1.500

Figura 2.33 Resultado de $empleado \bowtie\!\subset trabajo_a_tiempo_completo$.

nombre_empleado	*calle*	*ciudad*	*nombre_sucursal*	*sueldo*
Barea	*nulo*	*nulo*	Fuenlabrada	5.300
Domínguez	Viaducto	Villaconejos	Majadahonda	1.300
Gómez	Bailén	Alcorcón	*nulo*	*nulo*
Segura	Tebeo	La Loma	Majadahonda	1.500
Valdivieso	Fuencarral	Móstoles	Fuenlabrada	1.500

Figura 2.34 Resultado de $empleado \rightleftarrows trabajo_a_tiempo_parcial$.

$nulo$, Fuenlabrada, 5.300) es una tupla de este tipo. Por tanto, toda la información de la relación de la derecha se halla presente en el resultado de la reunión externa por la derecha.

La **reunión externa completa** ($\rightleftarrows$) realiza estas dos operaciones, rellenando las tuplas de la relación de la izquierda que no coinciden con ninguna tupla de la relación de la derecha y las tuplas de la relación de la derecha que no coinciden con ninguna tupla de la relación de la izquierda, y añadiéndolas al resultado de la reunión. La Figura 2.34 muestra el resultado de una reunión externa completa.

Puesto que las operaciones de reunión pueden generar resultados que contengan valores nulos, es necesario especificar la manera en que deben manejar estos valores las diferentes operaciones del álgebra relacional. El Apartado 2.5 aborda este problema.

Es interesante observar que las operaciones de reunión externa pueden expresarse mediante las operaciones básicas del álgebra relacional. Por ejemplo, la operación de reunión externa por la izquierda $r \rightleftarrows s$ se puede expresar como:

$$(r \bowtie s) \cup (r - \Pi_R(r \bowtie s)) \times \{(nulo, \ldots, nulo)\}$$

donde la relación constante $\{(nulo, \ldots, nulo)\}$ se encuentra en el esquema $S - R$.

2.5 Valores nulos

En este apartado se define la forma en que las diferentes operaciones del álgebra relacional tratan los valores nulos y las complicaciones que surgen cuando los valores nulos participan en las operaciones aritméticas o en las comparaciones. Como se verá, a menudo hay varias formas de tratar los valores nulos y, en consecuencia, las siguientes definiciones pueden ser a veces arbitrarias. Las operaciones y las comparaciones con valores nulos se deben evitar siempre que sea posible.

Dado que el valor especial *nulo* indica "valor desconocido o no existente", cualquier operación aritmética (como $+, -, *$ y $/$) que incluya valores nulos debe devolver un valor nulo.

De manera parecida, cualquier comparación (como $<, <=, >, >=$ y $\neq$) que incluya un valor nulo tiene como resultado el valor especial **desconocido**; no se puede decir si el resultado de la comparación es cierto o falso, así que se dice que el resultado es el nuevo valor lógico *desconocido*.

Las comparaciones que incluyen valores nulos pueden aparecer dentro de expresiones booleanas que incluyan las operaciones y (conjunción), o (disyunción) y no (negación). Por tanto, se debe definir la forma en que estas tres operaciones lógicas tratan el valor lógico *desconocido*.

- **Y**: (*cierto* **y** *desconocido*) = *desconocido*; (*falso* **y** *desconocido*) = *falso*; (*desconocido* **y** *desconocido*) = *desconocido*.
- **O**: (*cierto* **o** *desconocido*) = *cierto*; (*falso* **o** *desconocido*) = *desconocido*; (*desconocido* **o** *desconocido*) = *desconocido*.
- **No**: (**no** *desconocido*) = *desconocido*.

Ahora es posible describir la forma en que las diferentes operaciones del álgebra relacional tratan los valores nulos.

- **Selección**. La operación selección evalúa el predicado P en $\sigma_P(E)$ sobre cada tupla t de E. Si el predicado devuelve el valor *cierto*, se añade t al resultado. En caso contrario, si el predicado devuelve *desconocido* o *falso*, t no se añade al resultado.

- **Reunión**. Las reuniones se pueden expresar como un producto cartesiano seguido de una selección. Por tanto, la definición de la forma en que la selección trata los nulos también define la forma en que lo hacen las operaciones reunión.

  En una reunión natural, por ejemplo, $r \bowtie s$ se puede observar de la definición anterior que si dos tuplas, $t_r \in r$ y $t_s \in s$, tienen un valor nulo en un atributo común, entonces las tuplas no coinciden.

- **Proyección**. La operación proyección trata los valores nulos como cualesquiera otros valores al eliminar valores duplicados. Así, si dos tuplas del resultado de la proyección son exactamente iguales y las dos tienen valores nulos en los mismos campos, se tratan como duplicados.

  Esta decisión es un tanto arbitraria ya que, sin conocer el valor real, no se sabe si los dos valores nulos son duplicados o no.

- **Unión**, **intersección** y **diferencia**. Estas operaciones tratan los valores nulos igual que la operación proyección; tratan las tuplas que tienen los mismos valores en todos los campos como duplicados, aunque algunos de los campos tengan valores nulos en ambas tuplas.

  Este comportamiento es un tanto arbitrario, especialmente en el caso de la intersección y la diferencia, dado que no se sabe si los valores reales (si existen) representados por los nulos son iguales.

- **Proyección generalizada**. Se describió la manera en que se tratan los nulos en las expresiones al comienzo del Apartado 2.5. Las tuplas duplicadas que contienen valores nulos se tratan como en la operación proyección.

- **Funciones de agregación**. Cuando hay valores nulos en los atributos agregados, las operaciones de agregación los tratan igual que en el caso de la proyección: si dos tuplas son iguales en todos los atributos de agregación, la operación las coloca en el mismo grupo, aunque parte de los valores de los atributos sean valores nulos.

  Cuando aparecen valores nulos en los atributos agregados, la operación borra los valores nulos del resultado antes de aplicar la agregación. Si el multiconjunto resultante está vacío, el resultado agregado es nulo.

  Obsérvese que el tratamiento de los valores nulos en este caso es diferente que en las expresiones aritméticas ordinarias; se podría haber definido el resultado de una operación de agregación como nulo si tan sólo uno de los valores agregados fuera nulo. Sin embargo, esto significaría que un único valor desconocido en un gran grupo podría hacer que el resultado agregado sobre el grupo fuese nulo, y se perdería una gran cantidad de información útil.

- **Reunión externa**. Las operaciones de reunión externa se comportan igual que las operaciones de reunión, excepto sobre las tuplas que no aparecen en el resultado de la reunión. Esas tuplas se pueden añadir al resultado (dependiendo de si la operación es $\exists\bowtie, \bowtie\sqsubset$ o $\exists\bowtie\sqsubset$) rellenando con valores nulos.

2.6 Modificación de la base de datos

Hasta ahora se ha centrado la atención en la extracción de información de la base de datos. En este apartado se abordará la manera de añadir, eliminar y modificar información de la base de datos.

Las modificaciones de la base de datos se expresan mediante la operación asignación. Las asignaciones a las relaciones reales de la base de datos se realizan empleando la misma notación que se describió para la asignación en el Apartado 2.3.

2.6.1 Borrado

Las solicitudes de borrado se expresan básicamente igual que las consultas. Sin embargo, en lugar de mostrar las tuplas al usuario, se eliminan de la base de datos las tuplas seleccionadas. Sólo se pueden borrar tuplas enteras; no se pueden borrar valores de atributos concretos. En el álgebra relacional los borrados se expresan mediante

$$r \leftarrow r - E$$

donde *r* es una relación y *E* es una consulta del álgebra relacional.

A continuación se muestran varios ejemplos de borrado:

- Borrar todas las cuentas de Gómez.

$$impositor \leftarrow impositor - \sigma_{nombre_cliente = \text{“Gómez”}}(impositor)$$

- Borrar todos los préstamos con importes entre 0 y 50.

$$préstamo \leftarrow préstamo - \sigma_{importe \geq 0 \wedge importe \leq 50}(préstamo)$$

- Borrar todas las cuentas de las sucursales de Arganzuela

$$\begin{aligned} r_1 &\leftarrow \sigma_{ciudad_sucursal = \text{“Arganzuela”}}(cuenta \bowtie sucursal) \\ r_2 &\leftarrow \Pi_{nombre_sucursal, número_cuenta, saldo}(r_1) \\ cuenta &\leftarrow cuenta - r_2 \end{aligned}$$

  Obsérvese que en el último ejemplo se ha simplificado la expresión mediante la asignación a relaciones temporales (r_1 y r_2).

2.6.2 Inserción

Para insertar datos en una relación hay que especificar la tupla que se va a insertar o escribir una consulta cuyo resultado sea el conjunto de tuplas que se van a insertar. Evidentemente, el valor de los atributos de las tuplas insertadas debe ser miembro del dominio de cada atributo. De manera parecida, las tuplas insertadas deben tener la aridad correcta. El álgebra relacional expresa las inserciones mediante

$$r \leftarrow r \cup E$$

donde *r* es una relación y *E* es una expresión del álgebra relacional. La inserción de una sola tupla se expresa haciendo que *E* sea una relación constante que contiene una tupla.

Supóngase que se desea insertar el hecho de que Gómez tiene 1.200 € en la cuenta C-973 de la sucursal de Navacerrada. Hay que escribir

$$\begin{aligned} &cuenta \leftarrow cuenta \cup \{(\text{C-973, “Navacerrada”, 1200})\} \\ &impositor \leftarrow impositor \cup \{(\text{“Gómez”, C-973})\} \end{aligned}$$

De forma más general, puede que se desee insertar tuplas según el resultado de una consulta. Supóngase que se desea ofrecer una nueva cuenta de ahorro con 200 € como regalo a todos los clientes con préstamos concedidos en la sucursal de Navacerrada. Se usará el número de préstamo como número de esta cuenta de ahorro. Se escribe:

$$\begin{aligned} r_1 &\leftarrow (\sigma_{nombre_sucursal = \text{“Navacerrada”}}(prestatario \bowtie préstamo)) \\ r_2 &\leftarrow \Pi_{número_préstamo, nombre_sucursal}(r_1) \\ cuenta &\leftarrow cuenta \cup (r_2 \times \{(200)\}) \\ impositor &\leftarrow impositor \cup \Pi_{nombre_cliente, número_préstamo}(r_1) \end{aligned}$$

En lugar de especificar una tupla como se hizo anteriormente, se especifica un conjunto de tuplas que se inserta en las relaciones *cuenta* e *impositor*. Cada tupla de la relación *cuenta* tiene un *número_cuenta* (que es igual que el número de préstamo), un *nombre_sucursal* (Navacerrada) y el saldo inicial de la nueva cuenta (200 €). Cada tupla de la relación *impositor* tiene como *nombre_cliente* el nombre del prestatario al que se le regala la nueva cuenta y el mismo número de cuenta que la correspondiente tupla de *cuenta*.

2.6.3 Actualización

Puede que, en algunas situaciones, se desee modificar un valor de una tupla sin modificar *todos* los valores de esa tupla. Se puede usar el operador proyección generalizada para llevar a cabo esta tarea:

$$r \leftarrow \Pi_{F_1, F_2, \ldots, F_n}(r)$$

donde cada F_i es, o bien el i-ésimo atributo de r en el caso de que este atributo no se vaya a actualizar o, en caso contrario, una expresión sólo con constantes y atributos de r que proporciona el nuevo valor del atributo. Téngase en cuenta que el esquema de la expresión resultante de la expresión de proyección generalizada debe coincidir con el esquema original de r.

Si se desea seleccionar varias tuplas de r y actualizar sólo esas tuplas, se puede usar la expresión siguiente, donde P denota la condición de selección que escoge las tuplas que hay que actualizar:

$$r \leftarrow \Pi_{F_1, F_2, \ldots, F_n}(\sigma_P(r)) \cup (r - \sigma_P(r))$$

Para ilustrar el uso de la operación actualización supóngase que se va a realizar el pago de los intereses y que hay que aumentar todos los saldos en un cinco por ciento:

$$cuenta \leftarrow \Pi_{número_cuenta, nombre_sucursal,\ saldo\ *\ 1.05}\ (cuenta)$$

Supóngase ahora que las cuentas con saldos superiores a 10.000 € reciben un interés del seis por ciento, mientras que el resto recibe un cinco por ciento:

$$cuenta \leftarrow \Pi_{número_cuenta, nombre_sucursal,\ saldo\ *\ 1.06}\ (\sigma_{saldo > 10000}\ (cuenta)) \cup \Pi_{número_cuenta, nombre_sucursal,\ saldo\ *\ 1.05}\ (\sigma_{saldo \leq 10000}\ (cuenta))$$

2.7 Resumen

- El **modelo de datos relacional** se basa en un conjunto de tablas. El usuario del sistema de bases de datos puede consultar esas tablas, insertar tuplas nuevas, borrar tuplas y actualizar (modificar) las tuplas. Hay varios lenguajes para expresar estas operaciones.
- El **álgebra relacional** define un conjunto de operaciones algebraicas que operan sobre las tablas y devuelven tablas como resultado. Estas operaciones se pueden combinar para obtener expresiones que definan las consultas deseadas. El álgebra define las operaciones básicas usadas en los lenguajes de consultas relacionales.
- Las operaciones del álgebra relacional se pueden dividir en:
  - Operaciones básicas.
  - Operaciones adicionales que se pueden expresar en términos de las operaciones básicas.
  - Operaciones extendidas, algunas de las cuales añaden mayor poder expresivo al álgebra relacional.
- Las bases de datos se pueden modificar con la **inserción**, el **borrado** y la **actualización** de tuplas. Se ha usado el álgebra relacional con el **operador asignación** para expresar estas modificaciones.
- El álgebra relacional es un lenguaje rígido y formal que no resulta adecuado para los usuarios ocasionales de los sistemas de bases de datos. Los sistemas comerciales de bases de datos, por tanto, usan lenguajes con más "azúcar sintáctico". En los Capítulos 3 y 4 se tomará en consideración el lenguaje más influyente, **SQL**, que está basado en el álgebra relacional.

Términos de repaso

- Tabla.
- Relación.
- Variable tupla.
- Dominio atómico.
- Valor nulo.
- Esquema de la base de datos.
- Ejemplar de la base de datos.
- Esquema de la relación.
- Ejemplar de la relación.
- Claves.
- Clave externa.
  - Relación referenciante.
  - Relación referenciada.
- Diagrama de esquema.
- Lenguaje de consultas.
- Lenguaje procedimental.

- Lenguaje no procedimental.
- Álgebra relacional.
- Operaciones del álgebra relacional:
  - ☐ Selección (σ).
  - ☐ Proyección (Π).
  - ☐ Unión ($\cup$).
  - ☐ Diferencia de conjuntos ($-$).
  - ☐ Producto cartesiano ($\times$).
  - ☐ Renombramiento (ρ).
- Operaciones adicionales:
  - ☐ Intersección de conjuntos ($\cap$).
  - ☐ Reunión natural ($\bowtie$).
  - ☐ División ($/$).
- Operación asignación.
- Operaciones del álgebra relacional extendida:
  - ☐ Proyección generalizada (Π).
  - ☐ Reunión externa.
    - — Reunión externa por la izquierda ($\mathbin{⟕}$).
    - — Reunión externa por la derecha ($\mathbin{⟖}$).
    - — Reunión externa completa ($\mathbin{⟗}$).
  - ☐ Agregación ($\mathcal{G}$).
- Multiconjuntos.
- Agrupación.
- Valores nulos.
- Valores lógicos:
  - ☐ *cierto*.
  - ☐ *falso*.
  - ☐ *desconocido*.
- Modificación de la base de datos.
  - ☐ Borrado.
  - ☐ Inserción.
  - ☐ Actualización.

Ejercicios prácticos

2.1 Considérese la base de datos relacional de la Figura 2.35, en la que las claves primarias están subrayadas. Obténgase una expresión del álgebra relacional para cada una de las consultas siguientes:

a. Determinar el nombre de todos los empleados que viven en la misma ciudad y en la misma calle que sus jefes.

b. Determinar el nombre de todos los empleados de esta base de datos que no trabajan para el Banco Importante.

c. Determinar el nombre de todos los empleados que ganan más que cualquier empleado del Banco Pequeño.

2.2 Las operaciones de reunión externa amplían la operación reunión natural de manera que las tuplas de las relaciones participantes no se pierdan en el resultado de la reunión. Descríbase la manera en que la operación reunión zeta puede ampliarse para que las tuplas de la relación de la izquierda, las de la relación de la derecha o las de ambas relaciones no se pierdan en el resultado de las reuniones zeta.

2.3 Considérese la base de datos relacional de la Figura 2.35. Obténgase una expresión del álgebra relacional para cada una de las peticiones siguientes:

a. Modificar la base de datos de manera que Santos viva ahora en Tres Cantos.

b. Dar a todos los jefes de la base de datos un aumento de sueldo del 10 por ciento.

Ejercicios

2.4 Descríbanse las diferencias de significado entre los términos *relación* y *esquema de la relación*.

empleado (*nombre_persona*, *calle*, *ciudad*)
trabaja (*nombre_persona*, *nombre_empresa*, *sueldo*)
empresa (*nombre_empresa*, *ciudad*)
jefe (*nombre_persona*, *nombre_jefe*)

Figura 2.35 Base de datos relacional para los ejercicios 2.1, 2.3, 2.5, 2.7 y 2.9.

2.5 Considérese la base de datos relacional de la Figura 2.35, en la que las claves primarias están subrayadas. Obténgase una expresión del álgebra relacional para expresar cada una de las consultas siguientes:

- **a.** Determinar los nombres de todos los empleados que trabajan para el Banco Importante.
- **b.** Determinar el nombre y la ciudad de residencia de todos los empleados que trabajan para el Banco Importante.
- **c.** Determinar el nombre, la calle y la ciudad de residencia de todos los empleados que trabajan para el Banco Importante y ganan más de 10.000 € anuales.
- **d.** Determinar el nombre de todos los empleados de esta base de datos que viven en la misma ciudad que la compañía para la que trabajan.
- **e.** Supóngase que las compañías pueden estar instaladas en varias ciudades. Hállense todas las compañías instaladas en cada ciudad en la que está instalado el Banco Pequeño.

2.6 Considérese la relación de la Figura 2.20, que muestra el resultado de la consulta "Determinar el nombre de todos los clientes que tienen concedido un préstamo en el banco". Vuélvase a escribir la consulta para incluir no sólo el nombre, sino también la ciudad de residencia de cada cliente. Obsérvese que ahora el cliente Sotoca ya no aparece en el resultado, aunque en realidad tiene un préstamo concedido por el banco.

- **a.** Explíquese el motivo de que Sotoca no aparezca en el resultado.
- **b.** Supóngase que se desea que Sotoca aparezca en el resultado. ¿Cómo habría que modificar la base de datos para conseguirlo?
- **c.** Una vez más, supóngase que se desea que Sotoca aparezca en el resultado. Escríbase una consulta que utilice una reunión externa que cumpla esta condición sin que haya que modificar la base de datos.

2.7 Considérese la base de datos relacional de la Figura 2.35. Obténgase una expresión del álgebra relacional para cada petición:

- **a.** Dar a todos los empleados del Banco Importante un aumento de sueldo del 10 por ciento.
- **b.** Dar a todos los jefes de la base de datos un aumento de sueldo del 10 por ciento, a menos que el sueldo resultante sea mayor que 100.000 €. En ese caso, dar sólo un aumento del 3 por ciento.
- **c.** Borrar todas las tuplas de la relación *trabajo* de los empleados de Banco Pequeño.

2.8 Usando el ejemplo bancario, escríbanse consultas del álgebra relacional para encontrar las cuentas abiertas por más de dos clientes:

- **a.** Usando una función de agregación.
- **b.** Sin usar funciones de agregación.

2.9 Considérese la base de datos relacional de la Figura 2.35. Obténgase una expresión del álgebra relacional para cada una de las consultas siguientes:

- **a.** Determinar la compañía con mayor número de empleados.
- **b.** Determinar la compañía con la nómina más reducida.
- **c.** Determinar las compañías cuyos empleados ganen un sueldo más elevado, en media, que el sueldo medio del Banco Importante.

2.10 Determínense dos motivos por los que se puedan introducir valores nulos en la base de datos.

2.11 Considérese el siguiente esquema de relación:

empleado(*número_empleado*, *nombre*, *sucursal*, *edad*)
libros(*isbn*, *título*, *autores*, *editorial*)
préstamo(*número_empleado*, *isbn*, *fecha*)

Escríbanse las consultas siguientes en el álgebra relacional.

- **a.** Determinar el nombre de los empleados que han tomado prestados libros editados por McGraw-Hill.

b. Determinar el nombre de los empleados que han tomado prestados todos los libros editados por McGraw-Hill.
c. Determinar el nombre de los empleados que han tomado prestados más de cinco libros diferentes editados por McGraw-Hill.
d. Para cada editorial, determinar el nombre de los empleados que han tomado prestados más de cinco libros de esa editorial.

Notas bibliográficas

E.F. Codd, del Laboratorio de investigación de San José de IBM propuso el modelo relacional a finales de los años sesenta (Codd [1970]). Este trabajo le supuso la obtención del prestigioso Premio Turing de la ACM en 1981 (Codd [1982]).

Siguiendo el trabajo original de Codd se constituyeron varios proyectos de investigación con el objetivo de crear sistemas de bases de datos relacionales prácticos, como System R en el Laboratorio de investigación de IBM de San José, Ingres en la Universidad de California en Berkeley, Query-by-Example en el Centro de investigación de IBM T.J. Watson.

Atzeni y Antonellis [1993] y Maier [1983] son textos dedicados exclusivamente a la teoría del modelo relacional de datos.

La definición original del álgebra relacional está en Codd [1970]. En Codd [1979] se presentan ampliaciones del modelo relacional y explicaciones sobre la incorporación de los valores nulos al álgebra relacional (el modelo RM/T), así como la de las reuniones externas. Codd [1990] es un compendio de los trabajos de E.F. Codd sobre el modelo relacional. Las reuniones externas también se estudian en Date [1993b].

Actualmente están disponibles comercialmente numerosos productos de bases de datos relacionales. Ejemplos de ellos son DB2 de IBM, Oracle, Sybase, Informix y SQL Server de Microsoft. Entre los sistemas de bases de datos relacionales de código abierto están MySQL y PostgreSQL. Ejemplos de productos de bases de datos para uso personal son Access de Microsoft y FoxPro.

CAPÍTULO 3

SQL

El álgebra relacional descrita en el Capítulo 2 proporciona una notación concisa y formal para la representación de las consultas. Sin embargo, los sistemas de bases de datos comerciales necesitan un lenguaje de consultas más cómodo para el usuario. En este capítulo y en el Capítulo 4 se estudia SQL, el lenguaje de consultas distribuido comercialmente de más influencia. SQL usa una combinación de constructores del álgebra relacional (Capítulo 2) y del cálculo relacional (Capítulo 5).

Aunque se haga referencia al lenguaje SQL como "lenguaje de consultas", puede hacer mucho más que consultar las bases de datos. Usando SQL es posible además definir la estructura de los datos, modificar los datos de la base de datos y especificar restricciones de seguridad.

No se pretende proporcionar un manual de usuario completo de SQL. En cambio, se presentan los constructores y conceptos fundamentales de SQL. Las distintas implementaciones de SQL pueden diferenciarse en detalles o admitir sólo un subconjunto del lenguaje completo.

3.1 Introducción

IBM desarrolló la versión original de SQL, originalmente denominado Sequel, como parte del proyecto System R a principios de 1970. El lenguaje Sequel ha evolucionado desde entonces y su nombre ha pasado a ser SQL (Structured Query Language, lenguaje estructurado de consultas). Hoy en día, numerosos productos son compatibles con el lenguaje SQL y se ha establecido como *el* lenguaje estándar para las bases de datos relacionales.

En 1986, ANSI (American National Standards Institute, Instituto nacional americano de normalización) e ISO (International Standards Organization, Organización internacional de normalización) publicaron una norma SQL, denominada SQL-86. En 1989 ANSI publicó una extensión de la norma para SQL denominada SQL-89. La siguiente versión de la norma fue SQL-92 seguida de SQL:1999; la versión más reciente es SQL:2003. Las notas bibliográficas proporcionan referencias a esas normas.

El lenguaje SQL tiene varios componentes:

- **Lenguaje de definición de datos** (LDD). El LDD de SQL proporciona comandos para la definición de esquemas de relación, borrado de relaciones y modificación de los esquemas de relación.
- **Lenguaje interactivo de manipulación de datos** (LMD). El LMD de SQL incluye un lenguaje de consultas basado tanto en el álgebra relacional (Capítulo 2) como en el cálculo relacional de tuplas (Capítulo 5). También contiene comandos para insertar, borrar y modificar tuplas.
- **Integridad**. El LDD de SQL incluye comandos para especificar las restricciones de integridad que deben cumplir los datos almacenados en la base de datos. Las actualizaciones que violan las restricciones de integridad se rechazan.
- **Definición de vistas**. El LDD de SQL incluye comandos para la definición de vistas.

- **Control de transacciones**. SQL incluye comandos para especificar el comienzo y el final de las transacciones.
- **SQL incorporado** y **SQL dinámico**. SQL incorporado y SQL dinámico definen cómo se pueden incorporar instrucciones de SQL en lenguajes de programación de propósito general como C, C++, Java, PL/I, Cobol, Pascal y Fortran.
- **Autorización**. El LDD de SQL incluye comandos para especificar los derechos de acceso a las relaciones y a las vistas.

En este capítulo se presenta una visión general del LMD básico y de las características básicas del LDD de SQL. Esta descripción se basa principalmente en la muy extendida norma SQL-92, pero también se tratan algunas extensiones de las normas SQL:1999 y SQL:2003.

En el Capítulo 4 se ofrece un tratamiento más detallado del sistema de tipos de SQL, de las restricciones de integridad y de las autorizaciones. En ese capítulo también se describen brevemente SQL incorporado y SQL dinámico, incluyendo las normas ODBC y JDBC para la interacción con las bases de datos desde programas escritos en los lenguajes C y Java. En el Capítulo 9 se esbozan las extensiones orientadas a objetos de SQL que se introdujeron en SQL:1999.

Muchos sistemas de bases de datos soportan la mayor parte de la norma SQL-92 y parte de los nuevos constructores de SQL:1999 y SQL:2003, aunque actualmente ninguno soporta todos los constructores nuevos. También se debe tener en cuenta que muchos sistemas de bases de datos no soportan algunas características de SQL-92, y que muchas bases de datos presentan algunas no estándar que no se estudian en este libro. En caso de que alguna característica del lenguaje aquí descrita no funcione en el sistema de bases de datos que se esté utilizando, se debe consultar el manual de usuario para determinar exactamente lo que soporta.

La empresa que se utiliza para los ejemplos de este capítulo y de los posteriores es una entidad bancaria. La Figura 3.1 muestra el esquema relacional que se utiliza en los ejemplos con los atributos que son claves primarias subrayados. Recuérdese que en el Capítulo 2 ya se definió el esquema de una relación R mediante una relación de sus atributos, y posteriormente se definió una relación r del esquema mediante la notación $r(R)$. La notación de la Figura 3.1 omite el nombre del esquema y define el esquema de la relación directamente mediante una relación de sus atributos.

3.2 Definición de datos

El conjunto de relaciones de cada base de datos debe especificarse en el sistema en términos de un lenguaje de definición de datos (LDD). El LDD de SQL no sólo permite la especificación de un conjunto de relaciones, sino también de la información relativa a esas relaciones, incluyendo:

- El esquema de cada relación.
- El dominio de valores asociado a cada atributo.
- Las restricciones de integridad.
- El conjunto de índices que se deben mantener para cada relación.
- La información de seguridad y de autorización de cada relación.
- La estructura de almacenamiento físico de cada relación en el disco.

sucursal(<u>*nombre_sucursal*</u>, *ciudad_sucursal*, *activos*)
cliente (<u>*nombre_cliente*</u>, *calle_cliente*, *ciudad_cliente*)
préstamo (<u>*número_préstamo*</u>, *nombre_sucursal*, *importe*)
prestatario (<u>*nombre_cliente*</u>, <u>*número_préstamo*</u>)
cuenta (<u>*número_cuenta*</u>, *nombre_sucursal*, *saldo*)
impositor (<u>*nombre_cliente*</u>, <u>*número_cuenta*</u>)

Figura 3.1 Esquema de la entidad bancaria.

A continuación se analizará la definición de los esquemas y los valores básicos de los dominios; el análisis de las demás características del LDD de SQL se tratará en el Capítulo 4.

3.2.1 Tipos básicos de dominios

La norma SQL soporta gran variedad de tipos de dominio predefinidos, entre ellos:

- **char**(n). Una cadena de caracteres de longitud fija, con una longitud n especificada por el usuario. También se puede utilizar la palabra completa **character**.
- **varchar**(n). Una cadena de caracteres de longitud variable con una longitud máxima n especificada por el usuario. La forma completa, **character varying**, es equivalente.
- **int**. Un entero (un subconjunto finito de los enteros dependiente de la máquina). La palabra completa, **integer**, es equivalente.
- **smallint**. Un entero pequeño (un subconjunto dependiente de la máquina del tipo de dominio entero).
- **numeric**(p, d). Un número de coma fija, cuya precisión la especifica el usuario. El número está formado por p dígitos (más el signo), y de esos p dígitos, d pertenecen a la parte decimal. Así, **numeric**(3,1) permite que el número 44.5 se almacene exactamente, pero ni 444.5 ni 0.32 se pueden almacenar exactamente en un campo de este tipo.
- **real, double precision**. Números de coma flotante y números de coma flotante de doble precisión, con precisión dependiente de la máquina.
- **float**(n). Un número de coma flotante cuya precisión es, al menos, de n dígitos.

En el Apartado 4.1 se tratan otros tipos de dominio.

3.2.2 Definición básica de esquemas en SQL

Las relaciones se definen mediante el comando **create table**:

create table $r(A_1\ D_1, A_2\ D_2, \ldots, A_n\ D_n,$
$\langle$restricción-integridad$_1\rangle$,
$\ldots,$
$\langle$restricción-integridad$_k\rangle)$

donde *r* es el nombre de la relación, cada A_i es el nombre de un atributo del esquema de la relación *r* y D_i es el tipo de dominio de los valores del dominio del atributo A_i. Hay varias restricciones de integridad válidas. En este apartado sólo se estudiarán las de clave primaria (primary key), que adopta la forma:

- **primary key** $(A_{j_1}, A_{j_2}, \ldots, A_{j_m})$. La especificación de **clave primaria** determina que los atributos $A_{j_1}, A_{j_2}, \ldots, A_{j_m}$ forman la clave primaria de la relación. Los atributos de la clave primaria tienen que ser *no nulos* y *únicos*; es decir, ninguna tupla puede tener un valor nulo para un atributo de la clave primaria y ningún par de tuplas de la relación puede ser igual en todos los atributos de la clave primaria[1]. Aunque la especificación de clave primaria es opcional, suele ser una buena idea especificar una clave primaria para cada relación.

Otras restricciones de integridad que puede incluir el comando **create table** se tratan más adelante, en el Apartado 4.2.

La Figura 3.2 presenta una definición parcial en el LDD de SQL de la base de datos bancaria. Obsérvese que, al igual que en capítulos anteriores, no se intenta modelar con precisión el mundo real en el ejemplo de la base de datos bancaria. En el mundo real varias personas pueden llamarse igual, por lo que *nombre_cliente* no sería clave primaria de *cliente*; probablemente se utilizaría un *id_cliente* como clave primaria.

1. En SQL-89, los atributos que forman la clave primaria no estaban declarados implícitamente como **not null**; era necesaria una declaración explícita.

```
create table cliente
    (nombre_cliente     char(20),
     calle_cliente      char(30),
     ciudad_cliente     char(30),
     primary key (nombre_cliente))

create table sucursal
    (nombre_sucursal    char(15),
     ciudad_sucursal    char(30),
     activos            numeric(16,2),
     primary key (nombre_sucursal))

create table cuenta
    (número_cuenta      char(10),
     nombre_sucursal    char(15),
     saldo              numeric(12,2),
     primary key (número_cuenta))

create table impositor
    (nombre_cliente     char(20),
     número_cuenta      char(10),
     primary key (nombre_cliente, número_cuenta))
```

Figura 3.2 Definición de datos en SQL para parte de la base de datos del banco.

Aquí se utiliza *nombre_cliente* como clave primaria para mantener el esquema de la base de datos sencillo y de pequeño tamaño.

Si una tupla recién insertada o recién modificada de una relación contiene valores nulos para cualquiera de los atributos que forman parte de la clave primaria, o si tiene el mismo valor en ellos que otra tupla de la relación, SQL notifica el error e impide la actualización.

Las relaciones recién creadas están inicialmente vacías. Se puede utilizar el comando **insert** para añadir datos a la relación. Por ejemplo, si se desea añadir el hecho de que hay una cuenta C-9732 en la sucursal de Navacerrada con un saldo de 1.200 €, hay que escribir

```
insert into cuenta
        values ('C-9732', 'Navacerrada', 1200)
```

Los valores se especifican en el orden en que se relacionan los atributos correspondientes en el esquema de la relación. El comando **insert** ofrece una serie de características útiles, y se trata con más detalle en el Apartado 3.10.2.

Se puede utilizar el comando **delete** para borrar tuplas de una relación. El comando

```
delete from cuenta
```

borraría todas las tuplas de la relación *cuenta*. Otras formas del comando **delete** permiten borrar sólo tuplas concretas; el comando **delete** se trata con más detalle en el Apartado 3.10.1.

Para eliminar una relación de una base de datos SQL se utiliza el comando **drop table**. Este comando elimina de la base de datos toda la información de la relación. La instrucción

```
drop table r
```

es una acción más drástica que

```
delete from r
```

La última conserva la relación *r*, pero borra todas sus tuplas. La primera no sólo borra todas las tuplas de la relación *r*, sino que también elimina su esquema. Una vez eliminada *r*, no se puede insertar ninguna tupla en dicha relación, a menos que se vuelva a crear con la instrucción **create table**.

El comando **alter table** se utiliza para añadir atributos a una relación existente. Se asigna a todas las tuplas de la relación un valor *nulo* como valor del atributo nuevo. La forma del comando **alter table** es

alter table *r* **add** A D

donde *r* es el nombre de una relación ya existente, A es el nombre del atributo que se desea añadir y D es el dominio del atributo añadido. Se pueden eliminar atributos de una relación utilizando el comando

alter table *r* **drop** A

donde *r* es el nombre de una relación ya existente y A es el nombre de un atributo de la relación. Muchos sistemas de bases de datos no permiten el borrado de atributos, aunque sí permiten el borrado de tablas completas.

3.3 Estructura básica de las consultas SQL

Las bases de datos relacionales están formadas por un conjunto de relaciones, a cada una de las cuales se le asigna un nombre único. Cada relación posee una estructura similar a la presentada en el Capítulo 2. SQL permite el uso de valores nulos para indicar que el valor es desconocido o no existe. También permite al usuario especificar los atributos que no pueden contener valores nulos, como se indicó en el Apartado 3.2.

La estructura básica de una expresión SQL consta de tres cláusulas: **select**, **from** y **where**.

- La cláusula **select** se corresponde con la operación proyección del álgebra relacional. Se usa para obtener una relación de los atributos deseados en el resultado de una consulta.
- La cláusula **from** se corresponde con la operación producto cartesiano del álgebra relacional. Genera una lista de las relaciones que deben ser analizadas en la evaluación de la expresión.
- La cláusula **where** se corresponde con el predicado selección del álgebra relacional. Es un predicado que engloba a los atributos de las relaciones que aparecen en la cláusula **from**.

Que el término *select* tenga un significado diferente en SQL que en el álgebra relacional es un hecho histórico desafortunado. En este capítulo se destacan las diferentes interpretaciones para reducir al mínimo las posibles confusiones.

Las consultas habituales de SQL tienen la forma

select $A_1,\ A_2, \ldots, A_n$
from $r_1,\ r_2, \ldots, r_m$
where P

Cada A_i representa un atributo y cada r_i, una relación. P es un predicado. La consulta es equivalente a la expresión del álgebra relacional

$$\Pi_{A_1,\ A_2, \ldots,\ A_n}(\sigma_P\,(r_1\ \times\ r_2\ \times\ \cdots\ \times\ r_m))$$

Si se omite la cláusula **where**, el predicado P es **cierto**. Sin embargo, a diferencia de la expresión del álgebra relacional, el resultado de la consulta SQL puede contener varias copias de algunas tuplas; este aspecto se analizará de nuevo en el Apartado 3.3.8.

SQL forma el producto cartesiano de las relaciones incluidas en la cláusula **from**, lleva a cabo la selección del álgebra relacional utilizando el predicado de la cláusula **where** y después proyecta el resultado sobre los atributos de la cláusula **select**. En la práctica, SQL puede convertir la expresión en una forma equivalente que puede ser procesada de manera más eficiente. No obstante, las cuestiones relativas a la eficiencia se posponen hasta los Capítulos 13 y 14.

3.3.1 La cláusula select

El resultado de las consultas SQL es, por supuesto, una relación. Considérese la consulta simple basada en el ejemplo bancario "Obtener el nombre de todas las sucursales de la relación *préstamo*":

select *nombre_sucursal*
from *préstamo*

El resultado es una relación consistente en un único atributo con el encabezado *nombre_sucursal*.

Los lenguajes formales de consultas están basados en el concepto matemático de que las relaciones son conjuntos. Así, nunca aparecen tuplas duplicadas en las relaciones. En la práctica, la eliminación de duplicados consume tiempo. Por tanto, SQL (como la mayoría de los lenguajes de consultas comerciales) permite duplicados en las relaciones, así como en el resultado de las expresiones SQL. Así, la consulta anterior mostrará el valor de *nombre_sucursal* una vez por cada tupla de la relación *préstamo* en la que aparezca.

En los casos en los que se desea forzar la eliminación de los valores duplicados, se inserta la palabra clave **distinct** después de **select**. Se puede reescribir la consulta anterior como

select distinct *nombre_sucursal*
from *préstamo*

si se desea eliminar los duplicados.

SQL permite utilizar la palabra clave **all** para especificar de manera explícita que no se eliminen los duplicados:

select all *nombre_sucursal*
from *préstamo*

Dado que la retención de duplicados es la opción predeterminada, no se utilizará **all** en los ejemplos. Para garantizar la eliminación de duplicados en el resultado de los ejemplos de consultas, se usará la cláusula **distinct** siempre que sea necesario. En la mayoría de las consultas en que no se utiliza **distinct**, el número exacto de copias duplicadas de cada tupla presentes en el resultado de la consulta no es importante. Sin embargo, el número sí que es importante en ciertas aplicaciones; esto se volverá a tratar en el Apartado 3.3.8.

El símbolo asterisco " * " se puede usar para denotar "todos los atributos". Así, el uso de *préstamo*.* en la cláusula **select** anterior indicaría que se seleccionarían todos los atributos de *préstamo*. Las cláusulas **select** de la forma **select** * indican que se deben seleccionar todos los atributos de todas las relaciones que aparecen en la cláusula **from**.

La cláusula **select** puede contener también expresiones aritméticas que contengan los operadores $+$, $-$, $*$ y $/$ y operen sobre constantes o atributos de la tuplas. Por ejemplo, la consulta

select *número_préstamo*, *nombre_sucursal*, *importe* * 100
from *préstamo*

devolverá una relación que es igual que la relación *préstamo*, salvo que el atributo *importe* está multiplicado por 100.

SQL también proporciona tipos de datos especiales, tales como varias formas del tipo *date* (fecha) y permite que varias funciones aritméticas operen sobre esos tipos.

3.3.2 La cláusula where

A continuación se muestra el uso de la cláusula **where** en SQL. Considérese la consulta "Obtener todos los números de préstamo de los préstamos concedidos en la sucursal de Navacerrada e importe superior a 1.200 €". Esta consulta puede escribirse en SQL como:

select *número_préstamo*
from *préstamo*
where *nombre_sucursal* = 'Navacerrada' **and** *importe* > 1200

SQL usa las conectivas lógicas **and**, **or** y **not** (en lugar de los símbolos matemáticos $\wedge$, $\vee$ y $\neg$) en la cláusula **where**. Los operandos de las conectivas lógicas pueden ser expresiones que contengan los operadores de comparación <, <=, >, >=, = y <>. SQL permite utilizar los operadores de comparación para comparar cadenas y expresiones aritméticas, así como tipos especiales, como los tipos de fecha.

SQL incluye también un operador de comparación **between** para simplificar las cláusulas **where**, el cual especifica que un valor sea menor o igual que un valor y mayor o igual que otro valor. Si se desea obtener el número de préstamo de aquellos préstamos con importe entre 90.000 € y 100.000 €, se puede usar la comparación **between** para escribir

select *número_préstamo*
from *préstamo*
where *importe* **between** 90000 **and** 100000

en lugar de

select *número_préstamo*
from *préstamo*
where *importe* >= 90000 **and** *importe* <= 100000

De forma análoga, se puede usar el operador de comparación **not between**.

3.3.3 La cláusula from

Finalmente, se estudiará el uso de la cláusula **from**. La cláusula **from** define por sí misma un producto cartesiano de las relaciones que aparecen en la cláusula. Dado que la reunión natural se define en términos de un producto cartesiano, una selección y una proyección, resulta relativamente sencillo escribir una expresión de SQL para la reunión natural. Se escribe la expresión del álgebra relacional

$$\Pi_{nombre_cliente,\, número_préstamo,\, importe} \, (prestatario \bowtie préstamo)$$

para la consulta "Para todos los clientes que tienen un préstamo del banco, obtener el nombre, el número de préstamo y su importe". Esta consulta puede escribirse en SQL como

select *nombre_cliente, prestatario.número_préstamo, importe*
from *prestatario, préstamo*
where *prestatario.número_préstamo = préstamo.número_préstamo*

Obsérvese que SQL usa la notación *nombre-relación.nombre-atributo*, como lo hace el álgebra relacional, para evitar ambigüedades en los casos en los que un atributo aparece en más de un esquema de relación. También se podría haber usado *prestatario.nombre_cliente* en lugar de *nombre_cliente* en la cláusula **select**. Sin embargo, como el atributo *nombre_cliente* sólo aparece en una de las relaciones de la cláusula **from**, no existe ambigüedad al escribir *nombre_cliente*.

Se puede extender la consulta anterior y considerar un caso más complicado en el que se también se exija que el préstamo se haya concedido en la sucursal de Navacerrada: "Determinar el nombre, el número de préstamo y su importes de todos los préstamos de la sucursal de Navacerrada". Para escribir esta consulta hay que definir dos restricciones en la cláusula **where**, enlazadas con la conectiva lógica **and**:

select *nombre_cliente, prestatario.número_préstamo, importe*
from *prestatario, préstamo*
where *prestatario.número_préstamo = préstamo.número_préstamo* **and**
nombre_sucursal = 'Navacerrada'

SQL incluye extensiones para llevar a cabo reuniones naturales y reuniones externas en la cláusula **from**. Estas extensiones se estudian en el Apartado 3.11.

3.3.4 La operación renombramiento

SQL proporciona un mecanismo para renombrar tanto relaciones como atributos. Utiliza la cláusula **as** de la forma siguiente:

nombre-antiguo **as** *nombre-nuevo*

La cláusula **as** puede aparecer tanto en la cláusula **select** como en la cláusula **from**.

Considérese de nuevo la consulta anterior:

select *nombre_cliente, prestatario.número_préstamo, importe*
from *prestatario, préstamo*
where *prestatario.número_préstamo* = *préstamo.número_préstamo*

El resultado de esta consulta es una relación con los atributos siguientes:

nombre_cliente, número_préstamo, importe

Los nombres de los atributos en el resultado proceden de los nombres de los atributos de las relaciones que aparecen en la cláusula **from**.

Sin embargo, no se pueden obtener siempre los nombres de este modo por varios motivos. En primer lugar, dos relaciones de la cláusula **from** pueden tener atributos con el mismo nombre, en cuyo caso, un nombre de atributo aparecerá duplicado en el resultado. En segundo lugar, si se ha utilizado una expresión aritmética en la cláusula **select**, los atributos resultantes no tienen nombre. En tercer lugar, incluso si el nombre de un atributo se puede obtener a partir de las relaciones base, como en el ejemplo anterior, puede que se desee cambiar el nombre del atributo en el resultado. Por ello, SQL proporciona una forma de renombrar los atributos de las relaciones resultantes.

Por ejemplo, si se desea cambiar el nombre del atributo *número_préstamo* por *id_préstamo*, se puede volver a escribir la consulta anterior como

select *nombre_cliente, prestatario.número_préstamo* **as** *id_préstamo, importe*
from *prestatario, préstamo*
where *prestatario.número_préstamo* = *préstamo.número_préstamo*

3.3.5 Variables tupla

La cláusula **as** resulta especialmente útil en la definición del concepto de variable tupla. Las variables tupla en SQL se deben asociar con una relación concreta. Las variables tupla se definen en la cláusula **from** mediante la cláusula **as**. Para ilustrarlo, se reescribirá la consulta "Para todos los clientes que tienen concedido un préstamo por el banco, obtener el nombre, el número de préstamo y su importe" como

select *nombre_cliente, T.número_préstamo, S.importe*
from *prestatario* **as** *T*, *préstamo* **as** *S*
where *T.número_préstamo* = *S.número_préstamo*

Obsérvese que la variable tupla se define en la cláusula **from** colocándola después del nombre de la relación a la cual está asociada y detrás de la palabra clave **as** (la palabra clave **as** es opcional). Al escribir expresiones de la forma *nombre_relación.nombre_atributo*, el nombre de la relación es, en efecto, una variable tupla definida implícitamente.

Las variables tupla son de gran utilidad para comparar dos tuplas de la misma relación. Hay que recordar que, en estos casos, se puede usar la operación renombramiento del álgebra relacional. Supóngase que se desea formular la consulta "Determinar el nombre de todas las sucursales que tienen activos mayores que, al menos, una sucursal de Arganzuela". Se puede escribir la expresión SQL siguiente:

select distinct *T.nombre_sucursal*
from *sucursal* **as** *T*, *sucursal* **as** *S*
where *T.activos* $>$ *S.activos* **and** *S.ciudad_sucursal* $=$ 'Arganzuela'

Obsérvese que no se puede utilizar la notación *sucursal.activos*, puesto que no estaría claro qué referencia a *sucursal* se pretende hacer.

SQL permite usar la notación $(v_1, v_2, \ldots, v_n)$ para designar una tupla de aridad n que contiene los valores $v_1, v_2, \ldots, v_n$. Los operadores de comparación se pueden utilizar sobre las tuplas, y el orden se define lexicográficamente. Por ejemplo, $(a_1, a_2) <= (b_1, b_2)$ es cierto si $a_1 < b_1$, o si se cumple que $(a_1 = b_1) \wedge (a_2 <= b_2)$; de manera parecida, dos tuplas son iguales si lo son todos sus atributos.

3.3.6 Operaciones con cadenas de caracteres

SQL especifica las cadenas de caracteres encerrándolas entre comillas simples, como en 'Navacerrada', como ya se ha visto anteriormente. Los caracteres de comillas que formen parte de una cadena de caracteres se pueden especificar usando dos caracteres de comillas; por ejemplo, la cadena de caracteres "Peter O'Toole es un gran actor" se puede especificar como "Peter O''Toole es un gran actor".

La operación más utilizada sobre las cadenas de caracteres es la comparación de patrones, para la que se usa el operador **like**. Para la descripción de los patrones se utilizan dos caracteres especiales:

- Tanto por ciento (%). El carácter % coincide con cualquier subcadena de caracteres.
- Subrayado (_). El carácter _ coincide con cualquier carácter.

Los patrones distinguen la caja; es decir, los caracteres en mayúsculas se consideran diferentes de los caracteres en minúscula, y viceversa. Para ilustrar la comparación de patrones, se considerarán los siguientes ejemplos:

- 'Nava %' coincide con cualquier cadena de caracteres que empiece con "Nava".
- ' %cer %' coincide con cualquier cadena que contenga "cer" como subcadena, por ejemplo 'Navacerrada', 'Cáceres' y 'Becerril'.
- '___' coincide con cualquier cadena que tenga exactamente tres caracteres.
- '___%' encaja con cualquier cadena que tenga, al menos, tres caracteres.

Los patrones se expresan en SQL utilizando el operador de comparación **like**. Considérese la consulta "Determinar el nombre de todos los clientes cuya dirección contenga la subcadena de caracteres 'Mayor'". Esta consulta se puede escribir como

select *nombre_cliente*
from *cliente*
where *calle_cliente* **like** ' %Mayor %'

Para que los patrones puedan contener los caracteres especiales de patrón (esto es, % y _) SQL permite la especificación de un carácter de escape. El carácter de escape se utiliza inmediatamente antes de los caracteres especiales de patrón para indicar que ese carácter especial se va a tratar como un carácter normal. El carácter de escape para una comparación **like** se define utilizando la palabra clave **escape**. Para ilustrar esto, considérense los siguientes patrones, que utilizan una barra invertida (\) como carácter de escape:

- **like** 'ab\ %cd %' **escape** '\' coincide con todas las cadenas que empiecen por "ab %cd".
- **like** 'ab\\cd %' **escape** '\' coincide con todas las cadenas que empiecen por "ab\cd".

SQL permite buscar discordancias en lugar de concordancias utilizando el operador de comparación **not like**.

SQL también permite varias funciones que operan sobre cadenas de caracteres, tales como la concatenación (utilizando "||"), la extracción de subcadenas de caracteres, el cálculo de la longitud de las cadenas de caracteres, la conversión a mayúsculas (utilizando **upper**()) y minúsculas (utilizando **lower**()), etc. SQL:1999 también ofrece una operación **similar to**, que proporciona una comparación de patrones más potente que la operación **like**; la sintaxis para especificar patrones es parecida a la usada para las expresiones regulares de Unix.

Existen variaciones en el conjunto concreto de funciones para cadenas de caracteres que soportan los diferentes sistemas de bases de datos. Algunos sistemas de bases de datos no distinguen entre mayúsculas y minúsculas al comparar cadenas de caracteres. Así, 'ABC' **like** 'abc' daría como resultado cierto, igual que 'ABC' = 'abc', en esos sistemas. Otros ofrecen extensiones para especificar que la comparación de cadenas de caracteres debe ignorar si están en mayúsculas o en minúsculas. Conviene leer el manual de cada sistema de bases de datos para conocer más detalles sobre las funciones de cadenas de caracteres que soporta exactamente.

3.3.7 Orden en la presentación de las tuplas

SQL ofrece al usuario cierto control sobre el orden en el cual se presentan las tuplas de una relación. La cláusula **order by** hace que las tuplas del resultado de una consulta se presenten en un cierto orden. Para obtener una relación en orden alfabético de todos los clientes que tienen un préstamo en la sucursal de Navacerrada hay que escribir:

select distinct *nombre_cliente*
from *prestatario, préstamo*
where *prestatario.número_préstamo* = *préstamo.número_préstamo* **and**
nombre_sucursal = 'Navacerrada'
order by *nombre_cliente*

De manera predeterminada, la cláusula **order by** coloca los elementos en orden ascendente. Para especificar el tipo de ordenación se puede especificar **desc** para orden descendente o **asc** para orden ascendente. Además, se puede ordenar con respecto a más de un atributo. Supóngase que se desea colocar toda la relación *préstamo* en orden descendente de *importe*. Si varios préstamos tienen el mismo importe, se ordenan de manera ascendente según sus números de préstamo. Esta consulta en SQL se expresa del modo siguiente:

select *
from *préstamo*
order by *importe* **desc**, *número_préstamo* **asc**

SQL debe llevar a cabo una ordenación para evaluar **order by**. Como ordenar un gran número de tuplas puede resultar costoso, sólo debería hacerse cuando sea estrictamente necesario.

3.3.8 Duplicados

El uso de relaciones con duplicados presenta ventajas en diferentes situaciones. En consecuencia, SQL no sólo define las tuplas que están en el resultado de una consulta, sino también el número de copias de cada una de esas tuplas que aparece en el resultado. La semántica de duplicados de una consulta SQL se define utilizando versiones *multiconjunto* de los operadores relacionales. A continuación se definen las versiones multiconjunto de varios de los operadores del álgebra relacional. Dadas las relaciones multiconjunto r_1 y r_2,

1. Si existen c_1 copias de la tupla t_1 en r_1, y t_1 satisface la selección σ_θ, entonces hay c_1 copias de t_1 en $\sigma_\theta(r_1)$.
2. Para cada copia de la tupla t_1 en r_1, hay una copia de la tupla $\Pi_A(t_1)$ en $\Pi_A(r_1)$, donde $\Pi_A(t_1)$ denota la proyección de la única tupla t_1.

3. Si existen c_1 copias de la tupla t_1 en r_1 y c_2 copias de la tupla t_2 en r_2, entonces hay $c_1 * c_2$ copias de la tupla $t_1.t_2$ en $r_1 \times r_2$.

Por ejemplo, supóngase que las relaciones r_1 con esquema (A, B) y r_2 con esquema (C) son los multiconjuntos siguientes:

$$r_1 = \{(1, a), (2, a)\} \qquad r_2 = \{(2), (3), (3)\}$$

Entonces, $\Pi_B(r_1)$ sería $\{(a), (a)\}$, mientras que $\Pi_B(r_1) \times r_2$ sería

$$\{(a, 2), (a, 2), (a, 3), (a, 3), (a, 3), (a, 3)\}$$

Ahora se puede definir cuántas copias de cada tupla aparecen en el resultado de una consulta SQL. Una consulta SQL de la forma

select $A_1,\ A_2, \ldots, A_n$
from $r_1,\ r_2, \ldots, r_m$
where P

es equivalente a la expresión del álgebra relacional

$$\Pi_{A_1,\ A_2, \ldots, A_n}(\sigma_P\,(r_1\ \times\ r_2\ \times\ \cdots\ \times\ r_m))$$

que usa las versiones multiconjunto de los operadores relacionales σ, Π y $\times$.

3.4 Operaciones sobre conjuntos

Las operaciones de SQL **union**, **intersect** y **except** operan sobre relaciones y se corresponden con las operaciones del álgebra relacional $\cup$, $\cap$ y $-$. Al igual que la unión, la intersección y la diferencia de conjuntos del álgebra relacional, las relaciones que participan en las operaciones han de ser *compatibles*; esto es, deben tener el mismo conjunto de atributos.

A continuación se estudia el modo de formular en SQL varias de las consultas de ejemplo consideradas en el Capítulo 2. Ahora se crearán consultas que incluyan las operaciones **union**, **intersect** y **except** de dos conjuntos: el conjunto de todos los clientes que tienen cuenta en el banco, que puede obtenerse con

select *nombre_cliente*
from *impositor*

y el conjunto de los clientes que tienen concedido un préstamo por el banco, que puede obtenerse con

select *nombre_cliente*
from *prestatario*

A partir de ahora, se hará referencia a las relaciones obtenidas como resultado de las consultas anteriores como *i* y *p*, respectivamente.

3.4.1 La operación unión

Para determinar todos los clientes del banco que tienen un préstamo, una cuenta o las dos cosas en el banco, hay que escribir:

(**select** *nombre_cliente*
from *impositor*)
union
(**select** *nombre_cliente*
from *prestatario*)

A diferencia de la cláusula **select**, la operación **union** (unión) elimina los valores duplicados automáticamente. Así, en la consulta anterior, si un cliente—por ejemplo, Santos—tiene varias cuentas o préstamos (o ambas cosas) en el banco, sólo aparecerá una vez en el resultado.

Si se desea conservar todos los duplicados hay que escribir **union all** en lugar de **union**:

(**select** *nombre_cliente*
from *impositor*)
union all
(**select** *nombre_cliente*
from *prestatario*)

El número de tuplas duplicadas en el resultado es igual al número total de valores duplicados que aparecen en *i* y *p*. Así, si Santos tiene tres cuentas y dos préstamos en el banco, en el resultado aparecerán cinco tuplas con el nombre de Santos.

3.4.2 La operación intersección

Para encontrar todos los clientes que tienen tanto un préstamo como una cuenta en el banco, hay que escribir:

(**select distinct** *nombre_cliente*
from *impositor*)
intersect
(**select distinct** *nombre_cliente*
from *prestatario*)

La operacion **intersect** (intersección) elimina los valores duplicados automáticamente. Así, en la consulta anterior, si un cliente—por ejemplo, Santos—tiene varias cuentas o préstamos en el banco, sólo aparecerá una vez en el resultado.

Si se desean conservar todos los valores duplicados hay que escribir **intersect all** en lugar de **intersect**:

(**select** *nombre_cliente*
from *impositor*)
intersect all
(**select** *nombre_cliente*
from *prestatario*)

El número de tuplas duplicadas que aparecen en el resultado es igual al número mínimo de valores duplicados que aparecen en *i* y *p*. Por tanto, si Santos tiene tres cuentas y dos préstamos en el banco, en el resultado de la consulta aparecerán dos tuplas con el nombre de Santos.

3.4.3 La operación excepto

Para determinar todos los clientes que tienen cuenta en el banco pero no tienen ningún préstamo concedido hay que escribir

(**select distinct** *nombre_cliente*
from *impositor*)
except
(**select** *nombre_cliente*
from *prestatario*)

La operación **except** (excepto) elimina los valores duplicados automáticamente. Así, en la consulta anterior, sólo aparecerá en el resultado (exactamente una vez) una tupla con el nombre de Santos si Santos tiene una cuenta en el banco pero no tiene ningún préstamo concedido.

Si se desea conservar todos los valores duplicados hay que escribir **except all** en lugar de **except**:

(**select** *nombre_cliente*
from *impositor*)
except all
(**select** *nombre_cliente*
from *prestatario*)

El número de copias duplicadas de una tupla en el resultado es igual al número de copias duplicadas de dicha tupla en *impositor* menos el número de copias duplicadas de la misma tupla en *prestatario*, siempre que la diferencia sea positiva. Así, si Santos tiene tres cuentas y un préstamo en el banco, en el resultado aparecerán dos tuplas con el nombre de Santos. Si, por el contrario, ese cliente tiene dos cuentas y tres préstamos en el banco, en el resultado no habrá ninguna tupla con el nombre de Santos.

3.5 Funciones de agregación

Las *funciones de agregación* son funciones que toman una colección (un conjunto o multiconjunto) de valores como entrada y devuelven un solo valor. SQL ofrece cinco funciones de agregación incorporadas:

- Media: **avg**
- Mínimo: **min**
- Máximo: **max**
- Total: **sum**
- Recuento: **count**

Los datos de entrada para **sum** y **avg** deben ser una colección de números, pero los otros operadores también pueden operar sobre colecciones de datos de tipo no numérico como las cadenas de caracteres.

A modo de ejemplo, considérese la consulta "Determinar el saldo medio de las cuentas de la sucursal de Navacerrada". Esta consulta se puede formular del modo siguiente:

select avg (*saldo*)
from *cuenta*
where *nombre_sucursal* = 'Navacerrada'

El resultado de esta consulta es una relación con un único atributo, que contiene una sola tupla con un valor numérico correspondiente al saldo medio de la sucursal de Navacerrada. Opcionalmente se puede dar un nombre al atributo de la relación resultante, utilizando la cláusula **as**.

Existen circunstancias en las cuales sería deseable aplicar las funciones de agregación no sólo a un único conjunto de tuplas sino también a un grupo de conjuntos de tuplas; esto se especifica en SQL mediante la cláusula **group by**. El atributo o atributos especificados en la cláusula **group by** se usan para formar grupos. Las tuplas con el mismo valor en todos los atributos de la cláusula **group by** constituyen un mismo grupo.

Como ejemplo, considérese la consulta "Determinar el saldo medio de las cuentas de cada sucursal". Esta consulta se formula del modo siguiente:

select *nombre_sucursal*, **avg** (*saldo*)
from *cuenta*
group by *nombre_sucursal*

La conservación de los valores duplicados es importante para el cálculo de medias. Supóngase que los saldos de las cuentas en la (pequeña) sucursal de Galapagar son 1.000 €, 3.000 €, 2.000 € y 1.000 €. El saldo medio es 7.000/4 = 1.750,00 €. Si se eliminasen los valores duplicados se obtendría un resultado erróneo (6.000/3 = 2.000 €).

A veces se desea tratar toda la relación como un solo grupo. En estos casos no se utiliza la cláusula **group by**. Considérese la consulta "Determinar el saldo medio de todas las cuentas". Esta consulta se formula del modo siguiente:

```
select avg (saldo)
from cuenta
```

Con frecuencia se usa la función de agregación **count** para contar el número de tuplas de una relación. La notación de SQL para esta función es **count** (*). Por tanto, para determinar el número de tuplas de la relación *cliente* hay que escribir

```
select count (*)
from cliente
```

Se dan casos en los cuales es necesario eliminar los valores duplicados antes de calcular una función de agregación. Si se desea eliminar los valores duplicados hay que utilizar la palabra clave **distinct** en la expresión de agregación. A modo de ejemplo, considérese la consulta "Determinar el número de impositores de cada sucursal". En este caso cada impositor sólo se debe contar una vez, independientemente del número de cuentas que pueda tener. La consulta se formula del modo siguiente:

```
select nombre_sucursal, count (distinct nombre_cliente)
from impositor, cuenta
where impositor.número_cuenta = cuenta.número_cuenta
group by nombre_sucursal
```

SQL no permite el uso de **distinct** con **count** (*). Sí es legal el uso de **distinct** con **max** y **min**, aunque el resultado no cambia. Se puede utilizar la palabra clave **all** en lugar de **distinct** para especificar la conservación de los valores duplicados pero, como **all** es la opción predeterminada, no hace falta hacerlo.

A veces resulta útil establecer una condición que se aplique a los grupos en vez de a las tuplas. Por ejemplo, puede que sólo estemos interesados en las sucursales en las que el saldo medio de las cuentas sea superior a 1.200 €. Esta condición no se aplica a una sola tupla, sino a cada grupo creado por la cláusula **group by**. Para expresar este tipo de consultas se utiliza la cláusula **having** de SQL. SQL aplica los predicados de la cláusula **having** una vez formados los grupos, de modo que se puedan usar las funciones de agregación. Esta consulta se expresa en SQL del modo siguiente:

```
select nombre_sucursal, avg (saldo)
from cuenta
group by nombre_sucursal
having avg (saldo) > 1200
```

Si en la misma consulta aparecen una cláusula **where** y una cláusula **having**, SQL aplica primero el predicado de la cláusula **where**. Las tuplas que satisfacen el predicado de la cláusula **where** se dividen luego en grupos según la cláusula **group by**. Después se aplica la cláusula **having** (si existe) a cada grupo; SQL elimina los grupos que no satisfacen el predicado de la cláusula **having**. La cláusula **select** utiliza los grupos restantes para generar las tuplas del resultado de la consulta.

Para ilustrar el uso conjunto de la cláusula **where** y la cláusula **having** en la misma consulta, considérese la consulta "Determinar el saldo medio de cada cliente que vive en Peguerinos y tiene, como mínimo, tres cuentas".

```
select impositor.nombre_cliente, avg (saldo)
from impositor, cuenta, cliente
where impositor.número_cuenta = cuenta.número_cuenta and
      impositor.nombre_cliente = cliente.nombre_cliente and
      ciudad_cliente = 'Peguerinos'
group by impositor.nombre_cliente
having count (distinct impositor.número_cuenta) >= 3
```

3.6 Valores nulos

SQL permite el uso de valores *nulos* para indicar la ausencia de información sobre el valor de un atributo.

Se puede utilizar la palabra clave especial **null** en un predicado para comprobar si un valor es nulo. Así, para determinar todos los números de préstamo que aparecen en la relación *préstamo* con valores nulos para *importe* hay que escribir

select *número_préstamo*
from *préstamo*
where *importe* **is null**

El predicado **is not null** comprueba la ausencia de valores nulos.

El uso de valores *nulos* en las operaciones aritméticas y de comparación causa algunos problemas. En el Apartado 2.5 se vio cómo se manejan los valores nulos en el álgebra relacional. Ahora se va a describir cómo lo hace SQL.

El resultado de las expresiones aritméticas (que incluyan por ejemplo $+$, $-$, $*$ o $/$) es nulo si cualquiera de los valores de entrada es nulo. SQL trata como **unknown** (desconocido) el resultado de cualquier comparación que implique un valor *nulo* (excepto con **is null** y **is not null**).

Dado que el predicado de las cláusulas **where** puede incluir operaciones booleanas como **and**, **or** y **not** sobre los resultados de las comparaciones, se amplían las definiciones de las operaciones booleanas para que manejen el valor **unknown**, como se describe en el Apartado 2.5.

- **and**. El resultado de *cierto* **and** *desconocido* es *desconocido*, *falso* **and** *desconocido* es *falso*, mientras que *desconocido* **and** *desconocido* es *desconocido*.
- **or**. El resultado de *cierto* **or** *desconocido* es *cierto*, *falso* **or** *desconocido* es *desconocido*, mientras que *desconocido* **or** *desconocido* es *desconocido*.
- **not**. El resultado de **not** *desconocido* es *desconocido*.

El resultado de una instrucción SQL de la forma:

$$\textbf{select} \ldots \textbf{from } R_1, \cdots, R_n \textbf{ where } P$$

contiene (proyecciones de) tuplas de $R_1 \times \cdots \times R_n$ para las que el predicado P se evalúa como **cierto**. Si el predicado se evalúa como **falso** o **desconocido** para alguna tupla de $R_1 \times \cdots \times R_n$, (la proyección de) esa tupla no se añade al resultado.

SQL también permite comprobar si el resultado de una comparación es desconocido, en lugar de cierto o falso, mediante las cláusulas **is unknown** (es desconocido) e **is not unknown** (no es desconocido).

La existencia de valores nulos también complica el procesamiento de los operadores de agregación. Por ejemplo, supóngase que algunas tuplas en la relación *préstamo* tienen valor nulo para el atributo *importe*. Considérese la siguiente consulta, que calcula el total de los importes de todos los préstamos:

select sum (*importe*)
from *préstamo*

Entre los valores que se van a sumar en la consulta anterior hay valores nulos, puesto que algunas tuplas tienen valor nulo para *importe*. En lugar de decir que la suma total es *nula*, la norma SQL establece que el operador **sum** debe ignorar los valores nulos de los datos de entrada.

En general, las funciones de agregación tratan los valores nulos según la regla siguiente: todas las funciones de agregación excepto **count (*)** ignoran los valores nulos de la colección de datos de entrada. Como consecuencia de ignorar los valores nulos, la colección de valores de entrada puede estar vacía. El cálculo de **count (*)** de una colección vacía se define como 0 y todas las demás operaciones de agregación devuelven un valor nulo cuando se aplican sobre una colección de datos vacía. El efecto de los valores nulos en algunos de los constructores más complicados de SQL puede ser sutil.

En SQL:1999 se introdujo un tipo de datos **Boolean**, que puede tomar los valores **true**, **false** y **unknown**. Las funciones de agregación **some** (algún) y **every** (cada), que significan exactamente lo que se espera de ellas, se pueden aplicar a las colecciones de valores booleanos.

3.7 Subconsultas anidadas

SQL proporciona un mecanismo para anidar subconsultas. Las subconsultas son expresiones **select-from-where** que están anidadas dentro de otra consulta. Una finalidad habitual de las subconsultas es llevar a cabo comprobaciones de pertenencia a conjuntos, hacer comparaciones de conjuntos y determinar cardinalidades de conjuntos. Estos usos se estudian en los apartados siguientes.

3.7.1 Pertenencia a conjuntos

SQL permite comprobar la pertenencia de las tuplas a una relación. La conectiva **in** comprueba la pertenencia a un conjunto, donde el conjunto es la colección de valores resultado de una cláusula **select**. La conectiva **not in** comprueba la no pertenencia a un conjunto.

Como ejemplo considérese de nuevo la consulta "Determinar todos los clientes que tienen tanto un préstamo como una cuenta en el banco". Anteriormente se escribió esta consulta como la intersección de dos conjuntos: el conjunto de los impositores del banco y el conjunto de los prestatarios del banco. Se puede adoptar el enfoque alternativo consistente en determinar todos los titulares de cuentas del banco que son miembros del conjunto de prestatarios. Claramente, esta formulación genera el mismo resultado que la anterior, pero obliga a formular la consulta usando la conectiva de SQL **in**. Se comienza determinando todos los titulares de cuentas con:

(**select** *nombre_cliente*
from *impositor*)

A continuación, se determinan los clientes que son prestatarios del banco y que aparecen en la lista de titulares de cuentas obtenida en la subconsulta anterior. Esto se consigue anidando la subconsulta en un **select** más externo. La consulta resultante es:

select distinct *nombre_cliente*
from *prestatario*
where *nombre_cliente* **in** (**select** *nombre_cliente*
from *impositor*)

Este ejemplo muestra que en SQL es posible escribir la misma consulta de diversas formas. Esta flexibilidad es de gran utilidad, puesto que permite a los usuarios pensar en las consultas del modo que les parezca más natural. Más adelante se verá que en SQL hay una gran cantidad de redundancia.

En el ejemplo anterior se comprobaba la pertenencia a un conjunto en una relación con un solo atributo. En SQL también es posible comprobar la pertenencia a un conjunto en una relación cualquiera. Así, es posible formular la consulta "Determinar todos los clientes que tienen tanto una cuenta como un préstamo en la sucursal de Navacerrada" de otra manera distinta:

select distinct *nombre_cliente*
from *prestatario, préstamo*
where *prestatario.número_préstamo* = *préstamo.número_préstamo* **and**
nombre_sucursal = 'Navacerrada' **and**
(*nombre_sucursal, nombre_cliente*) **in**
(**select** *nombre_sucursal, nombre_cliente*
from *impositor, cuenta*
where *impositor.número_cuenta* = *cuenta.número_cuenta*)

El constructor **not in** se utiliza de manera parecida. Por ejemplo, para encontrar todos los clientes que tienen concedido un préstamo en el banco pero no tienen abierta cuenta, se puede escribir:

```
select distinct nombre_cliente
from prestatario
where nombre_cliente not in (select nombre_cliente
                             from impositor)
```

Los operadores **in** y **not in** también se pueden utilizar sobre conjuntos enumerados. La consulta siguiente selecciona los nombres de los clientes que tienen concedido un préstamo en el banco y cuyos nombres no son ni Santos ni Gómez.

```
select distinct nombre_cliente
from prestatario
where nombre_cliente not in ('Santos', 'Gómez')
```

3.7.2 Comparación de conjuntos

Como ejemplo de la capacidad de comparar conjuntos de las consultas anidadas, considérese la consulta "Determinar el nombre de todas las sucursales que poseen activos mayores que, al menos, una sucursal de Arganzuela". En el Apartado 3.3.5 se formulaba esta consulta del modo siguiente:

```
select distinct T.nombre_sucursal
from sucursal as T, sucursal as S
where T.activos > S.activos and S.ciudad_sucursal = 'Arganzuela'
```

SQL ofrece, sin embargo, un estilo alternativo de formular la consulta anterior. La expresión: "mayor que, al menos, una" se representa en SQL por $>$ **some** Este constructor permite volver a escribir la consulta de forma más parecida a la formulación de la consulta en lenguaje natural.

```
select nombre_sucursal
from sucursal
where activos > some (select activos
                      from sucursal
                      where ciudad_sucursal = 'Arganzuela')
```

La subconsulta

```
(select activos
 from sucursal
 where ciudad_sucursal = 'Arganzuela')
```

genera el conjunto de todos los valores de activos para todas las sucursales situadas en Arganzuela. La comparación $>$ **some** de la cláusula **where** de la cláusula **select** más externa es cierta si el valor del atributo *activos* de la tupla es mayor que, al menos, un miembro del conjunto de todos los valores de activos de las sucursales de Arganzuela.

SQL también permite realizar las comparaciones $<$ **some**, $<=$ **some**, $>=$ **some**, $=$ **some** y $<>$ **some**. Como ejercicio, compruébese que $=$ **some** es idéntico a **in**, mientras que $<>$ **some** *no* es lo mismo que **not in**. En SQL, la palabra clave **any** es sinónimo de **some**. Las primeras versiones de SQL sólo admitían **any**. Versiones posteriores añadieron la alternativa **some** para evitar la ambigüedad lingüística de la palabra inglesa *any* en su idioma original.

Ahora se va a modificar ligeramente la consulta. Se trata de determinar el nombre de todas las sucursales que tienen activos superiores al de todas las sucursales de Arganzuela. El constructor $>$ **all** se corresponde con la expresión "superiores al de todas". Usando este constructor se puede escribir la consulta del modo siguiente:

```
select nombre_sucursal
from sucursal
where activos > all (select activos
                     from sucursal
                     where ciudad_sucursal = 'Arganzuela')
```

Al igual que con **some**, SQL también permite las comparaciones $<$ **all**, $<=$ **all**, $>=$ **all**, $=$ **all** y $<>$ **all**. Como ejercicio, compruébese que $<>$ **all** es lo mismo que **not in**.

Como ejemplo adicional de comparación de conjuntos, considérese la consulta "Determinar la sucursal que tiene el saldo medio máximo". En SQL las funciones de agregación no se pueden componer. Por tanto, no se puede utilizar **max** (**avg** (...)). En vez de eso, se puede seguir la siguiente estrategia: se comienza por escribir una consulta que determine todos los saldos medios y luego se anida como subconsulta de una consulta mayor que determina las sucursales en las que el saldo medio es mayor o igual que todos los saldos medios:

```
select nombre_sucursal
from cuenta
group by nombre_sucursal
having avg (saldo) >= all (select avg (saldo)
                           from cuenta
                           group by nombre_sucursal)
```

3.7.3 Comprobación de relaciones vacías

SQL incluye la posibilidad de comprobar si las subconsultas tienen alguna tupla en su resultado. El constructor **exists** devuelve el valor **true** si su argumento subconsulta no resulta vacía. Mediante el constructor **exists** se puede formular la consulta "Determinar todos los clientes que tienen tanto una cuenta abierta como un préstamo concedido en el banco" de otra forma más:

```
select nombre_cliente
from prestatario
where exists (select *
              from impositor
              where impositor.nombre_cliente = prestatario.nombre_cliente)
```

Mediante el constructor **not exists** se puede comprobar la inexistencia de tuplas en el resultado de las subconsultas. Se puede utilizar el constructor **not exists** para simular la operación de continencia de conjuntos (es decir, superconjuntos). Se puede escribir "la relación A contiene a la relación B" como "**not exists** (B **except** A)" (aunque no forma parte de las normas SQL-92 ni SQL:1999 el operador **contains** aparecía en algunos de los primeros sistemas relacionales). Para ilustrar el operador **not exists**, considérese otra vez la consulta "Determinar todos los clientes que tienen una cuenta en todas las sucursales de Arganzuela". Hay que comprobar para cada cliente si el conjunto de todas las sucursales en las que dicho cliente tiene cuenta contiene al conjunto de todas las sucursales de Arganzuela. Mediante el constructor **except** se puede formular la consulta del modo siguiente:

```
select distinct S.nombre_cliente
from impositor as S
where not exists ((select nombre_sucursal
                   from sucursal
                   where ciudad_sucursal = 'Arganzuela')
                  except
                   (select R.nombre_sucursal
                    from impositor as T, cuenta as R
                    where T.número_cuenta = R.número_cuenta and
                          S.nombre_cliente = T.nombre_cliente))
```

En este ejemplo, la subconsulta

```
(select nombre_sucursal
 from sucursal
 where ciudad_sucursal = 'Arganzuela')
```

obtiene todas las sucursales de Arganzuela. La subconsulta

```
(select R.nombre_sucursal
 from impositor as T, cuenta as R
 where T.número_cuenta = R.número_cuenta and
       S.nombre_cliente = T.nombre_cliente)
```

obtiene todas las sucursales en las cuales el cliente *S.nombre_cliente* tiene cuenta. Por tanto, el **select** más externo toma cada cliente y comprueba si el conjunto de todas las sucursales en las que dicho cliente tiene cuenta contiene al conjunto de todas las sucursales de Arganzuela.

En las consultas que contienen subconsultas se aplica una regla de visibilidad para las variables tupla. En las subconsultas, de acuerdo con esta regla, sólo se pueden usar variables tupla definidas en la propia subconsulta o en cualquier consulta que la contenga. Si una variable tupla está definida tanto localmente en una subconsulta como globalmente en una consulta que contiene a la subconsulta, se aplica la definición local. Esta regla es análoga a las reglas de visibilidad utilizadas habitualmente para las variables en los lenguajes de programación.

3.7.4 Comprobación de la ausencia de tuplas duplicadas

SQL incluye la posibilidad de comprobar si las subconsultas tienen tuplas duplicadas en su resultado. El constructor **unique** devuelve el valor **true** si la subconsulta que se le pasa como argumento no contiene tuplas duplicadas. Mediante el constructor **unique** se puede formular la consulta "Determinar todos los clientes que tienen, a lo sumo, una cuenta en la sucursal de Navacerrada" del siguiente modo:

```
select T.nombre_cliente
from impositor as T
where unique (select R.nombre_cliente
              from cuenta, impositor as R
              where T.nombre_cliente = R.nombre_cliente and
                    R.número_cuenta = cuenta.número_cuenta and
                    cuenta.nombre_sucursal = 'Navacerrada')
```

La existencia de tuplas duplicadas en una subconsulta se puede comprobar mediante el constructor **not unique**. Para ilustrar este constructor, considérese la consulta "Determinar todos los clientes que tienen, al menos, dos cuentas en la sucursal de Navacerrada", que se puede formular:

```
select distinct T.nombre_cliente
from impositor T
where not unique (select R.nombre_cliente
                  from cuenta, impositor as R
                  where T.nombre_cliente = R.nombre_cliente and
                        R.número_cuenta = cuenta.número_cuenta and
                        cuenta.nombre_sucursal = 'Navacerrada')
```

Formalmente, la evaluación de **unique** sobre una relación se define como falsa si y sólo si la relación contiene dos tuplas t_1 y t_2 tales que $t_1 = t_2$. Como la comprobación $t_1 = t_2$ es falsa si algún campo de t_1 o de t_2 es nulo, es posible que el resultado de **unique** sea cierto aunque haya varias copias de una misma tupla, siempre que, al menos, uno de los atributos de la tupla sea nulo.

3.8 Consultas complejas

Las consultas complejas a menudo resultan difíciles o imposibles de escribir como un único bloque de SQL o como unión, intersección o diferencia de bloques de SQL (cada bloque de SQL consta de una sola instrucción **select-from-where**, posiblemente con cláusulas **group by** y **having**). En este apartado se estudian dos formas de componer varios bloques de SQL para expresar consultas complejas: las relaciones derivadas y la cláusula **with**.

3.8.1 Relaciones derivadas

SQL permite el uso de expresiones de subconsulta en las cláusulas **from**. Si se utiliza una expresión de este tipo, hay que proporcionar un nombre a la relación resultado y será posible renombrar los atributos mediante la cláusula **as**. Por ejemplo, considérese la subconsulta:

(**select** *nombre_sucursal*, **avg** (*saldo*)
from *cuenta*
group by *nombre_sucursal*)
as *media_sucursal* (*nombre_sucursal*, *saldo_medio*)

Esta subconsulta genera una relación consistente en los nombres de todas las sucursales y sus correspondientes saldos de cuenta medios. El resultado de la subconsulta recibe el nombre de *media_sucursal*, con los atributos *nombre_sucursal* y *saldo_medio.*

Para ilustrar el uso de una expresión de subconsulta en la cláusula **from**, considérese la consulta "Determinar el saldo medio de las cuentas de las sucursales en las que el saldo medio de las cuentas sea superior a 1.200 €". En el Apartado 3.5 se formulaba esta consulta utilizando la cláusula **having**. Ahora se puede reescribir sin utilizar esa cláusula, de la siguiente forma:

select *nombre_sucursal*, *avg_saldo*
from (**select** *nombre_sucursal*, **avg** (*saldo*)
from *cuenta*
group by *nombre_sucursal*)
as *media_sucursal* (*nombre_sucursal*, *saldo_medio*)
where *saldo_medio* $>$ 1200

Obsérvese que no hace falta utilizar la cláusula **having**, ya que la subconsulta de la cláusula **from** calcula el saldo medio y su resultado se denomina *media_sucursal*; los atributos de *media_sucursal* se pueden utilizar directamente en la cláusula **where**.

Como ejemplo adicional supóngase que se desea determinar el saldo total máximo de las sucursales. La cláusula **having** no sirve en este caso, pero esta consulta se puede escribir fácilmente usando una subconsulta en la cláusula **from** como se muestra a continuación:

select max(*saldo_total*)
from (**select** *nombre_sucursal*, **sum**(*saldo*)
from *cuenta*
group by *nombre_sucursal*) **as** *total_sucursal* (*nombre_sucursal*, *saldo_total*)

3.8.2 La cláusula with

Las consultas complicadas son mucho más fáciles de formular y de entender si se estructuran descomponiéndolas en vistas más simples que luego se combinan, igual que se estructuran los programas descomponiendo sus tareas en procedimientos. Sin embargo, a diferencia de la definición de procedimientos, las cláusulas **create view** crean definiciones de vistas en la base de datos y esa definición de vista sigue en la base de datos hasta que se ejecuta una orden **drop view** *nombre_vista*.

La cláusula **with** proporciona una forma de definir vistas temporales cuya definición sólo está disponible para la consulta en la que aparece la cláusula. Considérese la siguiente consulta, que selecciona cuentas con el saldo máximo; si hay muchas cuentas con el mismo saldo máximo, se seleccionan todas.

```
with saldo_máximo (valor) as
    select max(saldo)
    from cuenta
select número_cuenta
from cuenta, saldo_máximo
where cuenta.saldo = saldo_máximo.valor
```

La cláusula **with**, introducida en SQL:1999 actualmente sólo está soportada en algunas bases de datos.

La consulta anterior se podría haber escrito usando una subconsulta anidada, bien en la cláusula **from**, bien en la **where**. Sin embargo, el uso de subconsultas anidadas hace que la consulta sea más difícil de leer y de entender. La cláusula **with** hace que la lógica de la consulta sea más clara; también permite utilizar definiciones de vistas en varias partes de la consulta.

Por ejemplo, supóngase que se desea determinar todas las sucursales donde el depósito total de las cuentas es mayor que la media de los depósitos totales de las cuentas de todas las sucursales. Se puede escribir la consulta usando la cláusula **with** como se muestra a continuación.

```
with total_sucursales (nombre_sucursal, valor) as
    select nombre_sucursal, sum(saldo)
    from cuenta
    group by nombre_sucursal
with media_total_sucursales(valor) as
    select avg(valor)
    from total_sucursales
select nombre_sucursal
from total_sucursales, media_total_sucursales
where total_sucursales.valor >= media_total_sucursales.valor
```

Por supuesto, se puede crear una consulta equivalente sin la cláusula **with**, pero sería más complicada y difícil de entender. Como ejercicio se puede escribir la consulta equivalente.

3.9 Vistas

Hasta este momento los ejemplos se han limitado a operar en el nivel de los modelos lógicos. Es decir, se ha dado por supuesto que las relaciones facilitadas son las relaciones reales almacenadas en la base de datos.

No resulta deseable que todos los usuarios vean el modelo lógico completo. Las consideraciones de seguridad pueden exigir que se oculten ciertos datos a los usuarios. Considérese una persona que necesita saber el número de préstamo y el nombre de la sucursal de un cliente, pero no necesita ver el importe de ese préstamo. Esa persona debería ver una relación descrita (módulo renombramiento de atributos) en SQL mediante

```
select nombre_cliente, prestatario.número_préstamo, nombre_sucursal
from prestatario, préstamo
where prestatario.número_préstamo = préstamo.número_préstamo
```

Aparte de las consideraciones de seguridad, puede que se desee crear un conjunto personalizado de relaciones que se adapte mejor a la intuición de un usuario determinado que el modelo lógico. Puede que a un usuario del departamento de publicidad, por ejemplo, le guste ver una relación que consista en los clientes que tienen o bien cuenta abierta o bien préstamo concedido en el banco y las sucursales con las que trabajan. La relación que se crearía para ese empleado es la siguiente:

```
(select nombre_sucursal, nombre_cliente
from impositor, cuenta
where impositor.número_cuenta = cuenta.número_cuenta)
 union
(select nombre_sucursal, nombre_cliente
from prestatario, préstamo
where prestatario.número_préstamo = préstamo.número_préstamo)
```

Las relaciones que no forman parte del modelo lógico pero se hacen visibles a los usuarios como relaciones virtuales se denominan **vistas**. Es posible definir un gran número de vistas de cualquier conjunto dado de relaciones reales.

3.9.1 Definición de vistas

Las vistas en SQL se definen mediante la instrucción **create view**. Para definir una vista hay que darle un nombre e indicar la consulta que la va a calcular. La forma de la instrucción **create view** es

create view v **as** <expresión de consulta>

donde <expresión de consulta> es cualquier expresión legal de consulta. El nombre de la vista se representa mediante v.

A modo de ejemplo, considérese la vista consistente en las sucursales y sus clientes. Supóngase que se desea que esta vista se denomine *todos_los_clientes*. Esta vista se define de la manera siguiente:

```
create view todos_los_clientes as
    (select nombre_sucursal, nombre_cliente
     from impositor, cuenta
     where impositor.número_cuenta = cuenta.número_cuenta)
   union
    (select nombre_sucursal, nombre_cliente
     from prestatario, préstamo
     where prestatario.número_préstamo = préstamo.número_préstamo)
```

Una vez definida la vista se puede utilizar su nombre para hacer referencia a la relación virtual que genera. Utilizando la vista *todos_los_clientes* se puede determinar el nombre de todos los clientes de la sucursal de Navacerrada escribiendo

```
select nombre_cliente
from todos_los_clientes
where nombre_sucursal = 'Navacerrada'
```

Los nombres de las vistas pueden aparecer en cualquier lugar en el que puedan hacerlo los nombres de las relaciones, siempre y cuando no se ejecuten sobre las vistas operaciones de actualización. El asunto de las operaciones de actualización de las vistas se estudia en el Apartado 3.10.4.

Los nombres de los atributos de las vistas pueden especificarse de manera explícita de la manera siguiente:

```
create view total_préstamos_sucursal(nombre_sucursal, total_préstamos) as
select nombre_sucursal, sum(importe)
from préstamo
group by nombre_sucursal
```

La vista anterior da la suma del importe de todos los créditos de cada sucursal. Dado que la expresión **sum**(*importe*) no tiene nombre, el nombre del atributo se especifica de manera explícita en la definición de la vista.

De manera intuitiva, el conjunto de tuplas de la relación de vistas es el resultado de la evaluación de la expresión de consulta que define en ese momento la vista. Por tanto, si las relaciones de vistas se calculan y se guardan, pueden quedar desfasadas si las relaciones usadas para definirlas se modifican. Para evitarlo, las vistas suelen implementarse de la manera siguiente. Cuando se define una vista, el sistema de la base de datos guarda la definición de la vista, en vez del resultado de la evaluación de la expresión del álgebra relacional que la define. Siempre que aparece una relación de vistas en una consulta, se sustituye por la expresión de consulta almacenada. Por tanto, la relación de vistas se vuelve a calcular siempre que se evalúa la consulta.

Algunos sistemas de bases de datos permiten que se guarden las relaciones de vistas, pero se aseguran de que, si las relaciones reales utilizadas en la definición de la vista cambian, la vista se mantenga actualizada. Estas vistas se denominan **vistas materializadas**. El proceso de mantener actualizada la vista se denomina **mantenimiento de vistas**, y se trata en el Apartado 14.5. Las aplicaciones en las que se utiliza frecuentemente una misma vista aprovechan las vistas materializadas, al igual que las aplicaciones que exigen una respuesta rápida a ciertas consultas basadas en las vistas. Por supuesto, las ventajas para las consultas derivadas de la materialización de las vistas deben sopesarse frente a los costes de almacenamiento y la sobrecarga añadida de las actualizaciones.

3.9.2 Vistas definidas en función de otras

En el Apartado 3.9.1 se mencionó que las relaciones de vistas pueden aparecer en cualquier lugar en que puedan hacerlo los nombres de las relaciones, salvo las restricciones al uso de las vistas en expresiones de actualización. Por tanto, se pueden utilizar vistas en las expresiones que definen otras vistas. Por ejemplo, se puede definir la vista *cliente_navacerrada* de la manera siguiente:

create view *cliente_navacerrada* **as**
select *nombre_cliente*
from *todos_los_clientes*
where *nombre_sucursal* = 'Navacerrada'

donde *todos_los_clientes* es, a su vez, una relación de vistas.

La **expansión de vistas** es una manera de definir el significado de las vistas definidas en términos de otras. El procedimiento asume que las definiciones de vistas no son **recursivas**; es decir, no se utiliza ninguna vista en su propia definición, ni directa ni indirectamente mediante otras definiciones de vistas. Por ejemplo, si $v1$ se utiliza en la definición de $v2$, $v2$ se utiliza en la definición de $v3$ y $v3$ se usa en la definición de $v1$, entonces $v1$, $v2$ y $v3$ son recursivas. Las definiciones recursivas de las vistas resultan útiles en algunos casos, y se volverá a ellas en el contexto del lenguaje Datalog, en el Apartado 5.4.

Supóngase una vista v_1 definida mediante una expresión e_1 que puede contener a su vez relaciones de vistas. Las relaciones de vistas representan a las expresiones que definen las vistas y, por tanto, se pueden sustituir por las expresiones que las definen. Si se modifica una expresión sustituyendo una relación de vistas por su definición, la expresión resultante puede seguir conteniendo otras relaciones de vistas. Por tanto, la expansión de vistas de una expresión repite la etapa de sustitución de la manera siguiente:

repeat
  Buscar todas las relaciones de vistas v_i de e_1
  Sustituir la relación de vistas v_i por la expresión que define v_i
until no queden más relaciones de vistas en e_1

Mientras las definiciones de las vistas no sean recursivas, el bucle concluirá. Por tanto, una expresión e que contenga relaciones de vistas puede entenderse como la expresión resultante de la expansión de vistas de e, que no contiene ninguna relación de vistas.

Como ilustración de la expansión de vistas considérese la expresión siguiente:

select *
from *cliente_navacerrada*
where *nombre_cliente* = 'Martín'

El procedimiento de expansión de vistas produce inicialmente

```
select *
from  (select nombre_cliente
       from todos_los_clientes
       where nombre_sucursal = 'Navacerrada')
where nombre_cliente = 'Martín'
```

luego produce

```
select *
from (select nombre_cliente
      from ((select nombre_sucursal, nombre_cliente
             from impositor, cuenta
             where impositor.número_cuenta = cuenta.número_cuenta)
             union
             (select nombre_sucursal, nombre_cliente
             from prestatario, préstamo
             where prestatario.número_préstamo = préstamo.número_préstamo))
      where nombre_sucursal = 'Navacerrada')
where nombre_cliente = 'Martín'
```

En este momento no hay más usos de las relaciones de vistas y concluye la expansión de vistas.

3.10 Modificación de la base de datos

Hasta ahora se ha estudiado la extracción de información de las bases de datos. A continuación se mostrará cómo añadir, eliminar y modificar información utilizando SQL.

3.10.1 Borrado

Las solicitudes de borrado se expresan casi igual que las consultas. Sólo se pueden borrar tuplas completas; no se pueden borrar sólo valores de atributos concretos. SQL expresa los borrados mediante:

```
delete from r
where P
```

donde P representa un predicado y r representa una relación. La declaración **delete** busca primero todas las tuplas t en r para las que $P(t)$ es cierto y a continuación las borra de r. La cláusula **where** se puede omitir, en cuyo caso se borran todas las tuplas de r.

Obsérvese que cada comando **delete** sólo opera sobre una relación. Si se desea borrar tuplas de varias relaciones hay que utilizar una orden **delete** por cada relación. El predicado de la cláusula **where** puede ser tan complicado como la cláusula **where** de cualquier orden **select**. En el otro extremo, la cláusula **where** puede estar vacía. La consulta

```
delete from préstamo
```

borra todas las tuplas de la relación *préstamo* (los sistemas bien diseñados exigen una confirmación del usuario antes de ejecutar una petición tan devastadora).

A continuación se muestra una serie de ejemplos de peticiones de borrado en SQL:

- Borrar todas las tuplas de cuentas de la sucursal de Navacerrada.

```
delete from cuenta
where nombre_sucursal = 'Navacerrada'
```

- Borrar todos los préstamos con importe comprendido entre 1.300 € y 1.500 €.

```
delete from préstamo
where importe between 1300 and 1500
```

- Borrar todas las tuplas de cuenta de todas las sucursales de Arganzuela.

```
delete from cuenta
where nombre_sucursal in (select nombre_sucursal
                          from sucursal
                          where ciudad_sucursal = 'Arganzuela')
```

  Esta solicitud de borrado busca primero todas las sucursales de Arganzuela y luego borra todas las tuplas *cuenta* correspondientes a esas sucursales.

Obsérvese que, si bien sólo se pueden borrar tuplas de una única relación a la vez, se puede hacer referencia a cualquier número de relaciones en una expresión **select-from-where** anidada en la cláusula **where** de una orden **delete**. La petición **delete** puede contener un **select** anidado que haga referencia a la relación cuyas tuplas se van a borrar. Por ejemplo, supóngase que se desea borrar los registros de todas las cuentas con saldos inferiores a la media del banco. Se puede escribir:

```
delete from cuenta
where saldo < (select avg (saldo)
               from cuenta)
```

El comando **delete** comprueba primero cada tupla de la relación *cuenta* para ver si la cuenta tiene un saldo inferior a la media del banco. Luego se borran todas las tuplas que cumplen la condición; es decir, representan una cuenta con un saldo inferior a la media. Es importante realizar todas las comprobaciones antes de llevar a cabo ningún borrado; si se borra alguna tupla antes de comprobar las demás, el saldo medio puede cambiar ¡y el resultado final del borrado dependería del orden en que se procesaran las tuplas!

3.10.2 Inserción

Para insertar datos en una relación, se especifica la tupla que se desea insertar o se formula una consulta cuyo resultado sea el conjunto de tuplas que se desea insertar. Obviamente, los valores de los atributos de las tuplas que se inserten deben pertenecer al dominio de los atributos. De igual modo, las tuplas insertadas deben ser de la aridad correcta.

La instrucción **insert** más sencilla es una solicitud de inserción de una tupla. Supóngase que se desea insertar el hecho de que hay una cuenta C-9732 en la sucursal de Navacerrada y que tiene un saldo de 1.200 €. Hay que escribir

```
insert into cuenta
    values ('C-9732', 'Navacerrada', 1200)
```

En este ejemplo los valores se especifican en el mismo orden en que aparecen los atributos correspondientes en el esquema de la relación. Para beneficio de los usuarios, que puede que no recuerden el orden de los atributos, SQL permite que los atributos se especifiquen en la cláusula **insert**. Por ejemplo, las siguientes instrucciones **insert** tienen una función idéntica a la anterior:

```
insert into cuenta (número_cuenta, nombre_sucursal, saldo)
    values ('C-9732', 'Navacerrada', 1200)

insert into cuenta (nombre_sucursal, número_cuenta, saldo)
    values ('Navacerrada', 'C-9732', 1200)
```

De forma más general es posible que se desee insertar las tuplas que resultan de una consulta. Supóngase que se les desea ofrecer, como regalo, a todos los clientes titulares de préstamos de la sucursal de Navacerrada una cuenta de ahorro con 200 € por cada préstamo que tengan concedido. Así, se escribe:

```
insert into cuenta
    select número_préstamo, nombre_sucursal, 200
    from préstamo
    where nombre_sucursal = 'Navacerrada'
```

En lugar de especificar una tupla, como se hizo en los primeros ejemplos de este apartado, se utiliza una instrucción **select** para especificar un conjunto de tuplas. SQL evalúa en primer lugar la instrucción **select**, lo que produce un conjunto de tuplas que se inserta a continuación en la relación *cuenta*. Cada tupla tiene un un *número_préstamo* (que sirve como número de cuenta para la nueva cuenta), un *nombre_sucursal* (Navacerrada) y un saldo inicial de la cuenta (200 €).

También hay que añadir tuplas a la relación *impositor*; para ello hay que escribir

```
insert into impositor
    select nombre_cliente, número_préstamo
    from prestatario, préstamo
    where prestatario.número_préstamo = préstamo.número_préstamo and
        nombre_sucursal = 'Navacerrada'
```

Esta consulta inserta en la relación *impositor* una tupla (*nombre_cliente, número_préstamo*) por cada *nombre_cliente* que tenga concedido un préstamo en la sucursal de Navacerrada, con número de préstamo *número_préstamo*.

Es importante que la evaluación de la instrucción **select** finalice completamente antes de llevar a cabo ninguna inserción. Si se realizase alguna inserción mientras se evalúa la instrucción **select**, una petición del tipo

```
insert into cuenta
    select *
    from cuenta
```

¡podría insertar un número infinito de tuplas! La solicitud insertaría la primera tupla de nuevo en *cuenta*, creando así una segunda copia de la tupla. Como esta segunda copia ya forma parte de *cuenta*, la instrucción **select** puede encontrarla, y se insertaría una tercera copia en *cuenta*. La instrucción **select** puede entonces encontrar esta tercera copia e insertar una cuarta copia, y así indefinidamente. Al evaluar por completo la instrucción **select** antes de realizar ninguna inserción se evita este tipo de problemas.

La discusión de la instrucción **insert** sólo ha considerado ejemplos en los que se especificaba un valor para cada atributo de las tuplas insertadas. Es posible, como se vio en el Capítulo 2, dar valores únicamente a algunos de los atributos del esquema para las tuplas insertadas. A los atributos restantes se les asigna un valor nulo, que se denota por *null*. Considérese la petición

```
insert into cuenta
    values ('C-401', null, 1200)
```

Se sabe que la cuenta C-401 tiene un saldo de 1.200 €, pero no se conoce el nombre de la sucursal. Considérese la consulta

```
select número_cuenta
from cuenta
where nombre_sucursal = 'Navacerrada'
```

Como el nombre de la sucursal en que se ha abierto la cuenta C-401 es desconocido, no se puede determinar si es igual a "Navacerrada".

Se puede prohibir la inserción de valores nulos en atributos concretos utilizando el LDD de SQL, que se estudia en el Apartado 3.2.

La mayor parte de los productos de bases de datos tienen utilidades especiales de "carga masiva" para insertar en las relaciones grandes conjuntos de tuplas. Estas utilidades permiten leer datos de archivos de texto con formato y pueden ejecutarse mucho más rápido que la secuencia equivalente de instrucciones **insert**.

3.10.3 Actualizaciones

En determinadas situaciones puede ser deseable modificar un valor dentro de una tupla sin cambiar *todos* los valores de la misma. Para este tipo de situaciones se puede utilizar la instrucción **update**. Al igual que ocurre con **insert** y **delete**, se pueden elegir las tuplas que se van a actualizar mediante una consulta.

Supóngase que se va a realizar el pago anual de intereses y que hay que incrementar todos los saldos en un 5 por ciento. Hay que escribir

update *cuenta*
set *saldo* = *saldo* * 1.05

Esta instrucción de actualización se aplica una vez a cada tupla de la relación *cuenta*.

Si sólo se paga el interés a las cuentas con un saldo de 1.000 € o superior, se puede escribir

update *cuenta*
set *saldo* = *saldo* * 1.05
where *saldo* >= 1000

En general, la cláusula **where** de la instrucción **update** puede contener cualquier constructor permitido en la cláusula **where** de la instrucción **select** (incluyendo instrucciones **select** anidadas). Al igual que ocurre con **insert** y con **delete**, un **select** anidado en una instrucción **update** puede hacer referencia a la relación que se esté actualizando. Al igual que antes, SQL primero comprueba todas las tuplas de la relación para determinar si se deben actualizar y después realiza la actualización. Por ejemplo, se puede escribir la solicitud "Pagar un interés del 5 por ciento a las cuentas cuyo saldo sea mayor que la media" como sigue:

update *cuenta*
set *saldo* = *saldo* * 1.05
where *saldo* > (**select avg** (*saldo*)
from *cuenta*)

Supóngase que las cuentas con saldos superiores a 10.000 € reciben un 6 por ciento de interés, mientras que las demás reciben un 5 por ciento. Se pueden escribir dos instrucciones de actualización:

update *cuenta*
set *saldo* = *saldo* * 1.06
where *saldo* > 10000

update *cuenta*
set *saldo* = *saldo* * 1.05
where *saldo* <= 10000

Obsérvese que, como se vio en el Capítulo 2, el orden de las dos instrucciones **update** es importante. Si se cambiara el orden de las dos instrucciones, una cuenta con un saldo igual o muy poco inferior a 10.000 € recibiría un 11,3 por ciento de interés.

SQL ofrece un constructor **case** que se puede utilizar para llevar a cabo las dos instrucciones de actualización anteriores en una única instrucción **update**, evitando el problema del orden de actualización.

update *cuenta*
set *saldo* = **case**
when *saldo* <= 10000 **then** *saldo* * 1.05
else *saldo* * 1.06
end

La forma general de la instrucción **case** es la siguiente:

case
  when $pred_1$ **then** $result_1$
  when $pred_2$ **then** $result_2$
  ...
  when $pred_n$ **then** $result_n$
  else $result_0$
end

La operación devuelve $result_i$, donde i es el primero de $pred_1, pred_2, \ldots, pred_n$ que se satisface; si ninguno de ellos se satisface, la operación devuelve $result_0$. Las instrucciones **case** se pueden usar en cualquier lugar donde se espere un valor.

3.10.4 Actualización de vistas

Aunque las vistas resultan una herramienta útil para las consultas, presentan serios problemas si se expresa con ellas actualizaciones, inserciones o borrados. La dificultad radica en que las modificaciones de la base de datos expresadas en términos de vistas deben traducirse en modificaciones de las relaciones reales del modelo lógico de la base de datos.

Para ilustrar el problema, considérese un empleado que necesita ver todos los datos de préstamos de la relación *préstamo*, excepto *importe_préstamo*. Sea *sucursal_préstamo* la vista ofrecida al empleado. Esta vista se define como

create view *sucursal_préstamo* **as**
select *número_préstamo, nombre_sucursal*
from *préstamo*

Como se permite que el nombre de la vista aparezca en cualquier lugar en el que pueda aparecer el nombre de una relación, el empleado puede escribir

insert into *sucursal_préstamo*
  values ('P-37', 'Navacerrada')

Esta inserción debe representarse mediante una inserción en la relación *préstamo*, puesto que *préstamo* es la relación real a partir de la cual el sistema de bases de datos construye la vista *sucursal_préstamo*. Sin embargo, para insertar una tupla en *préstamo* hay que tener algún valor para *importe*. Hay dos enfoques razonables para tratar esta inserción:

- Rechazar la inserción y devolver al usuario un mensaje de error.
- Insertar la tupla (P-37, "Navacerrada", *null*) en la relación *préstamo*.

Otro problema con la modificación de la base de datos mediante vistas surge con vistas como

create view *info_préstamo* **as**
select *nombre_cliente, importe*
from *prestatario, préstamo*
where *prestatario.número_préstamo* = *préstamo.número_préstamo*

Esta vista muestra el importe del préstamo para cada préstamo que tenga concedido cualquier cliente del banco. Considérese la siguiente inserción mediante esta vista:

insert into *info_préstamo*
  values ('González', 1900)

El único método posible de inserción de tuplas en las relaciones *prestatario* y *préstamo* es insertar ("González", *null*) en *prestatario* y (*null*, *null*, 1900) en *préstamo*. Así se obtendrían las relaciones que pueden verse en la Figura 3.3. No obstante, esta actualización no tiene el efecto deseado, ya que la relación

préstamo

número_préstamo	*nombre_sucursal*	*importe*
P-11	Collado Mediano	900
P-14	Centro	1.500
P-15	Navacerrada	1.500
P-16	Navacerrada	1.300
P-17	Centro	1.000
P-23	Moralzarzal	2.000
P-93	Becerril	500
nulo	*nulo*	1.900

prestatario

nombre_cliente	*número_préstamo*
Fernández	P-16
Gómez	P-11
Gómez	P-23
López	P-15
Pérez	P-93
Santos	P-17
Sotoca	P-14
Valdivieso	P-17
González	*nulo*

Figura 3.3 Las tuplas insertadas en *préstamo* y *prestatario*.

de vistas *info_préstamo* sigue *sin* incluir la tupla ("González", 1900). Por tanto, no hay manera de actualizar las relaciones *prestatario* y *préstamo* mediante el uso de valores nulos para obtener la actualización deseada de *info_préstamo*.

Debido a problemas como éstos no se suelen permitir las actualizaciones de las relaciones de vistas, salvo en casos concretos. Los diferentes sistemas de bases de datos especifican condiciones diferentes para permitir las actualizaciones de las relaciones de vistas y es preciso consultar el manual de cada sistema para conocer los detalles. El problema general de la modificación de las bases de datos mediante las vistas ha sido objeto de amplia investigación, y las notas bibliográficas ofrecen indicaciones de parte de esa investigación.

En general se dice que una vista de SQL es **actualizable** (es decir, se le pueden aplicar inserciones, actualizaciones y borrados) si se cumplen todas las condiciones siguientes:

- La cláusula **from** sólo tiene una relación de base de datos.
- La cláusula **select** sólo contiene nombres de atributos de la relación y no tiene ninguna expresión, valor agregado ni especificación **distinct**.
- Cualquier atributo que no aparezca en la cláusula **select** puede definirse como nulo.
- La consulta no tiene cláusulas **group** ni **having**.

Con estas restricciones, las operaciones **update**, **insert** y **delete** estarían prohibidas en la vista de ejemplo *todos_los_clientes* que se había definido anteriormente.

Supóngase que se define la vista *cuenta_centro* de la manera siguiente:

create view *cuenta_centro* **as**
select *número_cuenta*, *nombre_sucursal*, *saldo*
from *cuenta*
where *nombre_sucursal* = 'Centro'

Esta vista es actualizable, ya que cumple las condiciones que se han fijado anteriormente.

Pese a las condiciones para la actualización, sigue existiendo el problema siguiente. Supóngase que un usuario intenta insertar la tupla ('C-999', 'Navacerrada', 1000) en la vista *cuenta_centro*. Esta tupla puede insertarse en la relación *cuenta*, pero no aparecerá en la vista *cuenta_centro*, ya que no cumple la selección impuesta por la vista.

De manera predeterminada, SQL permitiría que la actualización se llevara a cabo. Sin embargo, pueden definirse vistas con una cláusula **with check option** al final de la definición de la vista; por tanto, si una tupla insertada en la vista no cumple la condición de la cláusula **where** de la vista, el sistema de bases de datos rechaza la inserción. De manera parecida se rechazan las actualizaciones si los valores nuevos no cumplen las condiciones de la cláusula **where**.

SQL:1999 tiene un conjunto de reglas más complejo en cuanto a la posibilidad de ejecutar inserciones, actualizaciones y borrados sobre las vistas, el cual permite las actualizaciones en una clase más amplia de vistas; sin embargo, estas reglas son demasiado complejas para estudiarlas aquí.

3.10.5 Transacciones

Una **transacción** consiste en una secuencia de instrucciones de consulta o de actualización. La norma SQL especifica que una transacción comienza implícitamente cuando se ejecuta una instrucción SQL. Una de las siguientes instrucciones SQL debe finalizar la transacción:

- **Commit work** compromete la transacción actual; es decir, hace que las actualizaciones realizadas por la transacción pasen a ser permanentes en la base de datos. Una vez comprometida la transacción, se inicia de manera automática una nueva transacción.
- **Rollback work** provoca el retroceso de la transacción actual; es decir, deshace todas las actualizaciones realizadas por las instrucciones SQL de la transacción. Por tanto, el estado de la base de datos se restaura al que tenía antes de la ejecución de la primera instrucción de la transacción.

La palabra clave **work** es opcional en ambas instrucciones.

El retroceso de transacciones resulta útil si se detecta alguna condición de error durante la ejecución de la transacción. El compromiso es parecido, en cierto sentido, a guardar los cambios de un documento que se esté editando, mientras que el retroceso es como abandonar la sesión de edición sin guardar los cambios. Una vez que la transacción ha ejecutado **commit work**, sus efectos ya no se pueden deshacer con **rollback work**. El sistema de bases de datos garantiza en caso de fallo (como puede ser un error en una de las instrucciones SQL, una caída de tensión o una caída del sistema) provocar el retroceso de los efectos de la transacción si todavía no se había ejecutado **commit work**. En caso de caída de tensión o caída del sistema, el retroceso ocurre cuando el sistema se reinicia.

Por ejemplo, para transferir dinero de una cuenta a otra hay que actualizar los saldos de dos cuentas. Las dos instrucciones de actualización forman una transacción. Un error mientras una transacción ejecuta alguna de las instrucciones tiene como consecuencia que se deshagan los efectos de las instrucciones anteriores de esa transacción, por lo que la base de datos no se queda en un estado parcialmente actualizado. En el Capítulo 15 se estudian más propiedades de las transacciones.

Si un programa termina sin ejecutar ninguno de estos comandos, las actualizaciones se comprometen o se provoca su retroceso. La norma no especifica cuál de los dos cosas tiene lugar, con lo que la elección depende de la implementación. En muchas implementaciones de SQL cada instrucción de SQL se considera de manera predeterminada como una transacción en sí misma, y se compromete en cuanto se ejecuta. El compromiso automático de cada instrucción de SQL se puede desactivar si hay que ejecutar una transacción que consta de varias instrucciones SQL. La forma de desactivación del compromiso automático depende de cada implementación SQL concreta.

Una alternativa mejor, que forma parte de la norma SQL:1999 (aunque actualmente sólo está soportada por algunas implementaciones de SQL) es permitir que se encierren varias instrucciones SQL entre las palabras clave **begin atomic** ... **end**. Todas las instrucciones entre esas palabras clave forman así una única transacción.

3.11 Reunión de relaciones**

SQL no sólo proporciona el mecanismo básico del producto cartesiano para reunir las tuplas de las relaciones, también ofrece (en SQL-92 y versiones posteriores) otros mecanismos para reunir relaciones, como las reuniones condicionales y las reuniones naturales, así como diversas formas de reuniones externas. Estas operaciones adicionales suelen usarse como expresiones de subconsulta en la cláusula **from**.

número_préstamo	*nombre_sucursal*	*importe*
P-170	Centro	3.000
P-230	Moralzarzal	4.000
P-260	Navacerrada	1.700

préstamo

nombre_cliente	*número_préstamo*
Santos	P-170
Gómez	P-230
López	P-155

prestatario

Figura 3.4 Las relaciones *préstamo* y *prestatario*.

3.11.1 Ejemplos

Se van a ilustrar las diversas operaciones de reunión mediante las relaciones *préstamo* y *prestatario* en la Figura 3.4. Se comenzará con un ejemplo sencillo de reunión interna. La Figura 3.5 muestra el resultado de la expresión

préstamo **inner join** *prestatario* **on** $préstamo.número_préstamo = prestatario.número_préstamo$

La expresión calcula la reunión zeta de las relaciones *préstamo* y *prestatario* con la condición de reunión *préstamo.número_préstamo* = *prestatario.número_préstamo*. Los atributos del resultado son los atributos de la relación del lado izquierdo seguidos por los de la relación del lado derecho.

Obsérvese que el atributo *número_préstamo* aparece dos veces en la figura—la primera aparición se debe a la relación *préstamo* y la segunda a *prestatario*. La norma SQL no exige que los nombres de atributo en resultados como éstos sean únicos. Se debería usar una cláusula **as** para asignar nombres únicos a los atributos de los resultados de las consultas y subconsultas.

La relación resultado de la reunión y sus atributos se renombra usando una cláusula **as**, como se ilustra a continuación:

préstamo **inner join** *prestatario* **on** *préstamo.número_préstamo* = *prestatario.número_préstamo*
as *pp(número_préstamo, sucursal, importe, cliente, número_préstamo_cliente)*

La segunda aparición de *número_préstamo* se ha renombrado como *número_préstamo_cliente*. El orden de los atributos en el resultado de la reunión es importante a la hora de renombrarlos.

A continuación se toma en consideración un ejemplo de la operación reunión externa por la izquierda (**left outer-join**):

préstamo **left outer join** *prestatario* **on** *préstamo.número_préstamo* = *prestatario.número_préstamo*

Es posible calcular la operación reunión externa por la izquierda del modo siguiente. Primero se calcula el resultado de la reunión interna como antes. Luego, para cada tupla t de la relación del lado izquierdo *préstamo* que no coincide con ninguna tupla de la relación del lado derecho *prestatario* en la reunión interior se añade al resultado de la reunión una tupla r: los atributos de la tupla r que se obtienen a partir de la relación del lado izquierdo se rellenan con los valores de la tupla t y el resto de los atributos de r se rellena con valores nulos. La Figura 3.6 muestra la relación resultante. Las tuplas (P-170, Centro, 3.000) y (P-230, Moralzarzal, 4.000) se reúnen con las tuplas de *prestatario* y aparecen en el resultado de la reunión interna y, por tanto, en el resultado de la reunión externa por la izquierda. Por otra parte, la tupla (P-260, Navacerrada, 1.700) no coincide con ninguna tupla de *prestatario* en la reunión interna y, por eso, en el resultado de la reunión externa por la izquierda aparece la tupla (P-260, Navacerrada, 1.700, nulo, nulo).

número_préstamo	*nombre_sucursal*	*importe*	*nombre_cliente*	*número_préstamo*
P-170	Centro	3.000	Santos	P-170
P-230	Moralzarzal	4.000	Gómez	P-230

Figura 3.5 Resultado de *préstamo* **inner join** *prestatario* **on** $préstamo.número_préstamo = prestatario.número_préstamo$.

número_préstamo	*nombre_sucursal*	*importe*	*nombre_cliente*	*número_préstamo*
P-170	Centro	3.000	Santos	P-170
P-230	Moralzarzal	4.000	Gómez	P-230
P-260	Navacerrada	1.700	*nulo*	*nulo*

Figura 3.6 Resultado de *préstamo* **left outer join** *prestatario* **on** $préstamo.número_préstamo = prestatario.número_préstamo$.

Finalmente, se considera un ejemplo de la operación reunión natural (**natural join**).

préstamo **natural inner join** *prestatario*

Esta expresión calcula la reunión natural de las dos relaciones. El único nombre de atributo común a *préstamo* y a *prestatario* es *número_préstamo*. La Figura 3.7 muestra el resultado de la expresión. Este resultado es parecido al de la reunión interna con la condición **on** mostrada en la Figura 3.5 ya que, de hecho, tienen la misma condición de reunión. Sin embargo, el atributo *número_préstamo* sólo aparece una vez en el resultado de la reunión natural, mientras que aparece dos veces en el de la reunión con la condición **on**.

3.11.2 Tipos y condiciones de reunión

En el Apartado 3.11.1 se vieron ejemplos de las operaciones de reunión permitidas en SQL. Las operaciones de reunión toman dos relaciones y devuelven como resultado otra relación. Aunque las expresiones de reunión externa se usan normalmente en la cláusula **from**, se pueden utilizar en cualquier lugar en el que se pueda utilizar una relación.

Cada una de las variantes de las operaciones de reunión en SQL consiste en un *tipo de reunión* y una *condición de reunión*. La condición de reunión define las tuplas de las dos relaciones a casar y los atributos que se incluyen en el resultado de la reunión. El tipo de reunión define la manera de tratar las tuplas de cada relación que no casan con ninguna tupla de la otra relación (de acuerdo con la condición de reunión). La Figura 3.8 muestra algunos de los tipos y condiciones de reunión permitidos. El primer tipo de reunión es la reunión interna y los otros tres son reuniones externas. De las tres condiciones de reunión ya se han visto la reunión **natural** y la condición **on**, y se estudiará la condición **using** más adelante en este apartado.

El uso de condiciones de reunión es obligatorio en las reuniones externas, pero es opcional en las internas (si se omite, se obtiene un producto cartesiano). Sintácticamente, la palabra clave **natural** aparece delante del tipo de reunión, como se mostró anteriormente, mientras que las condiciones **on** y **using** aparecen al final de la expresión de reunión. Las palabras clave **inner** y **outer** son opcionales, ya que su forma permite deducir si la reunión es interna o externa.

número_préstamo	*nombre_sucursal*	*importe*	*nombre_cliente*
P-170	Centro	3.000	Santos
P-230	Moralzarzal	4.000	Gómez

Figura 3.7 Resultado de *préstamo* **natural inner join** *prestatario*.

Tipos de reunión
inner join
left outer join
right outer join
full outer join

Condiciones de reunión
natural
on <predicado>
using $(A_1, A_2, \ldots, A_n)$

Figura 3.8 Tipos y condiciones de reunión.

número_préstamo	*nombre_sucursal*	*importe*	*nombre_cliente*
P-170	Centro	3.000	Santos
P-230	Moralzarzal	4.000	Gómez
P-155	*nulo*	*nulo*	López

Figura 3.9 Resultado de *préstamo* **natural right outer join** *prestatario*.

El significado de la condición de reunión **natural**, en cuanto a las tuplas de las relaciones a casar, es claro. La ordenación de los atributos, dentro del resultado de la reunión natural es el siguiente. Los atributos de reunión (es decir, los atributos comunes a las dos relaciones) aparecen en primer lugar, en el orden en el que aparecen en la relación del lado izquierdo. A continuación se encuentran los demás atributos no comunes de la relación del lado izquierdo y, al final, todos los atributos no comunes de la relación del lado derecho.

La reunión externa por la derecha (**right outer join**) es simétrica a la reunión externa por la izquierda (**left outer join**). Las tuplas de la relación del lado derecho que no casan con ninguna tupla de la relación del lado izquierdo se rellenan con valores nulos y se añaden al resultado de la reunión externa por la derecha.

La siguiente expresión es un ejemplo de combinación de la condición de reunión natural con el tipo de reunión externa por la derecha:

préstamo **natural right outer join** *prestatario*

La Figura 3.9 muestra el resultado de esta expresión. Los atributos del resultado vienen definidos por el tipo de reunión, que es una reunión natural; por tanto, *número_préstamo* sólo aparece una vez. Las dos primeras tuplas del resultado provienen de la reunión natural interna de *préstamo* y *prestatario*. La tupla de la relación del lado derecho (López, P-155) no casa con ninguna tupla de la relación del lado izquierdo *préstamo* en la reunión interna. Así, la tupla (P-155, nulo, nulo, López) aparece en el resultado de la reunión.

La condición de reunión **using**$(A_1, A_2, \ldots, A_n)$ es parecida a la condición de reunión natural, salvo en que los atributos de reunión son los atributos $A_1, A_2, \ldots, A_n$, en lugar de todos los atributos comunes a ambas relaciones. Los atributos $A_1, A_2, \ldots, A_n$ sólo deben ser los comunes a ambas relaciones y sólo aparecen una vez en el resultado de la unión.

La reunión externa completa (**full outer join**) es una combinación de los tipos de reunión externa por la derecha y por la izquierda. Una vez que la operación ha calculado el resultado de la reunión interna, extiende con valores nulos las tuplas de la relación del lado izquierdo que no casan con ninguna tupla de la relación del lado derecho y las añade al resultado. De manera análoga, extiende con valores nulos las tuplas de la relación del lado derecho que no casan con ninguna tupla de la relación del lado izquierdo y las añade al resultado.

Por ejemplo, la Figura 3.10 muestra el resultado de la expresión

préstamo **full outer join** *prestatario* **using** (número_préstamo)

Como ejemplo adicional del uso de la operación reunión externa se puede escribir la consulta: "Determinar todos los clientes que tienen una cuenta abierta pero no tienen ningún préstamo concedido en el banco" como

número_préstamo	*nombre_sucursal*	*importe*	*nombre_cliente*
P-170	Centro	3.000	Santos
P-230	Moralzarzal	4.000	Gómez
P-260	Navacerrada	1.700	*nulo*
P-155	*nulo*	*nulo*	López

Figura 3.10 Resultado de *préstamo* **full outer join** *prestatario* **using** (*número_préstamo*).

```
select i_NC
from (impositor left outer join prestatario
      on impositor.nombre_cliente = prestatario.nombre_cliente)
      as db1 (i_NC, número_cuenta, p_NC, número_préstamo)
where p_NC is null
```

De forma análoga se puede escribir la consulta "Determinar todos los clientes que tienen una cuenta abierta o un préstamo concedido en el banco (pero no las dos cosas)" con el operador de reunión natural externa completa como:

```
select nombre_cliente
from (impositor natural full outer join prestatario)
where número_cuenta is null or número_préstamo is null
```

SQL-92 proporciona también otros dos tipos de reunión, denominados **cross join** (reunión cruzada) y **union join** (reunión de unión). El primero es equivalente a una reunión interna sin condición de reunión; el segundo es equivalente a una reunión externa completa con condición "false", es decir, donde la reunión interna está vacía.

3.12 Resumen

- Los sistemas comerciales de bases de datos no utilizan la concisa y formal álgebra relacional que se trató en Capítulo 2. El ampliamente usado lenguaje SQL, que se ha estudiado en este capítulo, está basado en el álgebra relacional, pero incluye mucho "azúcar sintáctico".
- El lenguaje de definición de datos de SQL se utiliza para crear relaciones con esquemas especificados. El LDD de SQL soporta diferentes tipos de datos, como **date** y **time**. Se pueden ver más detalles sobre el LDD de SQL, en especial sobre el soporte de las restricciones de integridad, en el Apartado. 3.2.
- SQL incluye varios constructores del lenguaje de consultas a la base de datos. Todas las operaciones del álgebra relacional, incluidas las operaciones del álgebra relacional extendida, se pueden expresar en SQL. SQL también permite la ordenación de los resultados de las consultas según los atributos especificados.
- SQL trata las consultas sobre relaciones que contienen valores nulos añadiendo el valor lógico "desconocido" a los valores lógicos habituales de cierto y falso.
- SQL permite subconsultas anidadas en las cláusulas **where**. La consulta más externa puede llevar a cabo gran variedad de operaciones sobre el resultado de la subconsulta, como la comprobación de relaciones vacías o de pertenencia de valores al resultado de una subconsulta. Las subconsultas de la cláusula **from** se denominan relaciones derivadas.
- Las relaciones se definen como relaciones que contienen el resultado de consultas. Las vistas resultan útiles para ocultar información innecesaria y para recopilar información de más de una relación en una única vista.
- Las vistas temporales definidas con la cláusula **with** también resultan útiles para descomponer consultas complejas en partes más pequeñas y más fáciles de comprender.
- SQL proporciona constructores para actualizar, insertar y borrar información. Las actualizaciones mediante vistas sólo se permiten cuando se cumplen algunas condiciones bastante restrictivas.
- Las transacciones son secuencias de consultas y de actualizaciones que, en su conjunto, desempeñan una tarea. Las transacciones se pueden comprometer o retroceder; cuando se hace retroceder una transacción, se deshacen los efectos de todas las actualizaciones llevadas a cabo por esa transacción.
- SQL soporta varios tipos de reunión externa con diferentes tipos de condiciones de reunión.

conductor (*número_carné*, *nombre*, *dirección*)
coche (*matrícula*, *modelo*, *año*)
accidente (*número_parte*, *fecha*, *lugar*)
posee (*número_carné*, *matrícula*)
participó (*número_carné*, *coche*, *número_parte*, *importe_daños*)

Figura 3.11 Base de datos de seguros.

Términos de repaso

- LDD: lenguaje de definición de datos.
- LMD: lenguaje de manipulación de datos.
- Cláusula **select**.
- Cláusula **from**.
- Cláusula **where**.
- Cláusula **as**.
- Variable tupla.
- Cláusula **order by**.
- Valores duplicados.
- Operaciones con conjuntos:
  - **union**, **intersect**, **except**.
- Funciones de agregación:
  - **avg**, **min**, **max**, **sum**, **count**.
  - **group by**.
- Valores nulos.
  - Valor lógico "desconocido".
- Subconsultas anidadas.
- Operaciones con conjuntos:
  - $\{<, <=, >, >=\}$ { **some, all** }.
  - **exists**.
  - **unique**.
- Relaciones derivadas (en la cláusula **from**).
- Cláusula **with**.
- Vistas.
  - Definición de vistas.
  - Expansión de vistas.
- Modificación de la base de datos.
  - **delete**, **insert**, **update**.
  - Actualización de vistas.
- Transacciones.
  - Compromiso.
  - Retroceso.
- Tipos de reunión:
  - Reuniones interna y externa.
  - Reuniones externas por la izquierda, por la derecha y completa.
  - **natural**, **using** y **on**.

Ejercicios prácticos

3.1 Considérese la base de datos de seguros de la Figura 3.11, en la que las claves primarias se han subrayado. Formúlense las siguientes consultas SQL para esta base de datos relacional.

a. Determinar el número total de personas cuyos coches se hayan visto involucrados en un accidente en 2005.
b. Añadir un nuevo accidente a la base de datos; supóngase cualquier valor para los atributos necesarios.
c. Borrar el Mazda de "Martín Gómez".

3.2 Considérese la base de datos de empleados de la Figura 3.12, donde las claves primarias se han subrayado. Proporciónese una expresión SQL para cada una de las consultas siguientes.

a. Determinar el nombre y ciudad de residencia de todos los empleados que trabajan en el Banco Importante.

empleado (*nombre_empleado*, *calle*, *ciudad*)
trabaja (*nombre_empleado*, *nombre_empresa*, *sueldo*)
empresa (*nombre_empresa*, *ciudad*)
jefe (*nombre_empleado*, *nombre_jefe*)

Figura 3.12 Base de datos de empleados.

b. Determinar el nombre, domicilio y ciudad de residencia de todos los empleados que trabajan en el Banco Importante y ganan más de 10.000 €.
c. Determinar todos los empleados de la base de datos que no trabajan para el Banco Importante.
d. Determinar todos los empleados de la base de datos que ganan más que cualquier empleado del Banco Pequeño.
e. Supóngase que las empresas pueden tener sede en varias ciudades. Determinar todas las empresas con sede en todas las ciudades en las que tiene sede el Banco Pequeño.
f. Determinar la empresa que tiene el mayor número de empleados.
g. Determinar las empresas cuyos empleados ganan un sueldo más alto, en media, que el sueldo medio del Banco Importante.

3.3 Considérese la base de datos relacional de la Figura 3.12. Formúlese una expresión en SQL para cada una de las siguientes consultas.

a. Modificar la base de datos de forma que Santos viva en Tres Cantos
b. Incrementar en un 10 por ciento el sueldo de todos los jefes del Banco Importante, a menos que su sueldo pase a ser mayor de 100.000 €, en cuyo caso se incrementará su sueldo sólo en un 3 por ciento.

3.4 SQL-92 proporciona una operación n-aria denominada **coalesce** (fusión) que se define del modo siguiente: **coalesce**$(A_1, A_2, \ldots, A_n)$ devuelve el primer A_i no nulo de la lista $A_1, A_2, \ldots, A_n$ y devuelve un valor nulo si todos son nulos.

Sean a y b relaciones con los esquemas A(*nombre, dirección, puesto*) y B(*nombre, dirección, sueldo*), respectivamente. Indíquese la manera de expresar a **natural full outer join** b, utilizando la operación **full outer-join** con una condición **on** y la operación **coalesce**. Compruébese que la relación resultado no contiene dos copias de los atributos *nombre* y *dirección* y que la solución es válida aunque alguna tupla de a o de b tenga valores nulos para los atributos *nombre* o *dirección*.

3.5 Supóngase que se tiene una relación *notas*(*id_estudiante*, *puntuación*) y que se quiere clasificar a los estudiantes en función de la puntuación del modo siguiente: SS si $puntuación < 40$, AP si $40 \leq puntuación < 60$, NT si $60 \leq puntuación < 80$ y SB si $80 \leq puntuación$. Escríbanse consultas para hacer lo siguiente:

a. Mostrar la clasificación de cada estudiante, en términos de la relación *notas*.
b. Determinar el número de estudiantes en cada clase.

3.6 Considérese la consulta SQL

select $p.a1$
from p, $r1$, $r2$
where $p.a1 = r1.a1$ **or** $p.a1 = r2.a1$

¿Bajo qué condiciones la consulta anterior selecciona los valores de $p.a1$ que están exclusivamente en $r1$ o en $r2$? Examínense cuidadosamente los casos en los que $r1$ o $r2$ pueden estar vacíos.

3.7 Algunos sistemas permiten los valores nulos *marcados*. Un valor nulo marcado $\perp_i$ es igual a sí mismo, pero si $i \neq j$, entonces $\perp_i \neq \perp_j$. Una aplicación de los valores nulos marcados es permitir ciertas actualizaciones mediante vistas. Considérese la vista *info_préstamo* (Apartado 3.9). Muéstrese la manera en que se pueden utilizar los valores nulos marcados para permitir la inserción de la tupla ("González", 1900) mediante *info_préstamo*.

Ejercicios

3.8 Considérese la base de datos de seguros de la Figura 3.11, en la que las claves primarias están subrayadas. Créense las siguientes consultas SQL para esta base de datos relacional.

a. Determinar el número de accidentes en que han estado implicados los coches pertenecientes a "Martín Gómez".

b. Actualizar a 3.000 € el importe de los daños del coche con número de matrícula "2000 ABC" en el accidente con número de parte "PA2197".

3.9 Considérese la base de datos de empleados de la Figura 3.12, en la que las claves primarias se han subrayado. Proporciónese una expresión SQL para cada una de las consultas siguientes.

a. Determinar el nombre de todos los empleados que trabajan en el Banco Importante.
b. Determinar todos los empleados de la base de datos que viven en la misma ciudad que la empresa para la que trabajan.
c. Determinar todos los empleados de la base de datos que viven en la misma ciudad y en la misma calle que sus jefes.
d. Determinar todos los empleados que ganan más que el sueldo medio de los empleados de su empresa.
e. Determinar la empresa que tiene la nómina más pequeña.

3.10 Considérese la base de datos relacional de la Figura 3.12. Formúlese una expresión en SQL para cada una de las siguientes consultas:

a. Incrementar en un 10 por ciento el sueldo de todos los empleados del Banco Importante.
b. Incrementar en un 10 por ciento el sueldo de todos los jefes del Banco Importante.
c. Borrar todas las tuplas de la relación *trabaja* correspondientes a los empleados del Banco Importante.

3.11 Considérense los esquemas de relación siguientes:

$$R = (A, B, C)$$
$$S = (D, E, F)$$

Considérense también las relaciones *r*(*R*) y *s*(*S*). Obténgase la expresión SQL equivalente a las siguientes consultas:

a. $\Pi_A(r)$
b. $\sigma_{B=17}\,(r)$
c. $r \times s$
d. $\Pi_{A,F}\,(\sigma_{C=D}(r \times s))$

3.12 Sea $R = (A, B, C)$ y sean r_1 y r_2 relaciones sobre el esquema *R*. Proporciónese una expresión SQL equivalente a cada una de las siguientes consultas:

a. $r_1 \cup r_2$
b. $r_1 \cap r_2$
c. $r_1 - r_2$
d. $\Pi_{AB}(r_1) \bowtie \Pi_{BC}(r_2)$

3.13 Demuéstrese que en SQL $<>$ **all** es equivalente a **not in**.

3.14 Considérese la base de datos relacional de la Figura 3.12. Utilizando SQL, defínase una vista consistente en *nombre_jefe* y el sueldo medio de todos los empleados que trabajan para ese jefe. Explíquese por qué el sistema de base de datos no debería permitir que las actualizaciones se expresaran en términos de esta vista.

3.15 Escríbase una consulta SQL, sin usar la cláusula **with**, para determinar todas las sucursales en las que el saldo total de las cuentas sea menor que la media del saldo total de todas las sucursales,

a. Usando una consulta anidada en la cláusula **from**.
b. Usando una consulta anidada en una cláusula **having**.

3.16 Determínense dos razones por las que se puedan introducir valores nulos en una base de datos.

3.17 Mostrar la manera de expresar la operación **coalesce** del Ejercicio 3.4 mediante la operación **case**.

3.18 Dada una definición de esquema SQL para la base de datos de empleados de la Figura 3.12, elíjase un dominio adecuado para cada atributo y una clave primaria adecuada para cada esquema de relación.

3.19 Usando las relaciones de la base de datos bancaria de ejemplo, escríbanse expresiones SQL para definir las vistas siguientes:

- **a.** Una vista que contenga el número de cuenta y el nombre de los clientes (pero no el saldo) de todas las cuentas de la sucursal de El Escorial.
- **b.** Una vista que contenga el nombre y dirección de todos los clientes que tengan una cuenta abierta en el banco, pero que no tengan concedido ningún préstamo.
- **c.** Una vista que contenga el nombre y saldo medio de la cuenta de todos los clientes de la sucursal de Collado Villalba.

3.20 Para cada una de las vistas definidas en el Ejercicio 3.19, explíquese cómo se llevarían a cabo las actualizaciones (si es que se deben permitir).

3.21 Considérese el siguiente esquema relacional

empleado(*número_empleado*, *nombre*, *sucursal*, *edad*)
libros(*isbn*, *título*, *autores*, *editorial*)
préstamo(*número_empleado*, *isbn*, *fecha*)

Escríbanse las siguientes consultas en SQL.

- **a.** Escribir el nombre de los empleados que hayan pedido algún libro publicado por McGraw-Hill.
- **b.** Escribir el nombre de los empleados que hayan pedido todos los libros publicados por McGraw-Hill.
- **c.** Para cada editorial, escribir el nombre de los empleados que han pedido más de cinco libros de esa editorial.

3.22 Considérese el esquema relacional

estudiante(*id_estudiante*, *nombre_estudiante*)
matriculado(*id_estudiante*, *id_asignatura*)

Escríbase una consulta SQL para obtener el ID de estudiante y el nombre de cada estudiante, así como el número total de asignaturas en las que se ha matriculado. Los estudiantes que no estén matriculados en ninguna asignatura deben aparecer también, con el número de asignaturas en las que se han matriculado como 0.

3.23 Supóngase que se tiene una relación *notas*(*id_estudiante*, *puntuación*). Escríbase una consulta SQL para determinar la *clasificación densa* de cada estudiante. Es decir, todos los estudiantes con la nota más alta obtienen una clasificación de 1, los que tienen la siguiente nota más alta obtienen una clasificación de 2, etc. *Sugerencia*: divídase la tarea en partes con la cláusula **with**.

Notas bibliográficas

La versión original de SQL, denominada Sequel 2, se describe en Chamberlin et al. [1976]. Sequel 2 procede de los lenguajes Square (Boyce et al. [1975] y Chamberlin y Boyce [1974]). La norma American National Standard SQL-86 se describe en ANSI [1986]. La definición de SQL según la Arquitectura de aplicación de sistemas (System Application Architecture) de IBM se describe en IBM [1987]. Las normas oficiales de SQL-89 y de SQL-92 están disponibles en ANSI [1989] y ANSI [1992], respectivamente.

Entre las descripciones del lenguaje SQL-92 en libros de texto están Date y Darwen [1997], Melton y Simon [1993] y Cannan y Otten [1993]. Date y Darwen [1997] y Date [1993a] incluyen una crítica de SQL-92.

Entre los libros de texto sobre SQL:1999 están Melton y Simon [2001] y Melton [2002]. Eisenberg y Melton [1999] ofrecen una visión general de SQL:1999. Donahoo y Speegle [2005] abordan SQL desde el punto de vista de los desarrolladores. Eisenberg et al. [2004] ofrecen una visión general de SQL:2003.

Las normas SQL:1999 y SQL:2003 están publicadas como conjuntos de documentos de normas ISO/IEC, que se describen con más detalle en el Apartado 23.3. Los documentos de normas tienen gran den-

sidad de información, resultan difíciles de leer y son útiles, sobre todo, para los implementadores de bases de datos. Los documentos de normas están disponibles para su compra electrónica en el sitio web http://webstore.ansi.org.

Muchos productos de bases de datos soportan más características de SQL que las especificadas en las normas, y puede que no soporten algunas características de la norma. Se puede encontrar más información sobre estas características en los manuales de usuario de SQL de los productos respectivos.

El procesamiento de las consultas SQL, incluidos los algoritmos y las consideraciones sobre rendimiento, se estudia en los Capítulos 13 y 14. En esos mismos capítulos hay referencias bibliográficas al respecto.

Las reglas usadas por SQL para determinar si es posible actualizar una vista y la manera en que se reflejan estas actualizaciones en las relaciones subyacentes de las bases de datos se definen en la norma SQL:1999 y están resumidas en Melton y Simon [2001].

CAPÍTULO 4

SQL avanzado

En el Capítulo 3 se ofreció un tratamiento detallado de la estructura básica de SQL. El lenguaje SQL ha evolucionado a partir de finales de los años setenta desde un lenguaje con pocas funcionalidades a un lenguaje complejo con características para satisfacer a muchos tipos diferentes de usuarios. En este capítulo se tratarán algunas de las características avanzadas de SQL. Se seguirá empleando en los ejemplos el esquema bancario, que se reproduce por comodidad en la Figura 4.1.

4.1 Tipos de datos y esquemas

Ya hemos visto que debe haber asociado un tipo, es decir, un dominio de valores posibles, con cada atributo. En el Capítulo 3 se vieron varios tipos de datos predefinidos soportados por SQL, como los tipos enteros, los reales y los de carácter. Hay otros tipos de datos predefinidos soportados por SQL, que se describirán a continuación. También se describirá la manera de crear en SQL tipos sencillos definidos por los usuarios.

4.1.1 Tipos de datos predefinidos

Además de los tipos de datos básicos que se presentaron en el Apartado 3.2, la norma SQL soporta otros tipos de datos predefinidos, como:

- **date**. Una fecha de calendario que contiene el año (de cuatro cifras), el mes y el día del mes.
- **time**. La hora del día, en horas, minutos y segundos. Se puede usar una variante, **time**(p), para especificar el número de cifras decimales para los segundos (el valor predeterminado es 0). También es posible almacenar la información del huso horario junto a la hora especificando **time with timezone**.
- **timestamp**. Una combinación de **date** y **time**. Se puede usar una variante, **timestamp**(p), para especificar el número de cifras decimales para los segundos (el valor predeterminado es seis). También se almacena información sobre el huso horario si se especifica **with timezone**.

sucursal (*nombre_sucursal*, *ciudad_sucursal*, *activos*)
cliente (*nombre_cliente*, *calle_cliente*, *ciudad_cliente*)
préstamo (*número_préstamo*, *nombre_sucursal*, *importe*)
prestatario (*nombre_cliente*, *número_préstamo*)
cuenta (*número_cuenta*, *nombre_sucursal*, *saldo*)
impositor (*nombre_cliente*, *número_cuenta*)

Figura 4.1 Esquema de la entidad bancaria.

Los valores de fecha y hora se pueden especificar de esta manera:

date '2001-04-25'
time '09:30:00'
timestamp '2001-04-25 10:29:01.45'

La fecha se debe especificar en el formato de año seguido del mes y del día, tal y como se muestra. El campo segundos de **time** y **timestamp** puede tener parte decimal, como puede verse en el ejemplo anterior.

Se pueden usar expresiones de la forma **cast** e **as** t para convertir una cadena de caracteres (o una expresión de tipo cadena de caracteres) e al tipo t, donde t es de tipo **date, time** o **timestamp**. La cadena de caracteres debe tener el formato adecuado, como se indicó al comienzo de este párrafo. Si es necesaria, la información sobre el huso horario puede deducirse de la configuración del sistema.

Para extraer campos concretos de un valor d de **date** o de **time** se puede utilizar **extract** (*campo* **from** d), donde *campo* puede ser **year, month, day, hour, minute** o **second**. La información sobre el huso horario puede obtenerse mediante **timezone_hour** y **timezone_minute**.

SQL también define varias funciones útiles para obtener la fecha y la hora actuales. Por ejemplo, **current_date** devuelve la fecha actual, **current_time** devuelve la hora actual (con su huso horario) y **localtime** devuelve la hora local actual (sin huso horario). Las marcas de tiempo (fecha y hora) se obtienen con **current_timestamp** (con huso horario) y **localtimestamp** (fecha y hora locales sin huso horario).

SQL permite realizar operaciones de comparación sobre todos los tipos de datos que se han mencionado aquí, así como operaciones aritméticas y de comparación sobre los diferentes tipos de datos numéricos. SQL también proporciona un tipo de datos denominado **interval** y permite realizar cálculos basados en fechas, horas e intervalos. Por ejemplo, si x e y son del tipo **date**, entonces $x - y$ es un intervalo cuyo valor es el número de días desde la fecha x hasta la y. De forma análoga, al sumar o restar un intervalo a una fecha o a una hora se obtiene como resultado otra fecha u hora, respectivamente.

A menudo resulta útil comparar valores de diferentes tipos de datos **compatibles**. Por ejemplo, supóngase que el tipo de datos de *nombre_cliente* es una cadena de caracteres de longitud 20 y el tipo de datos de *nombre_sucursal* es una cadena de caracteres de longitud 15. Aunque la longitud de las cadenas sea diferente, la norma SQL considera que los dos tipos de datos son compatibles. Como ejemplo adicional, como cada entero perteneciente al tipo **smallint** es también un entero, las comparaciones $x < y$, donde x es de tipo **smallint** e y es de tipo **int** (o viceversa), son válidas. Este tipo de comparación se lleva a cabo transformando primero el número x al tipo **int**. Las transformaciones de este tipo se denominan **coerción**. La **coerción de tipos** se emplea de manera habitual en los lenguajes de programación comunes, así como en los sistemas de bases de datos.

4.1.2 Tipos definidos por los usuarios

SQL soporta dos formas de tipos de datos definidos por los usuarios. La primera forma, que se tratará a continuación, se denomina **alias de tipos** (**distinct types**). La otra forma, denominada **tipos de datos estructurados**, permite la creación de tipos de datos complejos, que pueden anidar estructuras de registro, arrays y multiconjuntos. En este capítulo no se tratan los tipos de datos complejos, pero se describen más adelante, en el Capítulo 9.

Varios atributos pueden ser del mismo tipo de datos. Por ejemplo, los atributos *nombre_cliente* y *nombre_empleado* pueden tener el mismo dominio: el conjunto de todos los nombres propios. No obstante, los dominios de *saldo* y *nombre_sucursal*, ciertamente, deben ser diferentes. Quizás resulte menos evidente si *nombre_cliente* y *nombre_sucursal* deben tener el mismo dominio. En el nivel de implementación tanto los nombres de los clientes como los de las sucursales son cadenas de caracteres. Sin embargo, normalmente no se considera que la consulta "Determinar todos los clientes que tengan el mismo nombre que alguna sucursal" sea una consulta con sentido. Por tanto, si se considera la base de datos desde el nivel conceptual, en vez de hacerlo desde el nivel físico, *nombre_cliente* y *nombre_sucursal* deben tener dominios distintos.

A nivel práctico resulta más grave asignar el nombre de un cliente a una sucursal; de manera parecida, comparar directamente un valor monetario expresado en euros con otro valor monetario expresado en libras también es, seguramente, un error de programación. Un buen sistema de tipos de datos debe

poder detectar este tipo de asignaciones o comparaciones. Para soportar estas comprobaciones, SQL aporta el concepto de **alias de tipos**.

La cláusula **create type** puede utilizarse para definir tipos de datos nuevos. Por ejemplo, las instrucciones:

create type *Euros* **as numeric**(12,2) **final**
create type *Libras* **as numeric**(12,2) **final**

declaran los tipos de datos definidos por los usuarios *Euros* y *Libras* como números decimales con un total de doce cifras, dos de las cuales se hallan tras la coma decimal. (La palabra clave **final** no resulta realmente significativa en este contexto, pero la norma de SQL:1999 la exige por motivos que no vienen al caso; algunas implementaciones permiten omitir la palabra clave **final**). Los tipos recién creados pueden utilizarse, por ejemplo, como tipos de los atributos de las relaciones. Por ejemplo, se puede declarar la tabla *cuenta* como:

create table *cuenta*
(*número_cuenta* **char**(10),
nombre_sucursal **char**(15),
saldo Euros)

Cualquier intento de asignar un valor de tipo *Euros* a una variable de tipo *Libras* daría como resultado un error de compilación, aunque ambos sean del mismo tipo numérico. Una asignación de ese tipo probablemente se deba a un error de programación, en el que el programador haya olvidado las diferencias de divisas. La declaración de tipos distintos para divisas diferentes ayuda a detectar esos errores.

Como consecuencia del control riguroso de los tipos de datos, la expresión (*saldo.cuenta* + 20) no se aceptaría, ya que el atributo y la constante entera 20 tienen diferentes tipos de datos. Los valores de un tipo de datos pueden *convertirse* (*cast*) a otro dominio como se muestra a continuación:

cast (*saldo.cuenta* **to** *numeric*(12,2))

Se podría realizar la suma sobre el tipo numeric, pero para volver a guardar el resultado en un atributo del tipo *Euros* habría que emplear otra expresión cast para volver a convertir el tipo de datos a *Euros*.

SQL también ofrece las cláusulas **drop type** y **alter type** para eliminar o modificar los tipos de datos que se han creado anteriormente.

Antes incluso de la incorporación a SQL de los tipos de datos definidos por los usuarios (en SQL:1999), SQL tenía el concepto parecido, pero sutilmente diferente, de **tipo de dominio** (**domain type**) (introducido en SQL-92). Se podría definir el tipo de dominio *EEuros* de la manera siguiente:

create domain *EEuros* **as numeric**(12,2)

El tipo de dominio *EEuros* puede utilizarse como tipo de atributo, igual que se ha utilizado el tipo de datos *Euros*. Sin embargo, hay dos diferencias significativas entre los tipos de datos y los dominios:

1. Sobre los dominios se pueden especificar restricciones, como **not null**, y valores predeterminados para las variables del tipo de dominio, mientras que no se pueden especificar restricciones ni valores predeterminados sobre los tipos de datos definidos por los usuarios. Los tipos de datos definidos por los usuarios están diseñados no sólo para utilizarlos para especificar los tipos de datos de los atributos, sino también en extensiones procedimentales de SQL en las que puede que no sea posible aplicar restricciones. Más adelante se volverá al problema de las restricciones sobre los dominios, en el Apartado 4.2.4.

2. Los dominios no tienen tipos estrictos. En consecuencia, los valores de un tipo de dominio pueden asignarse a los de otro tipo de dominio, siempre y cuando los tipos subyacentes sean compatibles.

4.1.3 Tipos de datos para objetos grandes

Muchas aplicaciones de bases de datos de última generación necesitan almacenar atributos que pueden ser de gran tamaño (del orden de muchos kilobytes), como pueden ser fotografías de personas, o muy grande (del orden de muchos megabytes o, incluso, gigabytes), como las imágenes médicas de alta resolución o fragmentos de vídeo. SQL, por tanto, ofrece nuevos tipos de datos para objetos de gran tamaño para los datos de caracteres (**clob**) y para los datos binarios (**blob**). Las letras "lob" de estos tipos de datos significan "objeto grande" (Large Object). Por ejemplo, se pueden declarar los atributos

revista_literaria **clob**(10KB)
imagen **blob**(10MB)
película **blob**(2GB)

La ejecución de consultas SQL suele recuperar una o más filas del resultado en la memoria. Los objetos grandes suelen utilizarse en aplicaciones externas, y para objetos de tamaño muy grande (de varios megabytes a gigabytes), resulta poco eficiente o poco práctico recuperar en la memoria objetos de gran tamaño de tipo entero. En vez de eso, las aplicaciones suelen utilizar las consultas SQL para recuperar "localizadores" de los objetos de gran tamaño y luego emplean el localizador para manipular el objeto desde el lenguaje anfitrión. Por ejemplo, la interfaz de programación de aplicaciones JDBC (que se describe en el Apartado 4.5.2) permite recuperar un localizador en lugar de todo el objeto de gran tamaño; luego se puede emplear el localizador para recuperar este objeto en fragmentos más pequeños, en vez de recuperarlo todo de golpe, de manera parecida a como se leen los datos de los archivos del sistema operativo mediante llamadas a funciones de lectura.

4.1.4 Esquemas, catálogos y entornos

Para comprender la razón de que existan esquemas y catálogos, considérese la manera en que se dan nombres a los archivos en los sistemas de archivos. Los primeros sistemas de archivos eran "planos", es decir, todos los archivos se almacenaban en un único directorio. Los sistemas de archivos de última generación, evidentemente, tienen una estructura de directorios y los archivos se almacenan en diferentes subdirectorios. Para dar un nombre único a un archivo hay que especificar su nombre completo, por ejemplo, /usuarios/avi/libro-bd/capítulo4.tex.

Al igual que los primeros sistemas de archivos, los primeros sistemas de bases de datos tenían también un único espacio de nombres para todas las relaciones. Los usuarios tenían que coordinarse para asegurarse de que no intentaban emplear el mismo nombre para relaciones diferentes. Los sistemas de bases de datos contemporáneos ofrecen una jerarquía de tres niveles para los nombres de las relaciones. El nivel superior de la jerarquía consta de **catálogos**, cada uno de los cuales puede contener **esquemas**. Los objetos de SQL, como las relaciones y las vistas, están contenidos en **esquemas**. (Algunas implementaciones de las bases de datos emplean el término "base de datos" en lugar de catálogo).

Para llevar a cabo cualquier acción sobre la base de datos los usuarios (o los programas) primero deben *conectarse* a ella. El usuario debe proporcionar el nombre de usuario y, generalmente, una contraseña secreta para que se compruebe su identidad. Cada usuario tiene un catálogo y un esquema predeterminados, y esa combinación es única para cada usuario. Cuando un usuario se conecta a un sistema de bases de datos, se configuran para la conexión el catálogo y el esquema predeterminados; esto equivale a la conversión del directorio actual en el directorio de inicio del usuario cuando éste inicia sesión en un sistema operativo.

Para identificar de manera única cada relación hay que utilizar un nombre con tres partes. Por ejemplo,

catálogo5.esquema_bancario.cuenta

Se puede omitir el componente del catálogo, en cuyo caso la parte del nombre correspondiente al catálogo se considera que es el catálogo predeterminado de la conexión. Por tanto, si catálogo5 es el catálogo predeterminado, se puede utilizar esquema_bancario.cuenta para identificar de manera única la misma relación. Además, también se puede omitir el nombre del esquema y la parte correspondiente al esquema del nombre vuelve a considerarse el esquema predeterminado de la conexión. Así pues, se puede

emplear únicamente cuenta si el catálogo predeterminado es catálogo5 y el esquema predeterminado es esquema_bancario.

Cuando se dispone de varios catálogos y de varios esquemas, diferentes aplicaciones y usuarios pueden trabajar de manera independiente sin preocuparse de posibles coincidencias de nombres. Además, pueden ejecutarse varias versiones de la misma aplicación—una versión de producción, otras versiones de prueba—sobre el mismo sistema de bases de datos.

El catálogo y el esquema predeterminados forman parte de un **entorno SQL** que se configura para cada conexión. El entorno contiene, además, el identificador del usuario (también denominado *identificador de autorización*). Todas las instrucciones habituales de SQL, incluidas las instrucciones de LDD y de LMD, operan en el contexto de un esquema. Se pueden crear y descartar esquemas mediante las instrucciones **create schema** y **drop schema**. La creación y la eliminación de catálogos depende de cada implementación y no forma parte de la norma de SQL.

4.2 Restricciones de integridad

Las restricciones de integridad garantizan que las modificaciones realizadas en la base de datos por los usuarios autorizados no den lugar a una pérdida de la consistencia de los datos. Por tanto, las restricciones de integridad protegen contra daños accidentales a las bases de datos.

Algunos ejemplos de restricciones de integridad son:

- El saldo de las cuentas no puede ser nulo.
- No puede haber dos cuentas con el mismo número.
- Todos los números de cuenta de la relación *impositor* deben tener el número de cuenta correspondiente en la relación *cuenta*.
- El salario por hora de los empleados del banco debe ser, como mínimo, de 6,00 € la hora.

En general, las restricciones de integridad pueden ser predicados arbitrarios que hagan referencia a la base de datos. Sin embargo, puede resultar costosa la comprobación de estos predicados arbitrarios. Por tanto, la mayor parte de los sistemas de bases de datos permiten especificar restricciones de integridad que puedan probarse con una sobrecarga mínima. En el Capítulo 7 se estudia otra forma de restricción de integridad, denominada **dependencia funcional**, que se utiliza sobre todo en el proceso de diseño de esquemas.

4.2.1 Restricciones sobre una sola relación

En el Apartado 3.2 se describió la manera de definir tablas mediante el comando **create table**. Este comando también puede incluir instrucciones para restricciones de integridad. Además de la restricción "de clave primaria", hay varias más que pueden incluirse en el comando **create table**. Entre las restricciones de integridad permitidas se encuentran:

- **not null**
- **unique**
- **check**(<predicado>)

Cada uno de estos tipos de restricciones se trata en los apartados siguientes.

4.2.2 Restricción not null

Como se estudió en el Capítulo 2, el valor nulo (*null*) es miembro de todos los dominios y, en consecuencia, de manera predeterminada es un valor legal para todos los atributos de SQL. Para determinados atributos, sin embargo, los valores nulos pueden resultar poco adecuados. Considérese una tupla de la relación *cuenta* en la que *número_cuenta* sea nulo. Este tipo de tupla proporciona información de las cuentas desconocidas; por tanto, no contiene información útil. De manera parecida, no es deseable que

el saldo de la cuenta sea un valor nulo. En estos casos así se deseará prohibir los valores nulos, lo que se puede hacer restringiendo el dominio de los atributos *número_cuenta* y *saldo* para excluir los valores nulos, declarándolos de la manera siguiente:

número_cuenta **char**(10) **not null**
saldo **numeric**(12,2) **not null**

La especificación **not null** prohíbe la inserción de valores nulos para ese atributo. Cualquier modificación de la base de datos que haga que se inserte un valor nulo en un atributo declarado como **not null** genera un diagnóstico de error.

Existen muchas situaciones en las que se desea evitar los valores nulos. En concreto, SQL prohíbe los valores nulos en la clave primaria de los esquemas de las relaciones. Por tanto, en el ejemplo bancario, en la relación *cuenta*, si el atributo *número_cuenta* se declara clave primaria de *cuenta*, no puede adoptar valores nulos. En consecuencia, no hace falta declararlo **not null** de manera explícita.

La especificación **not null** también puede aplicarse a declaraciones de dominio definidas por los usuarios; en consecuencia, no se permitirá a los atributos de ese tipo de dominio tomar valores nulos. Por ejemplo, si se desea que el dominio *Euros* no tome valores nulos, se puede declarar de la manera siguiente:

create domain *Euros* **numeric**(12,2) **not null**

4.2.3 Restricción unique

SQL también soporta la restricción de integridad

unique $(A_{j_1}, A_{j_2}, \ldots, A_{j_m})$

La especificación **unique** indica que los atributos $A_{j_1}, A_{j_2}, \ldots, A_{j_m}$ forman una clave candidata; es decir, ningún par de tuplas de la relación puede ser igual en todos los atributos de la clave primaria. Sin embargo, se permite que los atributos de la clave candidata tengan valores nulos, a menos que se hayan declarado de manera explícita como **not null**. Recuérdese que los valores nulos no son iguales a ningún otro valor. (El tratamiento de los valores nulos en este caso es el mismo que el del constructor **unique** definido en el Apartado 3.7.4).

4.2.4 La cláusula check

La cláusula **check** de SQL puede aplicarse a declaraciones de relaciones y a declaraciones de dominios. Cuando se aplica a declaraciones de relaciones, la cláusula **check**(P) especifica un predicado P que deben cumplir todas las tuplas de la relación.

Un uso frecuente de la cláusula **check** es garantizar que los valores de los atributos cumplan las condiciones especificadas, lo que permite en realidad construir un potente sistema de tipos. Por ejemplo, la cláusula **check**(*activos*>=0) en el comando **create table** de la relación *sucursal* garantiza que el valor de **activos** no sea negativo.

Considérese ahora el siguiente ejemplo:

```
create table estudiante
    (nombre            char(15)  not null,
     id_estudiante     char(10),
     tipo_titulación   char(15),
     primary key (id_estudiante),
     check (tipo_titulación in ('Diplomatura', 'Licenciatura', 'Doctorado')))
```

En este caso se utiliza la cláusula **check** para simular un tipo enumerado, especificando que *tipo_titulación* debe ser 'Diplomatura', 'Licenciatura' o 'Doctorado'.

Cuando se aplica a un dominio, la cláusula **check** permite a los diseñadores de esquemas especificar un predicado que debe cumplir cualquier valor asignado a las variables cuyo tipo de datos sea el dominio.

Por ejemplo, una cláusula **check** puede garantizar que el dominio del salario por hora sólo permita valores mayores que un mínimo especificado (como puede ser el salario mínimo):

create domain *SalarioPorHora* **numeric**(5,2)
constraint *prueba_valor_salario* **check**(**value** $>= 6.00$)

Se ha impuesto sobre el dominio *SalarioPorHora* una restricción que garantiza que el salario por hora sea mayor o igual que 6.00. La cláusula **constraint** *prueba_valor_salario* es optativa, y se emplea para dar el nombre *prueba_valor_salario* a la restricción. El nombre lo utiliza el sistema para indicar la restricción que ha violado alguna actualización.

Como ejemplo adicional, se puede restringir un dominio para que sólo contenga un conjunto determinado de valores mediante la cláusula **in**:

create domain *TipoCuenta* **char**(10)
constraint *prueba_tipo_cuenta*
check (**value in** ('Corriente', 'Ahorro'))

Por tanto, la cláusula **check** permite restringir los atributos y los dominios de una forma potente que la mayor parte de los sistemas del tipos de los lenguajes de programación no permiten.

Las anteriores condiciones de **check** pueden comprobarse con relativa facilidad, al insertar o modificar una tupla. Sin embargo, en general, las condiciones de **check** pueden ser más complejas (y más difíciles de comprobar), ya que se permiten subconsultas que hagan referencias a otras relaciones en la condición **check**. Por ejemplo, se podría especificar esta restricción sobre la relación *cuenta*:

check (*nombre_sucursal* **in** (**select** *nombre_sucursal* **from** *sucursal*))

La condición de **check** comprueba que el *nombre_sucursal* de cada tupla de la relación *cuenta* es realmente el nombre de una sucursal de la relación *sucursal*. Por tanto, no sólo hay que comprobar la condición cuando se inserta o modifica una tupla de *cuenta*, sino también cuando se modifica la relación *sucursal* (en ese caso, cuando se borra o modifica alguna tupla de la relación *sucursal*).

La restricción anterior es, en realidad, un ejemplo de una clase de constantes denominadas restricciones de *integridad referencial*. Este tipo de restricciones, junto con una manera más sencilla de especificarlas en SQL, se estudian en el Apartado 4.2.5.

Las condiciones **check** complejas pueden ser útiles cuando se desea garantizar la integridad de los datos, pero deben emplearse con precaución, ya que pueden resultar costosas de comprobar.

4.2.5 Integridad referencial

A menudo se desea garantizar que el valor que aparece en una relación para un conjunto dado de atributos aparezca también para un conjunto determinado de atributos en otra relación. Esta condición se denomina **integridad referencial**.

Las claves externas pueden especificarse como parte de la instrucción de SQL **create table** mediante la cláusula **foreign key**. Las declaraciones de claves externas se ilustran mediante la definición en el LDD de SQL de parte de la base de datos bancaria, como puede verse en la Figura 4.2. La declaración de la tabla *cuenta* tiente una declaración "**foreign key** (*nombre_sucursal*) **references** *sucursal*". Esta declaración de clave externa especifica que para cada tupla de cuenta, el nombre de sucursal especificado en la tupla debe existir en la relación *sucursal*. Sin esta restricción, es posible que alguna cuenta especifique el nombre de una sucursal inexistente.

De manera más general, sean $r_1(R_1)$ y $r_2(R_2)$ relaciones con las claves primarias C_1 y C_2, respectivamente (recuérdese que R_1 y R_2 denotan el conjunto de atributos de r_1 y de r_2, respectivamente). Se dice que un subconjunto α de R_2 es una **clave externa** que hace referencia a C_1 de la relación r_1 si se exige que, para cada tupla t_2 de r_2, deba haber una tupla t_1 de r_1 tal que $t_1[C_1] = t_2[\alpha]$. Las exigencias de este tipo se denominan **restricciones de integridad referencial**, o **dependencias de subconjuntos**.

```
create table cliente
    (nombre_cliente     char(20),
     calle_cliente      char(30),
     ciudad_cliente     char(30),
     primary key (nombre_cliente))

create table sucursal
    (nombre_sucursal    char(15),
     ciudad_sucursal    char(30),
     activos            numeric(16,2),
     primary key (nombre_sucursal),
     check (activos >= 0))

create table cuenta
    (número_cuenta      char(10),
     nombre_sucursal    char(15),
     saldo              numeric(12,2),
     primary key (número_cuenta),
     foreign key (nombre_sucursal) references sucursal,
     check (saldo >= 0))

create table impositor
    (nombre_cliente     char(20),
     número_cuenta      char(10),
     primary key (nombre_cliente, número_cuenta),
     foreign key (nombre_cliente) references cliente,
     foreign key (número_cuenta) references cuenta)
```

Figura 4.2 Definición de datos con SQL para parte de la base de datos bancaria.

El último término se debe a que la anterior restricción de integrida referencial puede escribirse como $\Pi_\alpha\ (r_2)\ \subseteq\ \Pi_{C_1}\ (r_1)$. Obsérvese que, para que las restricciones de integridad referencial tengan sentido, α y C_1 deben ser conjuntos de atributos compatibles, es decir, o bien α debe ser igual a C_1, o bien deben contener el mismo número de atributos y los tipos de los atributos correspondientes deben ser compatibles (aquí se supone que α y C_1 están ordenados).

De manera predeterminada, en SQL las claves externas hacen referencia a los atributos de la clave primaria de la tabla referenciada. SQL también soporta una versión de la cláusula **references** en la que se puede especificar de manera explícita una lista de atributos de la relación a la que se hace referencia. La lista de atributos especificada, no obstante, debe declararse como clave candidata de la relación a la que hace referencia.

Se puede utilizar la siguiente forma abreviada como parte de la definición de atributos para declarar que el atributo forma una clave externa:

nombre_sucursal **char**(15) **references** *sucursal*

Cuando se viola una restricción de integridad referencial, el procedimiento normal es rechazar la acción que ha causado esa violación (es decir, la transacción que lleva a cabo la acción de actualización se retrocede). Sin embargo, la cláusula **foreign key** puede especificar que si una acción de borrado o de actualización de la relación a la que hace referencia viola la restricción, entonces, en lugar de rechazar la acción, el sistema realice los pasos necesarios para modificar la tupla de la relación que hace la referencia para que se restaure la restricción. Considérese esta definición de una restricción de integridad para la relación *cuenta*:

```
create table cuenta
    ( . . .
    foreign key (nombre_sucursal) references sucursal
                                  on delete cascade
                                  on update cascade,
    . . . )
```

Debido a la cláusula **on delete cascade** asociada con la declaración de la clave externa, si el borrado de una tupla de *sucursal* da lugar a que se viole la restricción de integridad referencial, el sistema no rechaza el borrado. En vez de eso, el borrado "pasa en cascada" a la relación *cuenta*, borrando la tupla que hace referencia a la sucursal que se ha borrado. De manera parecida, el sistema no rechaza las actualizaciones de los campos a los que hace referencia la restricción aunque la violen; en vez de eso, el sistema actualiza también al nuevo valor el campo *nombre_sucursal* de las tuplas de *cuenta* que hacen la referencia. SQL también permite que la cláusula **foreign key** especifique acciones diferentes de **cascade**, si se viola la restricción. El campo que hace la referencia (en este caso, *nombre_sucursal*) puede definirse como nulo (empleando **set null** en lugar de **cascade**) o con el valor predeterminado para el dominio (utilizando **set default**).

Si hay una cadena de dependencias de clave externa que afecta a varias relaciones, el borrado o la actualización de un extremo de la cadena puede propagarse por toda ella. En el Ejercicio práctico 4.4 aparece un caso interesante en el que la restricción de **clave externa** de una relación hace referencia a la misma relación. Si una actualización o un borrado en cascada provoca una violación de la restricción que no pueda tratarse con otra operación en cascada, el sistema aborta la transacción. En consecuencia, todas las modificaciones provocadas por la transacción y sus acciones en cascada se deshacen.

Los valores **nulos** complican la semántica de las restricciones de integridad referencial en SQL. Se permite que los atributos de las claves externas sean valores nulos, siempre que no hayan sido declarados no nulos previamente. Si todas las columnas de una clave externa son no nulas en una tupla dada, se utiliza para esa tupla la definición habitual de las restricciones de clave externa. Si alguna de las columnas de la clave externa es nula, la tupla se define de manera automática para que satisfaga la restricción.

Puede que esta definición no sea siempre la opción correcta, por lo que SQL ofrece también constructores que permiten modificar el comportamiento cuando hay valores nulos; los constructores no se van a estudiar aquí.

Se pueden añadir restricciones de integridad a relaciones ya existentes utilizando el comando **alter table** *nombre-tabla* **add** *restricción*, donde *restricción* puede ser cualquiera de las restricciones que se han examinado. Cuando se ejecuta un comando de este tipo, el sistema se asegura primero de que la relación satisfaga la restricción especificada. Si lo hace, se añade la restricción a la relación; en caso contrario, se rechaza el comando.

Las transacciones pueden constar de varios pasos, y las restricciones de integridad pueden violarse temporalmente tras uno de los pasos, pero puede que uno posterior subsane la violación. Por ejemplo, supóngase la relación *persona* con la clave primaria *nombre*, y el atributo *cónyuge* y supóngase que *cónyuge* es una clave externa de *persona*. Es decir, la restricción impone que el atributo *cónyuge* debe contener un nombre que aparezca en la tabla *persona*. Supóngase que se desea destacar el hecho de que Martín y María están casados entre sí mediante la inserción de dos tuplas, una para cada uno, en la relación anterior. La inserción de la primera tupla violaría la restricción de clave externa, independientemente de cuál de las dos tuplas se inserte primero. Una vez insertada la segunda tupla, la restricción de clave externa vuelve a cumplirse.

Para tratar estas situciones la norma de SQL permite que se añada la cláusula **initially deferred** a la especificación de la restricción; la restricción, en este caso, se comprueba al final de la transacción y no en los pasos intermedios[1]. Las restricciones pueden especificarse de manera alternativa como **deferrable**, que significa que, de manera predeterminada, se comprueba inmediatamente, pero puede diferirse si se desea. Para las restricciones declaradas como diferibles, la ejecución de la instrucción **set constraints**

1. Se puede evitar el problema del ejemplo anterior de otra manera, si el atributo *cónyuge* puede definirse como nulo: se definen los atributos cónyuge como nulos al insertar las tuplas de Martín y María y se actualizan posteriormente. Sin embargo, esta técnica es bastante confusa y no funciona si los atributos no pueden definirse como nulos.

lista-restricciones **deferred** como parte de una transacción hace que se difiera la comprobación de las restricciones especificadas hasta el final de esa transacción.

No obstante, hay que ser consciente de que el comportamiento predeterminado es comprobar las restricciones de manera inmediata, y de que muchas implementaciones no soportan la comprobación diferida de las restricciones.

4.2.6 Asertos

Un **aserto** es un predicado que expresa una condición que la base de datos debe satisfacer siempre. Las restricciones de dominio y las de integridad referencial son formas especiales de los asertos. Se ha prestado una atención especial a estos tipos de asertos porque se pueden verificar con facilidad y pueden afectar a una gran variedad de aplicaciones de bases de datos. Sin embargo, hay muchas restricciones que no se pueden expresar utilizando únicamente estas formas especiales. Ejemplos de estas restricciones son:

- La suma de los importes de todos los préstamos de cada sucursal debe ser menor que la suma de los saldos de todas las cuentas de esa sucursal.
- Cada préstamo tiene al menos un cliente que tiene una cuenta con un saldo mínimo de 1.000 €

En SQL los asertos adoptan la forma

create assertion <nombre-aserto> **check** <predicado>

En la Figura 4.3 se muestra cómo pueden escribirse estos dos ejemplos de restricciones en SQL. Dado que SQL no proporciona ningún mecanismo "para todo X, $P(X)$" (donde P es un predicado), no queda más remedio que implementarlo utilizando su equivalente "no existe X tal que no $P(X)$", que puede escribirse en SQL.

Cuando se crea un aserto, el sistema comprueba su validez. Si el aserto es válido, sólo se permiten las modificaciones posteriores de la base de datos que no hagan que se viole el aserto. Esta comprobación puede introducir una sobrecarga importante si se han realizado asertos complejos. Por tanto, los asertos deben utilizarse con mucha cautela. La elevada sobrecarga debida a la comprobación y al mantenimiento de los asertos ha llevado a algunos desarrolladores de sistemas a soslayar el soporte para los asertos generales, o bien a proporcionar formas especializadas de aserto que resultan más sencillas de comprobar.

```
create assertion restricción_suma check
    (not exists (select * from sucursal
        where (select sum(importe) from préstamo
                where préstamo.nombre_sucursal = sucursal.nombre_sucursal)
            >= (select sum(saldo) from cuenta
                where cuenta.nombre_sucursal = sucursal.nombre_sucursal)))

create assertion restricción_saldo check
    (not exists (select * from préstamo
        where not exists (select *
            from prestatario, impositor, cuenta
            where préstamo.número_préstamo = prestatario.número_préstamo
                and prestatario.nombre_cliente = impositor.nombre_cliente
                and impositor.número_cuenta = cuenta.número_cuenta
                and cuenta.saldo >= 1000)))
```

Figura 4.3 Dos ejemplos de aserto.

4.3 Autorización

Se pueden asignar a los usuarios varios tipos de autorización para diferentes partes de la base de datos. Por ejemplo:

- La autorización de lectura
- La autorización de inserción
- La autorización de actualización
- La autorización de borrado

Cada uno de estos tipos de autorización se denomina **privilegio**. Se puede conceder a cada usuario todos estos tipos de privilegios, ninguno de ellos o una combinación de los mismos sobre partes concretas de la base de datos, como puede ser una relación o una vista.

La norma de SQL incluye los privilegios **select**, **insert**, **update** y **delete**. El privilegio **select** autoriza al usuario a leer los datos. Además de estas formas de privilegio para el acceso a los datos, SQL soporta otros privilegios, como el de crear, borrar o modificar relaciones y ejecutar procedimientos. Estos privilegios se estudiarán más adelante, en el Apartado 8.7. **all privileges** puede utilizarse como forma abreviada de todos los privilegios que se pueden conceder. El usuario que crea una relación nueva recibe de manera automática todos los privilegios sobre esa relación.

Se puede permitir al usuario al que se le ha concedido alguna forma de autorización que transmita (conceda) esa autorización a otros usuarios o que retire (revoque) una autorización concedida previamente.

El lenguaje de definición de datos de SQL incluye comandos para conceder y retirar privilegios. La instrucción **grant** se utiliza para conceder autorizaciones. La forma básica de esta instrucción es:

grant <lista de privilegios> **on** <nombre de relación o de vista> **to** <lista de usuarios o de roles>

La *lista de privilegios* permite la concesión de varios privilegios con un solo comando. El concepto de rol se trata más adelante, en el Apartado 8.7.

La siguiente instrucción **grant** concede a los usuarios de la base de datos Martín y María la autorización **select** sobre la relación *cuenta*:

grant select on *cuenta* **to** *Martín*, *María*

La autorización **update** puede concederse sobre todos los atributos de la relación o sólo sobre algunos. Si se incluye la autorización **update** en una instrucción **grant**, la lista de atributos sobre los que se concede la autorización update puede aparecer entre paréntesis justo después de la palabra clave **update**. Si se omite la lista de atributos, el privilegio **update** se concede sobre todos los atributos de la relación.

La siguiente instrucción **grant** concede a los usuarios Martín y María la autorización update sobre el atributo *importe* de la relación *préstamo*:

grant update (*importe*) **on** *préstamo* **to** *Martín*, *María*

El privilegio **insert** también puede especificar una lista de atributos; cualquier inserción en la relación debe especificar sólo esos atributos y el sistema asigna al resto de los atributos valores predeterminados (si hay alguno definido para ellos) o los define como nulos.

El nombre de usuario **public** hace referencia a todos los usuarios actuales y futuros del sistema. Por tanto, los privilegios concedidos a **public** se conceden de manera implícita a todos los usuarios actuales y futuros.

De manera predeterminada el usuario o rol al que se le concede un privilegio no está autorizado a concedérselo a otro usuario o rol. SQL permite que la concesión de privilegios especifique que el destinatario puede concedérselo, a su vez, a otro usuario. Esta característica se describe con más detalle en el Apartado 8.7.

Para retirar una autorización se emplea la instrucción **revoke**. Su forma es casi idéntica a la de **grant**:

revoke <lista de privilegios> **on** <nombre de la relación o nombre de la vista>
from <lista de usuarios o de roles>

Por tanto, para retirar los privilegios que se han concedido anteriormente:

revoke select on *sucursal* **from** *Martín*, *María*
revoke update (*importe*) **on** *préstamo* **from** *Martín*, *María*

La retirada de privilegios resulta más complicada si el usuario al que se le retiran los privilegios se los ha concedido a otros usuarios. Se volverá a este problema en el Apartado 8.7.

4.4 SQL incorporado

SQL proporciona un lenguaje de consultas declarativo muy potente. La formulación de consultas en SQL es normalmente mucho más sencilla que la formulación de las mismas en un lenguaje de programación de propósito general. Sin embargo, los programadores deben tener acceso a la base de datos desde los lenguajes de programación de propósito general, al menos, por dos razones:

1. No todas las consultas pueden expresarse en SQL, ya que SQL no ofrece todo el poder expresivo de los lenguajes de propósito general. Es decir, hay consultas que se pueden expresar en lenguajes como C, Java o Cobol que no se pueden expresar en SQL. Para formular consultas de este tipo, se puede incorporar SQL en un lenguaje más potente.

   SQL está diseñado de tal forma que las consultas formuladas puedan optimizarse automáticamente y se ejecuten de manera eficiente—y proporcionar toda la potencia de los lenguajes de programación hace la optimización automática extremadamente difícil.

2. Las acciones no declarativas—como la impresión de informes, la interacción con los usuarios o el envío de los resultados de las consultas a una interfaz gráfica—no se pueden llevar a cabo desde el propio SQL. Normalmente, las aplicaciones contienen varios componentes y la consulta o actualización de los datos sólo es uno de ellos; los demás componentes se escriben en lenguajes de programación de propósito general. En el caso de las aplicaciones integradas, los programas escritos en el lenguaje de programación deben poder tener acceso a la base de datos.

La norma SQL define la incorporación de SQL en varios lenguajes de programación, tales como C, Cobol, Pascal, Java, PL/I, y Fortran. El lenguaje en el que se incorporan las consultas SQL se denomina lenguaje *anfitrión* y las estructuras de SQL que se admiten en el lenguaje anfitrión constituyen SQL *incorporado*.

Los programas escritos en el lenguaje anfitrión pueden usar la sintaxis de SQL incorporado para tener acceso a los datos almacenados en la base de datos y actualizarlos. Esta forma incorporada de SQL amplía aún más la capacidad de los programadores de manipular las bases de datos. En SQL incorporado toda la ejecución de las consultas la realiza el sistema de bases de datos, que luego pone el resultado de la consulta a disposición del programa tupla (registro) a tupla.

Los programas de SQL incorporado deben procesarlos un preprocesador especial antes de la compilación. El preprocesador sustituye las peticiones de SQL incorporado por declaraciones escritas en el lenguaje anfitrión y por llamadas a procedimientos que permiten la ejecución de los accesos a la base de datos en el momento de la ejecución. Posteriormente, el compilador del lenguaje anfitrión compila el programa resultante. Para identificar las peticiones al preprocesador de SQL incorporado se utiliza la instrucción EXEC SQL; tiene la forma

EXEC SQL <instrucción de SQL incorporado > END-EXEC

La sintaxis exacta de las peticiones de SQL incorporado depende del lenguaje en el que se haya incorporado SQL. Por ejemplo, cuando se incorpora SQL en C, se utiliza un punto y coma en lugar de END-EXEC. La incorporación de SQL en Java (denominada SQLJ) utiliza la sintaxis

SQL { <instrucción de SQL incorporado > };

En el programa se incluye la instrucción SQL INCLUDE para identificar el lugar donde el preprocesador debe insertar las variables especiales que se emplean para la comunicación entre el programa y el sistema de bases de datos. Las variables del lenguaje anfitrión se pueden utilizar en las instrucciones de SQL incorporado, pero deben ir precedidas de dos puntos (:) para distinguirlas de las variables de SQL.

Antes de ejecutar ninguna instrucción de SQL el programa debe conectarse con la base de datos. Esto se logra mediante

EXEC SQL **connect to** *servidor* **user** *nombre-usuario* END-EXEC

En este caso, *servidor* identifica al servidor con el que hay que establecer la conexión. Algunas implementaciones de las bases de datos pueden exigir que se proporcione una contraseña además del nombre de usuario.

Las instrucciones de SQL incorporado son parecidas en cuanto a la forma a las instrucciones de SQL que se han descrito en este capítulo. Sin embargo, hay varias diferencias que se indican a continuación.

Para formular una consulta relacional se emplea la instrucción **declare cursor**. El resultado de la consulta no se calcula todavía. En vez de eso, el programa debe utilizar los comandos **open** y **fetch** (que se analizarán más adelante en este apartado) para obtener las tuplas resultado.

Considérese el esquema bancario que se ha utilizado en este capítulo. Supóngase la variable del lenguaje anfitrión *importe* y que se desea determinar el nombre y ciudad de residencia de los clientes que tienen más de *importe* euros en alguna de sus cuentas. Se puede escribir esta consulta del modo siguiente:

```
EXEC SQL
        declare c cursor for
        select nombre_cliente, ciudad_cliente
        from impositor, cliente, cuenta
        where impositor.nombre_cliente = cliente.nombre_cliente and
                cuenta.número_cliente = impositor.número_cuenta and
                cuenta.saldo > :importe
END-EXEC
```

La variable *c* de la expresión anterior se denomina *cursor* de la consulta. Se utiliza esta variable para identificar la consulta en la instrucción **open**, que hace que se evalúe la consulta, y en la instrucción **fetch**, que hace que los valores de una tupla se coloquen en las variables del lenguaje anfitrión.

La instrucción **open** para la consulta de anterior es:

EXEC SQL **open** *c* END-EXEC

Esta instrucción hace que el sistema de base de datos ejecute la consulta y guarde el resultado en una relación temporal. La consulta tiene una variable del lenguaje anfitrión (:*importe*); la consulta utiliza el valor de la variable en el momento en que se ejecuta la instrucción **open**.

Si la consulta de SQL genera un error, el sistema de base de datos almacena un diagnóstico de error en las variables del área de comunicación de SQL (SQL communication-area, SQLCA), cuyas declaraciones inserta la instrucción SQL INCLUDE.

El programa de SQL incorporado ejecuta una serie de instrucciones **fetch** para recuperar las tuplas del resultado. La instrucción **fetch** necesita una variable del lenguaje anfitrión por cada atributo de la relación resultado. En la consulta de ejemplo se necesita una variable para almacenar el valor de *nombre_cliente* y otra para el de *ciudad_cliente*. Supóngase que esas variables son *nc* y *cc*, respectivamente. Entonces, la instrucción

EXEC SQL **fetch** *c* **into** :*cn*, :*cc* END-EXEC

genera una tupla de la relación resultado. El programa puede manipular entonces las variables *nc* y *cc* empleando las características del lenguaje de programación anfitrión.

Cada solicitud **fetch** devuelve una sola tupla. Para obtener todas las tuplas del resultado, el programa debe incorporar un bucle para iterar sobre todas las tuplas. SQL incorporado ayuda a los programadores en el tratamiento de esta iteración. Aunque las relaciones sean conceptualmente conjuntos, las tuplas del

resultado de las consultas se hallan en un orden físico determinado. Cuando el programa ejecuta una instrucción **open** sobre un cursor, ese cursor pasa a apuntar a la primera tupla del resultado. Cada vez que se ejecuta una instrucción **fetch**, el cursor se actualiza para que apunte a la siguiente tupla del resultado. Cuando no quedan por procesar más tuplas, la variable SQLSTATE de SQLCA se define como '02000' (que significa "sin datos"). Por tanto, se puede utilizar un bucle **while** (o su equivalente) para procesar cada tupla del resultado.

La instrucción **close** se debe utilizar para indicar al sistema de bases de datos que borre la relación temporal que guardaba el resultado de la consulta. Para el ejemplo anterior, esta instrucción tiene la forma

EXEC SQL **close** *c* END-EXEC

SQLJ, la incorporación en Java de SQL, ofrece una variación del esquema anterior, en la que se emplean los iteradores de Java en lugar de los cursores. SQLJ asocia los resultados de la consulta con un iterador, y se puede utilizar el método next() del iterador de Java para pasar de una tupla a otra del resultado, igual que los ejemplos anteriores utilizan **fetch** sobre el cursor.

Las expresiones de SQL incorporado para la modificación (**update**, **insert** y **delete**) de las bases de datos no devuelven ningún resultado. Por tanto, son, de algún modo, más sencillas de expresar. Una solicitud de modificación de la base de datos tiene la forma

EXEC SQL < cualquier **update, insert** o **delete** válido > END-EXEC

En la expresión de SQL para la modificación de la base de datos pueden aparecer variables del lenguaje anfitrión precedidas de dos puntos. Si se produce un error en la ejecución de la instrucción, se establece un diagnóstico en SQLCA.

Las relaciones de las bases de datos también se pueden actualizar mediante cursores. Por ejemplo, si se desea sumar 100 euros al atributo *saldo* de cada *cuenta* en la que el nombre de sucursal sea "Navacerrada", se puede declarar un cursor como sigue:

```
declare c cursor for
select *
from cuenta
where nombre_sucursal = 'Navacerrada'
for update
```

Después se itera por las tuplas ejecutando operaciones **fetch** sobre el cursor (como se mostró anteriormente) y, después de obtener cada tupla, se ejecuta el siguiente código:

```
update cuenta
set saldo = saldo + 100
where current of c
```

SQL incorporado permite que los programas en el lenguaje anfitrión tengan acceso a la base de datos, pero no proporciona ninguna ayuda para la presentación de los resultados al usuario ni para la generación de informes. La mayor parte de los productos comerciales de bases de datos incluyen herramientas para ayudar a los programadores de aplicaciones a crear interfaces de usuario e informes con formato. El Capítulo 8 describe la manera de crear aplicaciones de bases de datos con interfaces de usuario, concentrándose en las interfaces de usuario basadas en Web.

4.5 SQL dinámico

El componente *dinámico* de SQL permite que los programas construyan y remitan consultas de SQL en tiempo de ejecución. En cambio, las instrucciones de SQL incorporado deben hallarse presentes completamente en el momento de la compilación; las compila el preprocesador de SQL incorporado. Por medio de SQL dinámico los programas pueden crear consultas de SQL en tiempo de ejecución (quizás basadas en datos introducidos por el usuario) y hacer que se ejecuten inmediatamente o dejarlas *preparadas* para

su uso posterior. En el proceso de preparación de una instrucción SQL dinámica ésta se compila, y al usarla posteriormente se aprovecha su versión compilada.

SQL define normas para incorporar las llamadas dinámicas a SQL en el lenguaje anfitrión, por ejemplo, C, como se muestra en el siguiente ejemplo.

```
char * prog_SQL = "update cuenta set saldo = saldo *1.05
                   where número_cuenta = ?";
EXEC SQL prepare prog_din from :prog_SQL;
char cuenta[10] = "C-101";
EXEC SQL execute prog_din using :cuenta;
```

El programa SQL dinámico contiene un símbolo de interrogación ?, que representa un valor que se proporciona cuando se ejecuta el programa de SQL.

Sin embargo, esta sintaxis necesita extensiones para el lenguaje o un preprocesador para el lenguaje extendido. Una alternativa que se utiliza con frecuencia es emplear una API para enviar las consultas o actualizaciones de SQL a un sistema de bases de datos, sin realizar cambios en el propio lenguaje de programación.

En el resto de este apartado se examinan dos normas de conexión a bases de datos de SQL y la realización de consultas y de actualizaciones. Una, ODBC, es una interfaz para programas de aplicación desarrollada inicialmente para el lenguaje C, y extendida posteriormente a otros lenguajes como C++, C# y Visual Basic. La otra, JDBC, es una interfaz para programas de aplicación para el lenguaje Java.

Para entender estas normas hay que comprender el concepto de sesión de SQL. El usuario o la aplicación se *conectan* al servidor de SQL y establecen una sesión; ejecutan una serie de instrucciones y, finalmente, se *desconectan* de la sesión. Así, todas las actividades del usuario o de la aplicación están en el contexto de una sesión de SQL. Además de los comandos normales de SQL, las sesiones también pueden incluir comandos para *comprometer* el trabajo realizado en la sesión o para *retrocederlo*.

4.5.1 ODBC

La norma **ODBC** (Open Database Connectivity, conectividad abierta de bases de datos) define el modo de comunicación entre los programas de aplicación y los servidores de bases de datos. ODBC define una **interfaz para programas de aplicación** (API, Application Program Interface) que pueden utilizar las aplicaciones para abrir conexiones con las bases de datos, enviar las consultas y las actualizaciones y obtener los resultados. Las aplicaciones como las interfaces gráficas de usuario, los paquetes estadísticos y las hojas de cálculo pueden emplear la misma API de ODBC para conectarse a cualquier servidor de bases de datos compatible con ODBC.

Cada sistema de bases de datos compatible con ODBC proporciona una biblioteca que se debe enlazar con el programa cliente. Cuando el programa cliente realiza una llamada a la API de ODBC, el código de la biblioteca se comunica con el servidor para llevar a cabo la acción solicitada y obtener los resultados.

La Figura 4.4 muestra un ejemplo de código C que usa la API de ODBC. El primer paso en el uso de ODBC para comunicarse con un servidor es configurar su conexión. Para ello, el programa asigna en primer lugar un entorno de SQL, luego un manejador para la conexión a la base de datos. ODBC define los tipos HENV, HDBCy RETCODE. El programa abre a continuación la conexión a la base de datos empleando SQLConnect. Esta llamada tiene varios parámetros, como son el manejador de la conexión, el servidor al que hay que conectarse, el identificador de usuario y la contraseña para la base de datos. La constante SQL_NTS indica que el argumento anterior es una cadena terminada en un valor nulo.

Una vez configurada la conexión, el programa puede enviar comandos SQL a la base de datos empleando SQLExecDirect. Las variables del lenguaje C se pueden vincular a los atributos del resultado de la consulta, de forma que cuando se obtenga una tupla resultado mediante SQLFetch, los valores de sus atributos se almacenen en las variables de C correspondientes. La función SQLBindCol realiza esta tarea; el segundo argumento identifica la posición del atributo en el resultado de la consulta, y el tercer argumento indica la conversión de tipos de SQL a C requerida. El siguiente argumento proporciona la dirección de la variable. Para los tipos de datos de longitud variable como los arrays de caracteres, los dos últimos argumentos proporcionan la longitud máxima de la variable y una ubicación donde se debe almacenar la longitud real cuando se obtenga una tupla. Un valor negativo devuelto para el argumento

```
void ODBCexample()
{
    RETCODE error;
    HENV ent; /* entorno */
    HDBC con; /* conexión a la base de datos */

    SQLAllocEnv(&ent);
    SQLAllocConnect(ent, &con);
    SQLConnect(con, "db.yale.edu", SQL_NTS, "avi", SQL_NTS,
               "contraseñaAvi", SQL_NTS);
    {
        char nombresucursal[80];
        float saldo;
        int lenOut1, lenOut2;
        HSTMT stmt;

        char * consulta = "select nombre_sucursal, sum (saldo)
                           from cuenta
                           group by nombre_sucursal";
        SQLAllocStmt(con, &stmt);
        error = SQLExecDirect(stmt, consulta, SQL_NTS);
        if (error == SQL_SUCCESS) {
            SQLBindCol(stmt, 1, SQL_C_CHAR, nombresucursal , 80, &lenOut1);
            SQLBindCol(stmt, 2, SQL_C_FLOAT, &saldo, 0 , &lenOut2);
            while (SQLFetch(stmt) == SQL_SUCCESS) {
                printf (" %s %g\n", nombresucursal, saldo);
            }
        }
        SQLFreeStmt(stmt, SQL_DROP);
    }
    SQLDisconnect(con);
    SQLFreeConnect(con);
    SQLFreeEnv(ent);
}
```

Figura 4.4 Código de ejemplo ODBC.

de la longitud indica que el valor es **null**. Para los tipos de datos de longitud fija como integer o float, se ignora el argumento que indica la longitud máxima, mientras que la devolución de un valor negativo para el argumento de la longitud indica un valor nulo.

La instrucción SQLFetch se halla en un bucle **while** que se ejecuta hasta que SQLFetch devuelva un valor diferente de SQL_SUCCESS. En cada iteración, el programa almacena los valores en variables de C como se especifica mediante las llamadas a SQLBindCol e imprime esos valores.

Al final de la sesión, el programa libera el controlador de la instrucción, se desconecta de la base de datos y libera la conexión y los manejadores del entorno de SQL. Un buen estilo de programación exige que el resultado de cada llamada a una función se compruebe para asegurarse de que no haya errores; se han omitido la mayoría de estas comprobaciones en aras de la brevedad.

Es posible crear instrucciones SQL con parámetros; por ejemplo, considérese la instrucción insert into cuenta values(?,?,?). Los signos de interrogación representan los valores que se proporcionarán después. Esta instrucción se puede "preparar", es decir, compilar en la base de datos y ejecutar repetidamente proporcionando los valores reales para los parámetros—en este caso, proporcionando un número de cuenta, un nombre de sucursal y un saldo para la relación *cuenta*.

ODBC define funciones para gran variedad de tareas, tales como hallar todas las relaciones de la base de datos y los nombres y tipos de las columnas del resultado de una consulta o de una relación de la base de datos.

De forma predeterminada, cada instrucción SQL se trata como una transacción separada que se compromete automáticamente. La llamada SQLSetConnectOption(con, SQL_AUTOCOMMIT, 0) desactiva el compromiso automático en la conexión con, por lo que las transacciones se deben comprometer explícitamente mediante SQLTransact(con, SQL_COMMIT) o retroceder mediante SQLTransact(con, SQL_ROLLBACK).

La norma ODBC define *niveles de conformidad*, que especifican subconjuntos de la funcionalidad definida por la norma. Puede que una implementación de ODBC sólo proporcione las características básicas, o puede proporcionar características más avanzadas (de nivel 1 o de nivel 2). El nivel 1 exige soporte para la captura de información sobre el catálogo, como puede ser la información sobre las relaciones existentes y los tipos de sus atributos. El nivel 2 exige más características, como la capacidad de enviar y obtener arrays de valores de parámetros y la de obtener información del catálogo más detallada.

La norma de SQL define una **interfaz del nivel de llamadas** (CLI, Call Level Interface) que es parecida a la interfaz de ODBC. Las APIs ADO y ADO.NET son alternativas a ODBC, diseñadas para los lenguajes Visual Basic y C#; consúltense las notas bibliográficas si se desea más información.

La norma de **JDBC** define una API que pueden utilizar los programas de Java para conectarse a los servidores de bases de datos (la palabra JDBC era originalmente una abreviatura de **Java Database Connectivity**—conectividad de bases de datos con Java—, pero la forma completa ya no se emplea).

4.5.1.1 Apertura de conexiones y ejecución de consultas

La Figura 4.5 muestra un ejemplo de programa Java que utiliza la interfaz JDBC. El programa, en primer lugar, debe abrir una conexión con la base de datos y después puede ejecutar las instrucciones de SQL, pero antes de abrir la conexión, carga los controladores correspondientes para la base de datos empleando Class.forName. El primer parámetro de la llamada getConnection especifica el nombre de la máquina en la que se ejecuta el servidor (en este caso, db.yale.edu) y el número del puerto que emplea para las comunicaciones (en este caso, 2.000). El parámetro también especifica el esquema de la base de datos que se va a utilizar (en este caso, bdbanco), ya que cada servidor de bases de datos puede dar soporte a varios esquemas. El primer parámetro también especifica el protocolo que se va a utilizar para la comunicación con la base de datos (en este caso, jdbc:oracle:thin:). Obsérvese que JDBC especifica sólo la API, no el protocolo de comunicación. Los controladores de JDBC pueden dar soporte a varios protocolos y se debe especificar alguno compatible tanto con la base de datos como con el controlador. Los otros dos argumentos de **getConnection** son un identificador de usuario y una contraseña.

4.5.2 JDBC

El programa crea a continuación un manejador de instrucciones para la conexión y lo usa para ejecutar una instrucción SQL y obtener los resultados. En nuestro ejemplo, stmt.executeUpdate ejecuta una instrucción de actualización. El constructor try { . . . } catch { . . . } permite capturar cualquier excepción (condición de error) que surja cuando se realicen las llamadas JDBC, e imprime el mensaje correspondiente para el usuario.

El programa puede ejecutar una consulta mediante stmt.executeQuery. Puede recuperar el conjunto de filas del resultado en ResultSet y capturarlas tupla a tupla empleando la función next() en el conjunto de resultados. La Figura 4.5 muestra dos formas de recuperar los valores de los atributos de las tuplas: empleando el nombre del atributo (*nombre_sucursal*) y utilizando la posición del atributo (2, para denotar el segundo atributo).

La conexión se cierra al final del procedimiento. Obsérvese que es importante cerrar la conexión, ya que se ha impuesto un límite al número de conexiones con la base de datos; las conexiones no cerradas pueden hacer que ese límite se supere. Si esto ocurre, la aplicación no podrá abrir más conexiones con la base de datos.

```
public static void JDBCexample(String dbid, String idusuario, String contraseña)
{
    try
    {
        Class.forName (”oracle.jdbc.driver.OracleDriver”);
        Connection con = DriverManager.getConnection(
                "jdbc:oracle:thin:@db.yale.edu:2000:bdbanco",
                idusuario, contraseña);
        Statement stmt = con.createStatement();
        try {
            stmt.executeUpdate(
                "insert into cuenta values('C-9732', 'Navacerrada', 1200)");
        } catch (SQLException sqle)
        {
            System.out.println("No se pudo insertar la tupla. "+ sqle);
        }
        ResultSet rset = stmt.executeQuery(
                "select nombre_sucursal, avg (saldo)
                from cuenta
                group by nombre_sucursal");
        while (rset.next()) {
            System.out.println(rset.getString("nombre_sucursal") +  +
                    rset.getFloat(2));
        }
        stmt.close();
        con.close();
    }
    catch (SQLException sqle)
    {
        System.out.println("Excepción de SQL : "+ sqle);
    }
}
```

Figura 4.5 Un ejemplo de código JDBC.

4.5.2.1 Instrucciones preparadas

Se pueden crear instrucciones preparadas en las que algunos valores estén sustituidos por “?”, lo que indica que los valores reales se proporcionarán más tarde. Algunos sistemas de bases de datos compilan las consultas cuando se preparan; cada vez que se ejecuta la consulta (con valores nuevos), la base de datos puede volver a emplear la forma previamente compilada de la consulta. El fragmento de código de la Figura 4.6 muestra la manera de utilizar las instrucciones preparadas. La función setString (y otras funciones parecidas para otros tipos de datos básicos de SQL) permite especificar el valor de los parámetros.

Las instrucciones preparadas son el método preferido de ejecucion de las consultas de SQL, cuando utilizan valores que introducen los usuarios. Supóngase que el valor de las variables número_cuenta, nombre_sucursal, and saldo ha sido introducido por un usuario y hay que insertar la fila correspondiente en la relación *cuenta.* Supóngase que, en lugar de utilizar una instrucción preparada, se crea una consulta mediante la concatenación de las cadenas de caracteres de la manera siguiente:

```
"insert into cuenta values(' "+ número_cuenta + -', ' "+ nombre_sucursal + -', "
        + saldo + ")"
```

y la consulta se ejecuta directamente. Ahora, si el usuario escribiera una sola comilla en el campo del número de cuenta o en el del nombre de la sucursal, la cadena de caracteres de la consulta tendría un error

```
PreparedStatement pStmt = con.prepareStatement(
                    "insert into cuenta values(?,?,?)");
pStmt.setString(1, "C-9732");
pStmt.setString(2, "Navacerrada");
pStmt.setInt(3, 1200);
pStmt.executeUpdate();
pStmt.setString(1, "C-9733");
pStmt.executeUpdate();
```

Figura 4.6 Instrucciones preparadas en código JDBC.

de sintaxis. Es bastante probable que alguna sucursal tenga un apóstrofo en su nombre (especialmente si es un nombre irlandés como O'Donnell). Peor todavía, intrusos informáticos maliciosos pueden "inyectar" consultas de SQL propias escribiendo los caracteres adecuados en la cadena de caracteres. Una inyección de SQL de ese tipo puede dar lugar a graves fallos de seguridad.

La adición de caracteres de escape para tratar con los apóstrofos de las cadenas de caracteres es una manera de resolver este problema. El uso de las instrucciones preparadas es una manera más sencilla de hacerlo, ya que el método setString añade caracteres de escape de manera implícita. Además, cuando hay que ejecutar varias veces la misma instrucción con valores diferentes, las instrucciones preparadas suelen ejecutarse mucho más rápido que varias instrucciones de SQL por separado.

JDBC también proporciona una interfaz CallableStatement que permite la llamada a los procedimientos almacenados y a las funciones de SQL (que se describen más adelante, en el Apartado 4.6). Esta interfaz desempeña el mismo rol para las funciones y los procedimientos que prepareStatement para las consultas.

```
CallableStatement cStmt1 = con.prepareCall("{? = call alguna_función(?)}");
CallableStatement cStmt2 = con.prepareCall("{call algún_procedimiento(?,?)}");
```

Los tipos de datos de los valores devueltos por las funciones y de los parámetros externos de los procedimientos deben registrarse empleando el método registerOutParameter(), y pueden recuperarse utilizando métodos get parecidos a los de los conjuntos de resultados.

4.5.2.2 Metadatos

JDBC también proporciona mecanismos para examinar los esquemas de las bases de datos y determinar el tipo de datos de los atributos de los conjuntos de resultados. La interfaz ResultSet tiene un método getMetaData() para obtener objetos ResultSetMetaData y proporcionar los metadatos del conjunto de resultados. La interfaz ResultSetMetaData, a su vez, contiene métodos para determinar la información de los metadatos, como puede ser el número de columnas de un resultado, el nombre de una columna concreta o el tipo de datos de una columna dada. El programa de JDBC que se muestra a continuación ilustra el uso de la interfaz ResultSetMetaData para mostrar el nombre y el tipo de datos de todas las columnas de un conjunto de resultados. Se da por supuesto que la variable cr del siguiente código es un conjunto de resultados obtenido mediante la ejecución de una consulta.

```
ResultSetMetaData rsmd = cr.getMetaData();
for(int i = 1; i <= rsmd.getColumnCount(); i++) {
        System.out.println(rsmd.getColumnName(i));
        System.out.println(rsmd.getColumnTypeName(i));
}
```

El interfaz DatabaseMetaData ofrece una manera de determinar los metadatos de la base de datos. La interfaz Connection contiene un método getMetaData que devuelve el objeto DatabaseMetaData. La interfaz DatabaseMetaData, a su vez, contiene gran número de métodos para obtener los metadatos de la base de datos. El código de la Figura 4.7 ilustra la manera de obtener información sobre las columnas (atributos) de las relaciones de las bases de datos. Se da por supuesto que la variable con almacena una

```
DatabaseMetaData dbmd = con.getMetaData();
ResultSet sr = dbmd.getColumns(null, "bdbanco", "cuenta", "%");
      // Argumentos para getColumns: Catálogo, Patrón-esquema, Patrón-tabla y
      //            Patrón-columna
      // Devuelve: Una fila por columna; cada fila tiene varios atributos, como
      //            COLUMN_NAME (nombre de la columna), TYPE_NAME (nombre del tipo)
while( sr.next()) {
      System.out.println(sr.getString("COLUMN_NAME"), sr.getString("TYPE_NAME");
}
```

Figura 4.7 Búsqueda de información sobre las columnas en JDBC empleando DatabaseMetaData.

conexión con la base de datos ya abierta. El método getColumns toma cuatro argumentos: un nombre de catálogo (null significa que hay que ignorar el nombre del catálogo), un patrón para el nombre de los esquemas, un patrón para el nombre de las tablas y un patrón para el nombre de las columnas. El nombre del esquema, el de la tabla y los patrones del nombre de las columnas pueden utilizarse para especificar un nombre o un patrón. Los patrones pueden emplear los caracteres especiales para encaje de cadenas de caracteres de SQL "%" y "_"; por ejemplo, el patrón "%" encaja con todos los nombres.

Sólo se recuperan las columnas de las tablas de los esquemas que satisfacen el nombre o el patrón especificados. Cada fila del conjunto de resultados contiene información sobre una columna. Las filas tienen varias columnas, como pueden ser el nombre del catálogo, del esquema, de la tabla y de la columna, el tipo de datos de la columna, etc.

Otros métodos proporcionados por DatabaseMetaData permiten la recuperación de metadatos de las relaciones (getTables()), las referencias a claves externas (getCrossReference()), las autorizaciones, los límites de las bases de datos como el número máximo de conexiones, etc.

Las interfaces para metadatos pueden emplearse para gran variedad de tareas. Por ejemplo, se pueden utilizar par escribir un navegador de la base de datos que permita a los usuarios buscar las tablas de las bases de datos, examinar su esquema, examinar las filas de las tablas, aplicar selecciones para ver las filas deseadas, etc. La información sobre los metadatos puede utilizarse para crear código empleado para estas tareas genéricas; por ejemplo, se puede escribir código para mostrar las filas de las relaciones de manera que funcione en todas las relaciones posibles, independientemente de su esquema. De manera parecida, es posible escribir código que tome una cadena de caracteres de consulta, ejecute la consulta e imprima el resultado en forma de tabla con formato; el código puede funcionar independientemente de la consulta concreta que se remita.

4.5.2.3 Otras características

JDBC ofrece varias características más, como los **conjuntos de resultados actualizables**. Se pueden crear conjuntos de resultados actualizables a partir de consultas que lleven a cabo selecciones o proyecciones de las relaciones de las bases de datos. La actualización de una tupla del conjunto de resultados tiene como consecuencia que se actualice la tupla correspondiente de la relación de la base de datos.

De manera predeterminada, cada instrucción de SQL se trata como una transacción independiente que se compromete de manera automática. El método setAutoCommit() de la interfaz Connection de JDBC permite que este comportamiento se active o se desactive. Por tanto, si con es una variable que almacena una conexión abierta, conn.setAutoCommit(false) desactivará el compromiso automático. Las transacciones, entonces, deben comprometerse de manera explícita mediante con.commit() o retrocederse mediante con.rollback(). El compromiso automático puede activarse mediante con.setAutoCommit(true).

JDBC proporciona interfaces para el tratamiento de objetos de gran tamaño sin necesidad de crearlos completamente en la memoria. Para capturar objetos de gran tamaño la clase ResultSet proporciona los métodos getBlob() y getClob(), que son parecidos al método getString() pero devuelven objetos de los tipos de datos Blob y Clob, respectivamente. Estos objetos no almacenan los objetos de gran tamaño completos, sino que almacenan sus localizadores. Las clases Blob y Clob proporcionan métodos para recuperar objetos de gran tamaño en fragmentos más pequeños. También permiten que se almacenen objetos de gran tamaño en la base de datos; pueden asociarse con corrientes de datos de Java, que se

```
create function recuento_cuentas(nombre_cliente varchar(20))
    returns integer
    begin
    declare recuento_c integer;
        select count(*) into recuento_c
        from impositor
        where impositor.nombre_cliente = nombre_cliente
    return recuento_c;
    end
```

Figura 4.8 Función definida en SQL.

capturan de manera transparente y se envían a la base de datos en fragmentos de pequeño tamaño, por lo que no hace falta crear el objeto completo en la memoria.

JDBC también proporciona la clase RowSet, que ofrece las mismas características de ResultSet y varias características adicionales. Para obtener más información sobre JDBC, consúltese la información bibliográfica del final del capítulo.

4.6 Funciones y procedimientos**

Desde la versión SQL:1999, SQL permite la definición de funciones, procedimientos y métodos. Pueden definirse bien mediante el componente procedimental de SQL:1999 o bien mediante un lenguaje de programación externo como Java, C o C++. En primer lugar se examinarán las definiciones en SQL y luego se verá la manera de utilizar las definiciones en lenguajes externos.

Varios sistemas de bases de datos soportan sus propias extensiones procedimentales de SQL, como PL/SQL en Oracle y TransactSQL en Microsoft SQL Server. Estas extensiones recuerdan la parte procedimental de SQL, pero puede haber diferencias significativas en sintaxis y en semántica; consúltense los manuales de los sistemas respectivos para obtener más detalles.

4.6.1 Funciones y procedimientos de SQL

Supóngase que se desea una función que, dado el nombre de un cliente, devuelva el número de cuentas de las que es titular ese cliente. Se puede definir la función como se muestra en la Figura 4.8.

Esta función puede utilizarse en una consulta que devuelva el nombre y la dirección de todos los clientes con más de una cuenta:

```
select nombre_cliente, calle_cliente, ciudad_cliente
from cliente
where recuento_cuentas(nombre_cliente) > 1
```

Las funciones resultan particularmente útiles con tipos de datos especializados tales como las imágenes y los objetos geométricos. Por ejemplo, un tipo de datos de segmentos de recta empleado en las bases de datos cartográficas puede tener una función asociada que compruebe si se solapan dos segmentos, y un tipo de datos de imagen puede tener asociadas funciones para comparar la semejanza de dos imágenes. Las funciones se pueden escribir en un lenguaje externo como C, como se vio en el Apartado 4.6.3.

Desde la versión SQL:2003, SQL soporta las funciones que pueden devolver tablas como resultado; esas funciones se denominan **funciones de tabla**. Considérese la función definida en la Figura 4.9. Esa función devuelve una tabla que contiene todas las cuentas que tiene una persona dada. Obsérvese que se hace referencia a los parámetros de las funciones anteponiéndoles el nombre de la función (*cuentas_de.nombre_cliente*).

La función puede emplearse en consultas de la manera siguiente:

```
select *
from table(cuentas_de('Gómez'))
```

```
create function cuentas_de (nombre_cliente char(20))
    returns table (
        número_cuenta char(10),
        nombre_sucursal char(15),
        saldo numeric(12,2))
return table
    (select número_cuenta, nombre_sucursal, saldo
    from cuenta
    where exists (
        select *
        from impositor
        where impositor.nombre_cliente = cuentas_de.nombre_cliente and
            impositor.número_cuenta = cuenta.número_cuenta)
    )
```

Figura 4.9 Función de tabla en SQL.

Esta consulta devuelve todas las cuentas pertenecientes al cliente 'Gómez'. En el sencillo caso anterior, resulta fácil escribir esta consulta sin emplear funciones evaluadas sobre tablas. En general, sin embargo, las funciones evaluadas sobre tablas pueden considerarse **vistas paramétricas** que generalizan el concepto habitual de las vistas permitiendo la introducción de parámetros.

SQL:1999 también permite el uso de procedimientos. La función *recuento_cuentas* se puede escribir en forma de procedimiento:

```
create procedure proc_recuento_cuentas(in nombre_cliente varchar(20),
                                       out recuento_c integer)
begin
    select count(*) into recuento_c
    from impositor
    where impositor.nombre_cliente = proc_recuento_cuentas.nombre_cliente
end
```

Se pueden invocar los procedimientos mediante la instrucción **call** desde otro procedimiento de SQL o desde SQL incorporado:

```
declare recuento_cuentas integer;
call proc_recuento_cuentas('Gómez', recuento_c);
```

Los procedimientos y las funciones pueden invocarse desde SQL dinámico, como se ilustra mediante la sintaxis de JDBC en el Apartado 4.5.2.2.

SQL:1999 permite que haya más de un procedimiento con el mismo nombre, siempre que el número de argumentos de los procedimientos con el mismo nombre sea diferente. El nombre, junto con el número de argumentos, se utiliza para identificar los procedimientos. SQL permite también más de una función con el mismo nombre, siempre que las diferentes funciones con el mismo nombre tengan diferente número de argumentos o, para funciones con el mismo número de argumentos, se diferencien en el tipo de datos de un argumento, como mínimo.

4.6.2 Constructores procedimentales

Desde la versión SQL:1999, SQL soporta varios constructores procedimentales que le proporcionan casi toda la potencia de los lenguajes de programación de propósito general. La parte de la norma SQL que trata de estos constructores se denomina **módulo de almacenamiento persistente** (Persistent Storage Module, PSM).

El objetivo de PSM en SQL no es sustituir a los lenguajes de programación convencionales. Más bien, los constructores procedimentales permiten que se registre la "lógica de negocio" en forma de procedimientos almacenados de la base de datos y se ejecute en propia la base de datos. Por ejemplo, los bancos

suelen tener muchas reglas sobre la manera y el momento en que se pueden hacer los pagos a los clientes, como los límites máximos de retirada de efectivo, las exigencias de saldo mínimo, las facilidades para los descubiertos que permiten a los clientes retirar más dinero que el saldo disponible mediante la concesión automática de un préstamo, etc.

Aunque esa lógica de negocio puede codificarse en forma de procedimientos de los lenguajes de programación almacenados completamente fuera de las bases de datos, su definición como procedimientos almacenados de la base de datos presenta varias ventajas. Por ejemplo, permite que varias aplicaciones tengan acceso a los procedimientos, así como que baste con modificar un solo punto en caso de cambio de las reglas de negocio, sin que haga falta modificar la aplicación. El código de la aplicación puede llamar a los procedimientos almacenados, en lugar de actualizar directamente las relaciones de la base de datos.

Los constructores procedimentales son necesarios para permitir que se codifiquen como procedimientos almacenados las reglas de negocio complejas y, por tanto, se añadieron a SQL a partir de la versión SQL:1999 (ya las soportaban algunos productos de bases de datos incluso antes).

Las instrucciones compuestas son de la forma **begin** . . . **end**, y pueden contener varias instrucciones de SQL entre **begin** y **end**. En las instrucciones compuestas pueden declararse variables locales, como se ha visto en el Apartado 4.6.1.

SQL:1999 soporta las instrucciones **while** y **repeat** con la sintaxis siguiente:

```
declare n integer default 0;
while n < 10 do
    set n = n + 1;
end while
repeat
    set n = n − 1;
until n = 0
end repeat
```

Este código no lleva a cabo nada útil; simplemente pretende mostrar la sintaxis de los bucles **while** y **repeat**. Posteriormente se verán aplicaciones más significativas.

También hay un bucle **for** que permite la iteración por todos los resultados de una consulta:

```
declare n integer default 0;
for r as
        select saldo from cuenta
        where nombre_sucursal = 'Navacerrada'
do
    set n = n+ r.saldo
end for
```

El programa abre de manera implícita un cursor cuando comienza la ejecución del bucle **for** y lo utiliza para capturar los valores fila a fila en la variable de bucle **for** (*r*, en este ejemplo). Es posible ponerle nombre al cursor, insertando el texto *nc* **cursor for** justo detrás de la palabra clave **as**, donde *nc* es el nombre que se le desea dar al cursor. El nombre del cursor puede emplearse para llevar a cabo operaciones de actualización o borrado sobre la tupla a la que apunte el cursor. La instrucción **leave** se puede emplear para salir del bucle, mientras que **iterate** comienza con la siguiente tupla, a partir del comienzo del bucle, y se salta las instrucciones restantes

Entre las instrucciones condicionales soportadas por SQL están las instrucciones if-then-else con la sintaxis:

if *r.saldo* < 1000
  then set $l = l +$ *r.saldo*
elseif *r.saldo* < 5000
  then set $m = m +$ *r.saldo*
else set $h = h +$ *r.saldo*
end if

Este código da por supuesto que l, m y h son variables enteras y que r es una variable de fila. Si se sustituye la línea "**set** $n = n +$ *r.saldo*" del bucle **for** del párrafo anterior por el código **if-then-else** (y se proporcionan las declaraciones y los valores iniciales correspondientes de l, m y h), el bucle calculará los saldos totales de las cuentas que se hallan en las categorías de saldos bajo, medio y alto, respectivamente.

SQL también soporta instrucciones **case** parecidas a la instrucción **case** del lenguaje C/C++ (además de las expresiones case, que se vieron en el Capítulo 3).

Finalmente, SQL incluye los conceptos de señalización de las **condiciones de excepción** y de declarar los **manejadores** que pueden tratar esa excepción, como en este código:

declare *agotado* **condition**
declare exit handler for *agotado*
begin
. . .
end

Las instrucciones entre **begin** y **end** pueden provocar una excepción ejecutando **signal** *agotado*. El manejador dice que si se da la condición, la acción que hay que tomar es salir de la instrucción **begin end** circundante. Una acción alternativa sería **continue**, que continúa con la ejecución a partir de la siguiente instrucción que ha generado la excepción. Además de las condiciones definidas de manera explícita, también hay condiciones predefinidas como **sqlexception**, **sqlwarning** y **not found**.

La Figura 4.10 proporciona un ejemplo de mayor tamaño del uso de constructores procedimentales en SQL. El procedimiento *retirar* definido en la figura retira dinero de una cuenta y, si el saldo pasa a ser negativo, comienza el tratamiento del descubierto; no se muestra el código para el tratamiento del descubierto. La función devuelve un código de error; con un valor mayor o igual que cero indica éxito, mientras que un valor negativo indica una condición de error.

Otro ejemplo que ilustra los bucles **while** se presenta más adelante, en el Apartado 4.7.

4.6.3 Rutinas en otros lenguajes

SQL permite definir funciones en lenguajes de programación como Java, C#, C o C++. Las funciones definidas de esta manera pueden ser más eficientes que las definidas en SQL, y cálculos que no pueden llevarse a cabo en SQL pueden ejecutarse mediante estas funciones. Un ejemplo del uso de este tipo de funciones es llevar a cabo un cálculo aritmético complejo sobre los datos de una tupla.

Los procedimientos y funciones externos pueden especificarse de esta manera:

create procedure proc_recuento_cuentas(**in** *nombre_cliente* **varchar**(20),
                                out count **integer**)
language C
external name '/usr/avi/bin/proc_recuento_cuentas'

create function recuento_cuentas (*nombre_cliente* **varchar**(20))
returns integer
language C
external name '/usr/avi/bin/recuento_cuentas'

Los procedimientos del lenguaje externo tienen que tratar con los valores nulos y con las excepciones. Por tanto, deben tener varios parámetros adicionales: un valor **sqlstate** para indicar el estado de éxito o de fracaso, un parámetro para almacenar el valor devuelto por la función y variables indicadoras para el resultado de cada parámetro o función, para indicar si su valor es nulo. Una línea adicional **parameter**

```
create procedure retirar(
                in número_cuenta varchar(10)
                in importe numeric(12,2))
– – retira dinero de una cuenta
returns integer
begin
    declare nuevosaldo numeric(12,2);
    select saldo into nuevosaldo
    from cuenta
    where cuenta.número_cuenta = retirar.número_cuenta;
    nuevosaldo = nuevosaldo − importe;
    if (nuevosaldo < 0)
        begin
            ... el código para manejar el descubierto se inserta aquí
            ... si el importe es demasiado elevado para que lo maneje descubierto
            ... devuelve el código de error −1
        end
    else begin
    update cuenta
        set saldo = saldo − nuevosaldo
        where cuenta.número_cuenta = retirar.número_cuenta
    end
    return(0);
end
```

Figura 4.10 Procedimiento para la retirada de dinero de las cuentas.

style general añadida a la declaración anterior indica que los procedimientos o funciones externos sólo toman los argumentos mostrados y no tratan con valores nulos ni excepciones.

Las funciones definidas en un lenguaje de programación y compiladas fuera del sistema de base de datos pueden cargarse y ejecutarse con el código del sistema de bases de datos. Sin embargo, ello conlleva el riesgo de que un fallo del programa corrompa las estructuras internas de la base de datos y pueda saltarse la funcionalidad de control de accesos del sistema de bases de datos. Los sistemas de bases de datos que se preocupan más del rendimiento eficiente que de la seguridad pueden ejecutar los procedimientos de este modo. Los sistemas de bases de datos que están preocupados por la seguridad pueden ejecutar ese código como parte de un proceso diferente, comunicarle el valor de los parámetros y recuperar los resultados mediante la comunicación entre procesos. Sin embargo, el tiempo añadido que supone la comunicación entre los procesos es bastante elevado, en las arquitecturas de CPU habituales, decenas a centenares de millares de instrucciones se pueden ejecutar en el tiempo necesario para una comunicación entre los procesos.

Si el código está escrito en un lenguaje "seguro" como Java o C#, cabe otra posibilidad: ejecutar el código en un **cubo** dentro del propio proceso de ejecución de la consulta a la base de datos. El cubo permite que el código de Java o de C# tenga acceso a su propia área de memoria, pero evita que ese código lea o actualice la memoria del proceso de ejecución de la consulta, o tenga acceso a los archivos del sistema de archivos (la creación de cubos no es posible en lenguajes como C, que permite el acceso sin restricciones a la memoria mediante los punteros). El evitar la comunicación entre procesos reduce enormemente la sobrecarga de llamadas a funciones.

Varios sistemas de bases de datos actuales soportan que las rutinas de lenguajes externos se ejecuten en cubos dentro del proceso de ejecución de las consultas. Por ejemplo, Oracle y DB2 de IBM permiten que las funciones de Java se ejecuten como parte del proceso de la base de datos. SQL Server 2005 de Microsoft permite que los procedimientos compilados en CLR (Common Language Runtime) se ejecuten dentro del proceso de la base de datos; esos procedimientos pueden haberse escrito, por ejemplo, en C# o en Visual Basic.

nombre_empleado	*nombre_jefe*
Alández	Bariego
Bariego	Erice
Corisco	Dalma
Dalma	Santos
Erice	Santos
Santos	Marchamalo
Rienda	Marchamalo

Figura 4.11 La relación *jefe*.

4.7 Consultas recursivas**

Considérese una base de datos que contenga información sobre los empleados de una empresa. Supóngase que se tiene una relación *jefe*(*nombre_empleado, nombre jefe*), que especifica qué empleado es supervisado directamente por cada jefe. La Figura 4.11 muestra un ejemplo de la relación *jefe*.

Supóngase ahora que se desea determinar los empleados que son supervisados, directa o indirectamente por un jefe dado—por ejemplo, Santos. Es decir, se desea determinar los empleados supervisados directamente por Santos o por alguien que esté supervisado por Santos, o por alguien que esté supervisado por alguien que esté supervisado por Santos, y así sucesivamente.

Por tanto, si el jefe de Alández es Bariego y el jefe de Bariego es Erice y el jefe de Erice es Santos, entonces Alández, Bariego y Erice son los empleados supervisados por Santos.

4.7.1 Cierre transitivo mediante iteración

Una manera de escribir la consulta anterior es emplear la iteración: En primer lugar hay que determinar las personas que trabajan bajo Santos, luego las que trabajan bajo el primer conjunto, etc. La Figura 4.12 muestra la función *buscaEmpl*(*jef*) para llevar a cabo esa tarea; la función toma el nombre del jefe como parámetro (*jef*), calcula el conjunto de todos los subordinados directos e indirectos de ese jefe y devuelve el conjunto.

El procedimiento utiliza tres tablas temporales: *empl*, que se utiliza para almacenar el conjunto de tuplas que se va a devolver; *empnuevo*, que almacena los empleados localizados en la iteración anterior; y *temp*, que se utiliza como almacenamiento temporal mientras se manipulan los conjuntos de empleados. El procedimiento inserta todos los empleados que trabajan directamente para *jef* en *empnuevo* antes del bucle **repeat**. El bucle **repeat** añade primero todos los empleados de *empnuevo* a *empl*. Luego calcula los empleados que trabajan para los que están en *empnuevo*, excepto los que ya se sabe que son empleados de *jef* y los almacena en la tabla temporal *temp*. Finalmente, sustituye el contenido de *empnuevo* por el de *temp*. El bucle **repeat** termina cuando no encuentra más subordinados (indirectos) nuevos.

La Figura 4.13 muestra los empleados que se encontrarían en cada iteración si el procedimiento se llamara para el jefe llamado Santos.

Hay que destacar que el uso de la cláusula **except** en la función garantiza que la función trabaje incluso en el caso (anormal) de que haya un ciclo en la jerarquía de jefes. Por ejemplo, si *a* trabaja para *b*, *b* trabaja para *c* y *c* trabaja para *a*, hay un ciclo.

Aunque los ciclos pueden resultar poco realistas en el control de organigramas, son posibles en otras aplicaciones. Por ejemplo, supóngase que se tiene la relación *vuelos*(*a, desde*) que indica las ciudades a las que se puede llegar desde otra ciudad mediante un vuelo directo. Se puede escribir un código parecido al de la función *buscaEmpl* para determinar todas las ciudades a las que se puede llegar mediante una serie de uno o más vuelos desde una ciudad dada. Todo lo que hay que hacer es sustituir *jefe* por *vuelo* y el nombre de los atributos de manera acorde. En esta situación sí que puede haber ciclos de alcanzabilidad, pero la función trabajaría correctamente, ya que eliminaría las ciudades que ya se hubieran tomado en cuenta.

```
create function buscaEmpl(jef char(10))
    – – Busca todos los empleados que trabajan directa o indirectamente
    – – para jef
returns table (nombre char(10))
    – – La relación jefe(nombre_empleado, nombre_jefe)
    – – especifica quién trabaja directamente para quién.
begin
    create temporary table empl(nombre char(10));
        – – la tabla empl almacena el conjunto de empleados que
        – – se va a devolver
    create temporary table empnuevo(nombre char(10));
        – – la tabla empnuevo contiene los empleados hallados
        – – en la iteración anterior
    create temporary table temp(nombre char(10));
        – – la tabla temp es una tabla temporal utilizada
        – – para almacenar los resultados intermedios
    insert into empnuevo
        select nombre_empleado
        from jefe
        where nombre_jefe = jef
    repeat
        insert into empl
            select nombre
            from empnuevo;

        insert into temp
            (select jefe.nombre_empleado
                from empnuevo, jefe
                where empnuevo.nombre_empleado = jefe.nombre_jefe;
            )
            except (
                select nombre_empleado
                from empl
            );
        delete from empnuevo;
        insert into empnuevo
            select *
            from temp;
        delete from temp;

    until not exists (select * from empnuevo)
    end repeat;
    return table empl
end
```

Figura 4.12 Búsqueda de todos los empleados de cada jefe.

4.7.2 Recursión en SQL

El **cierre transitivo** de la relación *jefe* es una relación que contiene todos los pares (*emp*, *jef*) tales que *emp* es subordinado directo o indirecto de *jef*. Hay numerosas aplicaciones que exigen el cálculo de cierres transitivos parecidos sobre las **jerarquías**. Por ejemplo, las organizaciones suelen constar de varios niveles de unidades organizativas. Las máquinas constan de componentes que, a su vez, tienen subcomponentes, etc.; por ejemplo, una bicicleta puede tener subcomponentes como las ruedas y los pedales

Número de iteración	*Tuplas de empl*
0	
1	(Dalma), (Erice)
2	(Dalma), (Erice), (Bariego), (Corbacho)
3	(Dalma), (Erice), (Bariego), (Corbacho), (Alández)
4	(Dalma), (Erice), (Bariego), (Corbacho), (Alández)

Figura 4.13 Empleados de Santos en las iteraciones de la función *buscaEmpl.*

que, a su vez, tienen subcomponentes como las cámaras, las llantas y los radios. El cierre transitivo puede utilizarse sobre esas jerarquías para determinar, por ejemplo, todos los componentes de la bicicleta.

Resulta muy poco práctico especificar el cierre transitivo empleando la iteración. Hay un enfoque alternativo, el uso de las definiciones recursivas de las vistas, que resulta más sencillo de utilizar.

Se puede emplear la recursión para definir el conjunto de empleados controlados por un jefe determinado, digamos Santos, del modo siguiente. Las personas supervisadas (directa o indirectamente) por Santos son:

1. Personas cuyo jefe es Santos.
2. Personas cuyo jefe está supervisado (directa o indirectamente) por Santos.

Obsérvese que el caso 2 es recursivo, ya que define el conjunto de personas supervisadas por Santos en términos del conjunto de personas supervisadas por Santos. Otros ejemplos de cierre transitivo, como la búsqueda de todos los subcomponentes (directos o indirectos) de un componente dado también puede definirse de manera parecida, recursivamente.

Desde la versión SQL:1999 la norma de SQL soporta una forma limitada de recursión, que emplea la cláusula **with recursive**, en la que las vistas (o las vistas temporales) se expresan en términos de sí mismas. Se pueden utilizar consultas recursivas, por ejemplo, para expresar de manera concisa el cierre transitivo. Recuérdese que la cláusula **with** se emplea para definir una vista temporal cuya definición sólo está disponible para la consulta en la que se define. La palabra clave adicional **recursive** especifica que la vista es recursiva.

Por ejemplo, se pueden buscar todos los pares (*emp*, *jef*) tales que *emp* esté directa o indirectamente mandado por *jef*, empleando la vista recursiva de SQL que aparece en la Figura 4.14.

Todas las vistas recursivas deben definirse como la unión de dos subconsultas: una **consulta básica** que no es recursiva y una **consulta recursiva** que utiliza la vista recursiva. En el ejemplo de la Figura 4.14 la consulta básica es la selección sobre *jefe*, mientras que la consulta recursiva calcula la reunión de *jefe* y *empl*.

El significado de las vistas recursivas se comprende mejor de la manera siguiente. En primer lugar hay que calcular la consulta básica y añadir todas las tuplas resultantes a la vista (que se halla inicialmente vacía). A continuación hay que calcular la consulta recursiva empleando el contenido actual de la vista y volver a añadir todas las tuplas resultantes a la vista. Hay que seguir repitiendo este paso hasta que no

```
with recursive empl(nombre_empleado, nombre_jefe) as (
        select nombre_empleado, nombre_jefe
        from jefe
    union
        select jefe.nombre_empleado, empl.nombre_jefe
        from jefe, empl
        where jefe.nombre_jefe = empl.nombre_empleado
    )
select ∗
from empl
```

Figura 4.14 Consulta recursiva en SQL.

se añada ninguna tupla nueva a la vista. La instancia resultante de la vista se denomina **punto fijo** de la definición recursiva de la vista. (El término "fijo" hace referencia al hecho de que ya no se producen más modificaciones). La vista, por tanto, se define para que contenga exactamente las tuplas de la instancia del punto fijo.

Aplicando la lógica anterior a nuestro ejemplo, se hallan primero todos los subordinados directos de cada jefe ejecutando la consulta básica. La consulta recursiva añade un nivel de empleados en cada iteración, hasta alcanzar la profundidad máxima de la relación jefe-empleado. En ese punto ya no se añaden nuevas tuplas a la vista y la iteración ha alcanzado un punto fijo.

Obsérvese que no se exige que el sistema de bases de datos utilice esta técnica iterativa para calcular el resultado de la consulta recursiva; puede obtener el mismo resultado empleando otras técnicas que pueden ser más eficientes.

Hay algunas restricciones para la consulta recursiva de la vista recursiva, concretamente, la consulta debe ser **monótona**, es decir, su resultado sobre la instancia de una relación vista V_1 debe ser un superconjunto de su resultado sobre la instancia de una relación vista V_2 si V_1 es un superconjunto de V_2. De manera intuitiva, si se añaden más tuplas a la relación vista, la consulta recursiva debe devolver, como mínimo, el mismo conjunto de tuplas que antes, y posiblemente devuelva tuplas adicionales.

En concreto, las consultas recursivas no deben emplear ninguno de los constructores siguientes, ya que harían que la consulta no fuera monótona:

- Agregación sobre la vista recursiva.
- **not exists** sobre una subconsulta que utiliza la vista recursiva.
- Diferencia de conjuntos (**except**) cuyo lado derecho utilice la vista recursiva.

Por ejemplo, si la consulta recursiva fuera de la forma $r - v$, donde v es la vista recursiva, si se añade una tupla a v, el resultado de la consulta puede hacerse más pequeño; la consulta, por tanto, no es monótona.

El significado de las vistas recursivas puede definirse mediante el procedimiento recursivo mientras la consulta recursiva sea monótona; si la consulta recursiva no es monótona, el significado de la consulta resulta difícil de definir. SQL, por tanto, exige que las consultas sean monótonas. Las consultas recursivas se estudian con más detalle en el contexto del lenguaje de consultas Datalog, en el Apartado 5.4.6.

SQL también permite la creación de vistas permanentes definidas de manera recursiva mediante el uso de **create recursive view**, en lugar de **with recursive**. Algunas implementaciones soportan consultas recursivas que emplean una sintaxis diferente; consúltese el manual del sistema correspondiente para conocer más detalles.

4.8 Características avanzadas de SQL**

El lenguaje SQL ha crecido durante las dos últimas décadas desde ser un lenguaje sencillo con pocas características a ser un lenguaje bastante complejo con características para satisfacer a muchos tipos diferentes de usuarios. En este capítulo ya se han tratado los fundamentos de SQL. En este apartado se presentan al lector algunas de las características más complejas de SQL. Muchas de estas características se han introducido en versiones de la norma de SQL relativamente recientes y puede que no las soporten todos los sistemas de bases de datos.

4.8.1 Creación de tablas a partir de otras

Las aplicaciones necesitan a menudo la creación de tablas que tengan el mismo esquema que otra tabla ya existente. SQL proporciona la extensión **create table like** para soportar esta tarea:

create table *cuenta_temp* **like** *cuenta*

La instrucción anterior crea una nueva tabla *cuenta_temp* que tiene el mismo esquema que *cuenta*.

Al escribir consultas complejas suele resultar útil almacenar el resultado de la consulta en forma de nueva tabla; esa tabla suele ser temporal. Hacen falta dos instrucciones, una para crear la tabla (con las columnas correspondientes) y otra para insertar el resultado de la consulta en la tabla. SQL:2003

proporciona una técnica más sencilla para crear una tabla que contenga el resultado de una consulta. Por ejemplo, la instrucción siguiente crea la tabla *t1*, que contiene el resultado de una consulta.

```
create table t1 as
    (select *
    from cuenta
    where nombre_sucursal = 'Navacerrada')
with data
```

De manera predeterminada, el nombre y el tipo de datos de las columnas se deduce del resultado de la consulta. Se puede poner nombre a las columnas de manera explícita escribiendo la lista de los nombres de las columnas tras el nombre de la relación. Si se omite la cláusula **with data**, la tabla se crea, pero no se rellena con datos.

La instrucción **create table ... as** anterior se parece mucho a la instrucción **create view**, y las dos se definen mediante consultas. La principal diferencia es que el contenido de la tabla se define al crearla, mientras que el contenido de una vista siempre refleja el resultado actual de la consulta.

Hay que tener en cuenta que varias implementaciones soportan la funcionalidad de **create table ... like** y **create table ... as** pero emplean sintaxis diferentes; consúltese el manual del sistema respectivo para obtener más detalles.

4.8.2 Otros aspectos sobre las subconsultas

SQL:2003 permite que las subconsultas se produzcan siempre que haga falta un valor, siempre y cuando la consulta sólo devuelva un valor; estas subconsultas se denominan **subconsultas escalares**. Por ejemplo, se puede utilizar una subconsulta en la cláusula **select** como se ilustra en el ejemplo siguiente, que genera una lista de todos los clientes y del número de cuentas que tienen:

```
select nombre_cliente,
        (select count(*)
         from cuenta
         where cuenta.nombre_cliente = cliente.nombre_cliente) as número_cuentas
from cliente
```

Está garantizado que la subconsulta del ejemplo anterior sólo devuelva un único valor, ya que tiene un agregado **count**(*) sin el correspondiente **group by**. Las subconsultas sin agregados también están permitidas. Estas consultas pueden, en teoría, devolver más de una respuesta; si lo hacen, se produce un error en tiempo de ejecución.

Las subconsultas de la cláusula **select** de la consulta exterior pueden tener acceso a los atributos de las relaciones de la cláusula **from** de la consulta exterior, como *cliente.nombre_cliente* en este ejemplo.

Las subconsultas de la cláusula **from** (que ya se ha estudiado en el Apartado 3.8.1), sin embargo, normalmente no pueden tener acceso a los atributos de otras relaciones de la cláusula **from**; SQL:2003 soporta la cláusula **lateral**, que permite que las subconsultas de la cláusula **from** tengan acceso a los atributos de las subconsultas anteriores de esa misma cláusula **from**. Por tanto, la consulta anterior puede escribirse también del modo siguiente

```
select nombre_cliente, número_cuentas
    from cliente,
        lateral(select count(*)
                from cuenta
                where cuenta.nombre_cliente = cliente.nombre_cliente)
            as este_cliente(número_cuentas)
```

4.8.3 Constructores avanzados para la actualización de las bases de datos

Supóngase que se tiene la relación *fondos_recibidos*(*número_cuenta*, *importe*) que almacena los fondos recibidos (digamos, mediante transferencias electrónicas) para cada cuenta de un grupo. Supóngase ahora

que se desea añadir los importes a los saldos de las cuentas correspondientes. Para utilizar la actualización **update** de SQL para llevar a cabo esta tarea hay que examinar la tabla *fondos_recibidos* para cada tupla de la tabla *cuenta*. Se pueden utilizar subconsultas en la cláusula update para llevar a cabo esta tarea, como se muestra a continuación. En aras de la sencillez se supone que la relación *fondos_recibidos* contiene, como máximo, una tupla por cada cuenta.

```
update cuenta set saldo = saldo +
                (select importe
                 from fondos_recibidos
                 where fondos_recibidos.número_cuenta = cuenta.número_cuenta)
    where exists(
                select *
                from fondos_recibidos
                where fondos_recibidos.número_cuenta = cuenta.número _cuenta)
```

Téngase en cuenta que la condición de la cláusula **where** de la actualización garantiza que sólo se actualicen las cuentas con sus tuplas correspondientes en *fondos_recibidos*, mientras que la subconsulta de la cláusula **set** calcula el montante que hay que añadir a cada una de esas cuentas.

Hay muchas aplicaciones que necesitan actualizaciones como la que se acaba de ilustrar. Generalmente hay una tabla, que se puede llamar **tabla maestra**, y las actualizaciones de la tabla maestra se reciben por lotes. Luego hay que actualizar a su vez la tabla maestra. SQL:2003 proporciona un constructor especial, denominado **merge**, para simplificar la labor de mezcla de toda esa información. Por ejemplo, la actualización anterior puede expresarse empleando **merge** de la manera siguiente:

```
merge into cuenta as C
using    (select *
          from fondos_recibidos) as F
     on (C.número_cuenta = F.número_cuenta)
when matched then
     update set saldo = saldo+F.importe
```

Cuando un registro de la subconsulta de la cláusula **using** coincide con un registro de la relación *cuenta*, se ejecuta la cláusula **when matched**, que puede ejecutar una actualización de la relación; en este caso, el registro coincidente de la relación *cuenta* se actualiza de la manera mostrada.

La instrucción **merge** también puede tener una cláusula **when not matched then**, que permite la inserción de registros nuevos en la relación. En el ejemplo anterior, cuando no hay cuenta coincidente para alguna tupla de *fondos_recibidos*, la acción de inserción puede crear un nuevo registro de *cuenta* (con un valor nulo para *nombre_sucursal*) utilizando la cláusula siguiente.

```
when not matched then
     insert values (F.número_cuenta, null, F.importe)
```

Aunque no sea muy significativo en este ejemplo[2], la cláusula **when not matched clause** puede resultar bastante útil en otros casos. Por ejemplo, supóngase que la relación local es copia de una relación maestra y se reciben de la relación maestra registros actualizados y otros recién insertados. La instrucción **merge** puede actualizar los registros coincidentes (serán registros antiguos actualizados) e insertar los registros que no coincidan (serán registros nuevos).

No todas las implementaciones de SQL soportan actualmente la instrucción **merge**; consúltese el manual del sistema correspondiente para obtener más detalles.

2. Una solución mejor en este caso habría sido insertar estos registros en una relación de error, pero eso no se puede hacer con la instrucción **merge**.

4.9 Resumen

- El lenguaje de definición de datos de SQL ofrece soporte para la definición de tipos predefinidos como la fecha y la hora, así como para tipos de dominios definidos por los usuarios.
- Las restricciones de dominio especifican el conjunto de valores posibles que pueden asociarse con cada atributo. Estas restricciones también pueden prohibir el uso de valores nulos para atributos concretos.
- Las restricciones de integridad referencial aseguran que un valor que aparece en una relación para un conjunto de atributos dado aparezca también para un conjunto de atributos concreto en otra relación.
- Los asertos son expresiones declarativas que establecen predicados que necesitamos que siempre sean ciertos.
- Cada usuario puede tener varias formas de autorización para diferentes partes de la base de datos. La autorización es un medio de proteger el sistema de bases de datos contra el acceso malintencionado o no autorizado.
- Las consultas SQL se pueden invocar desde lenguajes anfitriones mediante SQL incorporado y SQL dinámico. Las normas ODBC and JDBC definen interfaces para programas de aplicación para el acceso a las bases de datos de SQL desde programas en los lenguajes C y Java. Los programadores utilizan cada vez más estas APIs para el acceso a las bases de datos.
- Las funciones y los procedimientos pueden definirse empleando SQL. También se han descrito las extensiones procedimentales proporcionadas por SQL:1999, que permiten la iteración y las instrucciones condicionales (if-then-else).
- Algunas consultas, como el cierre transitivo, pueden expresarse tanto mediante la iteración com mediante las consultas recursivas de SQL. La recursión puede expresarse mediante las vistas recursivas o mediante cláusulas **with** recursivas.
- También se ha visto una breve introducción de algunas características avanzadas de SQL, que simplifican determinadas tareas relacionadas con la definición de datos y la consulta y la actualización de los datos.

Términos de repaso

- Tipos definidos por los usuarios.
- Dominios.
- Objetos de gran tamaño.
- Catálogos.
- Esquemas.
- Restricciones de integridad.
- Restricciones de dominios.
- Restricción de unicidad.
- Cláusula **check**.
- Integridad referencial.
- Restricción de clave primaria.
- Restricción de clave externa.
- Borrados en cascada.
- Actualizaciones en cascada.
- Asertos.
- Autorizaciones.
- Privilegios:
  - **select**.
  - **insert**.
  - **update**.
  - **delete**.
  - **all privileges**.
- Concesión de privilegios.
- Retirada de privilegios.
- SQL incorporado.
- Cursores.
- Cursores actualizables.
- SQL dinámico.
- ODBC.
- JDBC.
- Instrucciones preparadas.

- Acceso a los metadatos.
- Funciones de SQL.
- Procedimientos almacenados.
- Constructores procedimentales.
- Rutinas de otros lenguajes.
- Consultas recursivas.
- Consultas monótonas.
- Instrucción **merge**.

Ejercicios prácticos

4.1 Complétese la definición del LDD de SQL de la base de datos bancaria de la Figura 4.2 para que incluya las relaciones *préstamo* y *prestatario*.

4.2 Considérese la siguiente base de datos relacional:

empleado(*nombre_empleado*, *calle*, *ciudad*)
trabaja(*nombre_empleado*, *nombre_empresa*, *sueldo*)
empresa(*nombre_empresa*, *ciudad*)
jefe(*nombre_empleado*, *nombre_jefe*)

Dese una definición de esta base de datos en el LDD de SQL. Identifíquense las restricciones de integridad referencial que deben cumplirse e inclúyanse en la definición del LDD.

4.3 Escríbanse condiciones **check** para el esquema que se ha definido en el Ejercicio 4.2 para garantizar que:

a. Cada empleado trabaje para una empresa ubicada en su ciudad de residencia.
b. Ningún empleado gane un sueldo mayor que el de su jefe.

4.4 SQL permite que las dependencias de clave externa hagan referencia a la misma relación, como en el ejemplo siguiente:

```
create table jefe
   (nombre_empleado char(20)  not null
    nombre_jefe     char(20)  not null,
    primary key nombre_empleado,
    foreign key (nombre_jefe) references jefe
                              on delete cascade )
```

Aquí, *nombre_empleado* es clave de la tabla *jefe*, lo que significa que cada empleado tiene como máximo un director. La cláusula de clave externa exige que cada director sea también empleado. Explíquese exactamente lo que ocurre cuando se borra una tupla de la relación *jefe*.

4.5 Escríbase un aserto para la base de datos bancaria que garantice que el valor de los activos de la sucursal de Navacerrada sea igual a la suma de todos los importes prestados por esa sucursal.

4.6 Descríbanse las circunstancias bajo las cuales es preferible utilizar SQL incorporado en lugar de SQL solo o un lenguaje de programación de propósito general sin más.

Ejercicios

4.7 Las restricciones de integridad referencial tal y como se han definido en este capítulo implican exactamente a dos relaciones. Considérese una base de datos que incluye las relaciones siguientes:

trabajador_jornada_completa(*nombre, despacho, teléfono, sueldo*)
trabajador_tiempo_parcial(*nombre, sueldo_por_hora*)
dirección(*nombre, calle, ciudad*)

Supóngase que se desea exigir que cada nombre que aparece en *dirección* aparezca también en *trabajador_jornada_completa* o en *trabajador_tiempo_parcial*, pero no necesariamente en ambos.

a. Propóngase una sintaxis para expresar estas restricciones.
b. Explíquense las acciones que debe realizar el sistema para aplicar una restricción de este tipo.

4.8 Escríbase una función de Java que emplee las características para metadatos de JDBC y tome un ResultSet como parámetro de entrada e imprima el resultado en forma tabular, con los nombres correspondientes como encabezados de las columnas.

4.9 Escríbase una función de Java que emplee las características para metadatos de JDBC e imprima una lista de todas las relaciones de la base de datos, mostrando para cada relación el nombre y el tipo de datos de los atributos.

4.10 Considérese una base de datos de empleados con dos relaciones

empleado(*nombre_empleado*, *calle*, *ciudad*)
trabaja(*nombre_empleado*, *nombre_empresa*, *sueldo*)

en las que se han subrayado las claves primarias. Escríbase una consulta para determinar las empresas cuyos empleados ganan sueldos más altos, en media, que el sueldo medio de Banco Importante.

a. Empleando las funciones de SQL necesarias.
b. Sin emplear las funciones de SQL.

4.11 Reescríbase la consulta del Apartado 4.6.1 que devuelve el nombre, la calle y la ciudad de todos los clientes con más de una cuenta en el banco empleando la cláusula **with** en lugar de la llamada a una función.

4.12 Compárese el uso de SQL incrustado con el uso en SQL de funciones definidas en lenguajes de programación de propósito general. ¿En qué circunstancias es preferible utilizar cada una de estas características?

4.13 Modifíquese la consulta recursiva de la Figura 4.14 para definir la relación

profundidad_empleado(*nombre_empleado*, *nombre_jefe*, *profundidad*)

en la que el atributo *profundidad* indica el número de niveles de jefes intermedios que hay entre el empleado y su jefe. Los empleados que se hallan directamente bajo un jefe tienen una profundidad de 0.

4.14 Considérese el esquema relacional

componente(*id_componente*, *nombre*, *coste*)
subcomponente(*id_componente*, *id_subcomponente*, *recuento*)

La tupla $(c_1, c_2, 3)$ de la relación *subcomponente* denota que el componente con identificador de componente c_2 es un subcomponente directo del componente con identificador de componente c_1, y que c_1 tiene 3 copias de c_2. Téngase en cuenta que c_2 puede, a su vez, tener más subcomponentes. Escríbase una consulta recursiva de SQL que genere el nombre de todos los subcomponentes del componente con identificador "C-100".

4.15 Considérese nuevamente el esquema relacional del Ejercicio 4.14. Escríbase una función de JDBC que utilice SQL no recursivo para determinar el coste total del componente "C-100", incluido el coste de todos sus subcomponentes. Hay que asegurarse de tener en cuenta el hecho de que cada componente puede tener varios ejemplares de un subcomponente. Se puede utilizar la recursión en Java si se desea.

Notas bibliográficas

Véanse las notas bibliográficas del Capítulo 3 para consultar las referencias a las normas de SQL y a los libros sobre SQL.

Muchos productos de bases de datos soportan más características de SQL que las especificadas en las normas, y puede que no soporten algunas características recogidas en ellas. Se puede conseguir más información sobre estas características en el manual del producto correspondiente.

java.sun.com/docs/books/tutorial es una excelente fuente de información actualizada sobre JDBC, y sobre Java en general. Las referencias a los libros sobre Java (incluido JDBC) también están disponibles en este URL. El API de ODBC se describe en Microsoft [1997] y en Sanders [1998]. Melton y Eisenberg [2000] proporcionan una guía de SQLJ, JDBCy las tecnologías relacionadas. Se puede encontrar más información sobre ODBC, ADO y ADO.NET en msdn.microsoft.com/data.

CAPÍTULO 5

Otros lenguajes relacionales

En el Capítulo 2 se ha descrito el álgebra relacional que constituye la base de SQL el lenguaje de consultas más usado. SQL se ha tratado con gran detalle en los Capítulos 3 y 4. En este capítulo se estudiarán primero otros dos lenguajes formales, el cálculo relacional de tuplas y el cálculo relacional de dominios, que son lenguajes de consultas declarativos basados en la lógica matemática. Estos dos lenguajes formales constituyen la base de dos lenguajes más fáciles de usar, QBE y Datalog, que se estudian más adelante en este mismo capítulo.

A diferencia de SQL, QBE es un lenguaje gráfico en el que las consultas *parecen* tablas. QBE y sus variantes se usan mucho en sistemas de bases de datos para computadoras personales. Datalog tiene una sintaxis derivada del lenguaje Prolog. Aunque actualmente no se usa de forma comercial, Datalog se ha usado en varios sistemas de bases de datos en investigación.

Para QBE y Datalog se presentan los constructores y los conceptos fundamentales en lugar de completos manuales de usuario. Hay que tener presente que las diferentes implementaciones de cada lenguaje pueden diferir en los detalles, o sólo dar soporte a un subconjunto del lenguaje completo.

5.1 El cálculo relacional de tuplas

Cuando se escribe una expresión del álgebra relacional se proporciona una serie de procedimientos que generan la respuesta a la consulta. El cálculo relacional de tuplas, en cambio, es un lenguaje de consultas **no procedimental**. Describe la información deseada sin dar un procedimiento concreto para obtenerla.

Las consultas se expresan en el cálculo relacional de tuplas como

$$\{t \mid P(t)\}$$

es decir, es el conjunto de todas las tuplas t tales que el predicado P es cierto para t. Siguiendo la notación usada anteriormente, se usa $t[A]$ para denotar el valor de la tupla t en el atributo A y $t \in r$ para denotar que la tupla t está en la relación r.

Antes de dar una definición formal del cálculo relacional de tuplas se volverán a examinar algunas de las consultas para las que se escribieron expresiones del álgebra relacional en el Apartado 2.2. Recuérdese que las consultas se realizan sobre el esquema siguiente:

sucursal(*nombre_sucursal*, *ciudad_sucursal*, *activos*)
cliente (*nombre_cliente*, *calle_cliente*, *ciudad_cliente*)
préstamo (*número_préstamo*, *nombre_sucursal*, *importe*)
prestatario (*nombre_cliente*, *número_préstamo*)
cuenta (*número_cuenta*, *nombre_sucursal*, *saldo*)
impositor (*nombre_cliente*, *número_cuenta*)

5.1.1 Ejemplos de consultas

Determinar el *nombre_sucursal*, *número_préstamo* e *importe* de los préstamos de más de 1.200 €:

$$\{t \mid t \in préstamo \wedge t[importe] > 1200\}$$

Supóngase que sólo se desea obtener el atributo *número_préstamo*, en vez de todos los atributos de la relación *préstamo*. Para escribir esta consulta en cálculo relacional de tuplas hay que incluir una expresión de relación sobre el esquema (*número_préstamo*). Se necesitan las tuplas de (*número_préstamo*) tales que hay una tupla de *préstamo* con el atributo *importe* $>$ 1200. Para expresar esta solicitud hay que usar el constructor "existe" de la lógica matemática. La notación

$$\exists t \in r\ (Q(t))$$

significa "existe una tupla *t* de la relación *r* tal que el predicado *Q* (*t*) es cierto".

Usando esta notación se puede escribir la consulta "Determinar el número de préstamo de todos los préstamos con importe superior a 1.200 €" como:

$$\{t \mid \exists s \in préstamo\ (t[número_préstamo] = s[número_préstamo] \wedge s[importe] > 1200)\}$$

En español la expresión anterior se lee "el conjunto de todas las tuplas *t* tales que existe una tupla *s* de la relación *préstamo* para la que los valores de *t* y de *s* para el atributo *número_préstamo* son iguales y el valor de *s* para el atributo *importe* es mayor que 1.200 €".

La variable tupla *t* sólo se define para el atributo *número_préstamo*, dado que es el único atributo para el que se especifica una condición para *t*. Por tanto, el resultado es una relación de (*número_préstamo*).

Considérese la consulta "Determinar el nombre de todos los clientes que tienen concedido un préstamo en la sucursal de Navacerrada". Esta consulta es un poco más compleja que las anteriores porque implica a dos relaciones: *prestatario* y *préstamo*. Como se verá, sin embargo, todo lo que necesita es que tengamos dos cláusulas "existe" en la expresión del cálculo relacional de tuplas, relacionadas mediante *y* ($\wedge$). La consulta se escribe de la manera siguiente:

$$\{t \mid \exists s \in prestatario\ (t[nombre_cliente] = s[nombre_cliente] \wedge \exists u \in préstamo\ (u[número_préstamo] = s[número_préstamo] \wedge u[nombre_sucursal] = \text{"Navacerrada"}))\}$$

En español esta expresión es "el conjunto de todas las tuplas (*nombre_cliente*) tales que el cliente tiene un préstamo concedido en la sucursal de Navacerrada". La variable tupla *u* garantiza que el cliente sea prestatario de la sucursal de Navacerrada. La variable tupla *s* está restringida para que corresponda al mismo número de préstamo que *s*. El resultado de esta consulta se muestra en la Figura 5.1.

Para determinar todos los clientes del banco que tienen concedido un préstamo, abierta una cuenta o ambas cosas, se usó la operación unión del álgebra relacional. En el cálculo relacional de tuplas serán necesarias dos instrucciones "existe" conectadas mediante *o* ($\vee$):

$$\{t \mid \exists s \in prestatario\ (t[nombre_cliente] = s[nombre_cliente]) \vee \exists u \in impositor\ (t[nombre_cliente] = u[nombre_cliente])\}$$

Esta expresión da el conjunto de todas las tuplas de *nombre_cliente* tales que se cumple al menos una de las condiciones siguientes:

nombre_cliente
Fernández
López

Figura 5.1 Nombre de todos los clientes que tienen concedido un préstamo en la sucursal de Navacerrada.

- *nombre_cliente* aparece en alguna tupla de la relación *prestatario* como prestatario del banco.
- *nombre_cliente* aparece en alguna tupla de la relación *impositor* como impositor del banco.

Si algún cliente tiene concedido un préstamo y abierta una cuenta en el banco, ese cliente sólo aparece una vez en el resultado, ya que la definición matemática de conjunto no permite elementos duplicados. El resultado de esta consulta ya se mostró en la Figura 2.11.

Si *sólo* queremos conocer los clientes que tienen en el banco *tanto* una cuenta *como* un préstamo, todo lo que hay que hacer es cambiar en la expresión anterior la *o* ($\vee$) por una *y* ($\wedge$) en la expresión anterior.

$$\{t \mid \exists\, s \in \mathit{prestatario}\ (t[\mathit{nombre_cliente}] = s[\mathit{nombre_cliente}]) \\ \wedge \exists\, u \in \mathit{impositor}\ (t[\mathit{nombre_cliente}] = u[\mathit{nombre_cliente}])\}$$

El resultado de esta consulta se mostró en la Figura 2.19.

Considérese ahora la consulta "Determinar todos los clientes que tienen una cuenta abierta en el banco pero no tienen concedido ningún préstamo". La expresión del cálculo relacional de tuplas para esta consulta es parecida a las que se acaban de ver, salvo en el uso del símbolo *no* ($\neg$):

$$\{t \mid \exists\, u \in \mathit{impositor}\ (t[\mathit{nombre_cliente}] = u[\mathit{nombre_cliente}]) \\ \wedge \neg \exists\, s \in \mathit{prestatario}\ (t[\mathit{nombre_cliente}] = s[\mathit{nombre_cliente}])\}$$

Esta expresión del cálculo relacional de tuplas usa la cláusula $\exists\, u \in \mathit{impositor}\ (\ldots)$ para exigir que el cliente tenga una cuenta abierta en el banco, y la cláusula $\neg \exists\, s \in \mathit{prestatario}\ (\ldots)$ para eliminar los clientes que aparecen en alguna tupla de la relación *prestatario* por tener un préstamo del banco. El resultado de esta consulta apareció en la Figura 2.12.

La consulta que se tomará ahora en consideración usa la implicación, denotada por $\Rightarrow$. La fórmula $P \Rightarrow Q$ significa "*P* implica *Q*"; es decir, "si *P* es cierto, entonces *Q* tiene que serlo también". Obsérvese que $P \Rightarrow Q$ es equivalente lógicamente a $\neg P \vee Q$. El empleo de la implicación en lugar de *no* y *o* sugiere una interpretación más intuitiva de la consulta en español.

Considérese la consulta que se usó en el Apartado 2.3.3 para ilustrar la operación división: "Determinar todos los clientes que tienen una cuenta en todas las sucursales de Arganzuela". Para escribir esta consulta en el cálculo relacional de tuplas se introduce el constructor "para todo", denotado por $\forall$. La notación

$$\forall\, t \in r\ (Q(t))$$

significa "*Q* es verdadera para todas las tuplas *t* de la relación *r*".

La expresión para la consulta se escribe de la manera siguiente:

$$\{t \mid \exists\, r \in \mathit{cliente}\ (r[\mathit{nombre_cliente}] = t[\mathit{nombre_cliente}]) \wedge \\ (\forall\, u \in \mathit{sucursal}\ (u[\mathit{ciudad_sucursal}] = \text{“Arganzuela”} \Rightarrow \\ \exists\, s \in \mathit{impositor}\ (t[\mathit{nombre_cliente}] = s[\mathit{nombre_cliente}] \\ \wedge \exists\, w \in \mathit{cuenta}\ (w[\mathit{número_cuenta}] = s[\mathit{número_cuenta}] \\ \wedge w[\mathit{nombre_sucursal}] = u[\mathit{nombre_sucursal}])))\}$$

En español esta expresión se interpreta como "el conjunto de todos los clientes (es decir, las tuplas *t* (*nombre_cliente*)) tales que, para *todas* las tuplas *u* de la relación *sucursal*, si el valor de *u* en el atributo *ciudad_sucursal* es Arganzuela, el cliente tiene una cuenta en la sucursal cuyo nombre aparece en el atributo *nombre_sucursal* de *u*".

Obsérvese que hay cierta sutileza en la consulta anterior: si no hay ninguna sucursal en Arganzuela, todos los nombres de cliente satisfacen la condición. La primera línea de la expresión de consulta es crítica en este caso—sin la condición

$$\exists\, r \in \mathit{cliente}\ (r[\mathit{nombre_cliente}] = t[\mathit{nombre_cliente}])$$

si no hay ninguna sucursal en Arganzuela, cualquier valor de *t* (incluyendo los que no son nombres de cliente de la relación *cliente*) valdría.

5.1.2 Definición formal

Con la preparación necesaria adquirida hasta el momento es posible proporcionar una definición formal. Las expresiones del cálculo relacional de tuplas son de la forma:

$$\{t \mid P(t)\}$$

donde P es una *fórmula*. En una fórmula pueden aparecer varias variables tupla. Se dice que una variable tupla es una *variable libre* a menos que esté cuantificada mediante $\exists$ o $\forall$. Por tanto, en

$$t \in préstamo \wedge \exists\, s \in cliente(t[nombre_sucursal] = s[nombre_sucursal])$$

t es una variable libre. La variable tupla s se denomina variable *ligada*.

Las fórmulas del cálculo relacional de tuplas se construyen con *átomos*. Los átomos tienen una de las formas siguientes:

- $s \in r$, donde s es una variable tupla y r es una relación (no se permite el uso del operador $\notin$)
- $s[x]\ \Theta\ u[y]$, donde s y u son variables tuplas, x es un atributo en el que está definida s, y es un atributo en el que está definida u, y Θ es un operador de comparación ($<, \leq, =, \neq, >, \geq$); se exige que los atributos x e y tengan dominios cuyos miembros puedan compararse mediante Θ
- $s[x]\ \Theta\ c$, donde s es una variable tupla, x es un atributo en el que está definida s, Θ es un operador de comparación y c es una constante en el dominio del atributo x

Las fórmulas se construyen a partir de los átomos usando las reglas siguientes:

- Cada átomo es una fórmula.
- Si P_1 es una fórmula, también lo son $\neg P_1$ y (P_1).
- Si P_1 y P_2 son fórmulas, también lo son $P_1 \vee P_2$, $P_1 \wedge P_2$ y $P_1 \Rightarrow P_2$.
- Si $P_1(s)$ es una fórmula que contiene la variable tupla libre s, y r es una relación,

  $$\exists\, s \in r\,(P_1(s)) \text{ y } \forall\, s \in r\,(P_1(s))$$

  también son fórmulas.

Igual que en el álgebra relacional, se pueden escribir expresiones equivalentes que no sean idénticas en apariencia. En el cálculo relacional de tuplas estas equivalencias incluyen las tres reglas siguientes:

1. $P_1 \wedge P_2$ es equivalente a $\neg\,(\neg(P_1) \vee \neg(P_2))$.
2. $\forall\, t \in r\,(P_1(t))$ es equivalente a $\neg\, \exists\, t \in r\,(\neg P_1(t))$.
3. $P_1 \Rightarrow P_2$ es equivalente a $\neg(P_1) \vee P_2$.

5.1.3 Seguridad de las expresiones

Queda un último asunto por tratar. Las expresiones del cálculo relacional de tuplas pueden generar relaciones infinitas. Supóngase que se escribe la expresión

$$\{t \mid \neg\,(t \in préstamo)\}$$

Hay infinitas tuplas que no están en *préstamo*. La mayor parte de estas tuplas contienen valores que ni siquiera aparecen en la base de datos. Resulta evidente que no se desea permitir expresiones de ese tipo.

Para ayudar a definir las restricciones del cálculo relacional de tuplas se introduce el concepto de **dominio** de las fórmulas relacionales de tuplas, P. De manera intuitiva, el dominio de P, denotado por $dom(P)$, es el conjunto de todos los valores a los que P hace referencia. Esto incluye a los valores mencionados en la propia P, así como a los valores que aparezcan en tuplas de relaciones mencionadas por P. Por tanto, el dominio de P es el conjunto de todos los valores que aparecen explícitamente en P, o en una o más relaciones cuyos nombres aparezcan en P. Por ejemplo, $dom(t \in préstamo \wedge t[importe] > 1200)$ es el conjunto que contiene a 1200 y el conjunto de todos los valores que aparecen en *préstamo*.

Además, $dom(\neg\ (t\ \in\ préstamo))$ es el conjunto de todos los valores que aparecen en *préstamo*, dado que la relación $préstamo$ se menciona en la expresión.

Se dice que una expresión $\{t \mid P(t)\}$ es *segura* si todos los valores que aparecen en el resultado son valores de $dom(P)$. La expresión $\{t \mid \neg\ (t\ \in\ préstamo)\}$ no es segura. Obsérvese que $dom(\neg\ (t\ \in\ préstamo))$ es el conjunto de todos los valores que aparecen en *préstamo*. Sin embargo, es posible tener una tupla t que no esté en *préstamo* que contenga valores que no aparezcan en *préstamo*. El resto de ejemplos de expresiones del cálculo relacional de tuplas que se han escrito en este apartado son seguros.

5.1.4 Potencia expresiva de los lenguajes

El cálculo relacional de tuplas restringido a expresiones seguras es equivalente en potencia expresiva al álgebra relacional básica (con los operadores $\cup, -, \times, \sigma$ y ρ, pero sin los operadores relacionales extendidos como la proyección generalizada $\mathcal{G}$ y las operaciones de reunión externa). Por tanto, para cada expresión del álgebra relacional que sólo utilice los operadores básicos, existe una expresión equivalente del cálculo relacional de tuplas y viceversa. No se probará aquí esta afirmación; las notas bibliográficas contienen referencias a su demostración. Algunas partes de la misma se incluyen en los ejercicios. Hay que destacar que el cálculo relacional de tuplas no tiene ningún equivalente de la operación agregación, pero se puede extender para contenerla. La extensión del cálculo relacional de tuplas para que maneje las expresiones aritméticas es sencilla.

5.2 El cálculo relacional de dominios

Una segunda forma de cálculo relacional, denominada **cálculo relacional de dominios**, usa variables de *dominio*, que toman sus valores del dominio de un atributo, en vez de hacerlo para una tupla completa. El cálculo relacional de dominios, no obstante, se halla estrechamente relacionado con el cálculo relacional de tuplas.

5.2.1 Definición formal

Las expresiones del cálculo relacional de dominios son de la forma

$$\{< x_1,\ x_2, \ldots, x_n >\ \mid P(x_1,\ x_2, \ldots, x_n)\}$$

donde $x_1,\ x_2, \ldots, x_n$ representan las variables de dominio, P representa una fórmula compuesta por átomos, como era el caso en el cálculo relacional de tuplas. Los átomos del cálculo relacional de dominios tienen una de las formas siguientes:

- $< x_1,\ x_2, \ldots, x_n > \in\ r$, donde r es una relación con n atributos y $x_1,\ x_2, \ldots, x_n$ son variables de dominio o constantes de dominio.
- $x\ \Theta\ y$, donde x e y son variables de dominio y Θ es un operador de comparación ($<, \leq, =, \neq, >, \geq$). Se exige que los atributos x e y tengan dominios que puedan compararse mediante Θ.
- $x\ \Theta\ c$, donde x es una variable de dominio, Θ es un operador de comparación y c es una constante del dominio del atributo para el que x es una variable de dominio.

Las fórmulas se construyen a partir de los átomos usando las reglas siguientes:

- Los átomos son fórmulas.
- Si P_1 es una fórmula, también lo son $\neg P_1$ y (P_1).
- Si P_1 y P_2 son fórmulas, también lo son $P_1\ \vee\ P_2$, $P_1\ \wedge\ P_2$ y $P_1\ \Rightarrow\ P_2$.
- Si $P_1(x)$ es una fórmula en x, donde x es una variable libre de dominio, también son fórmulas:

$$\exists\ x\ (P_1(x)) \text{ y } \forall\ x\ (P_1(x))$$

Como notación abreviada se escribe $\exists\ a, b, c\ (P(a, b, c))$ en lugar de $\exists\ a\ (\exists\ b\ (\exists\ c\ (P(a, b, c))))$.

5.2.2 Ejemplos de consultas

Ahora se van a presentar consultas del cálculo relacional de dominios para los ejemplos considerados anteriormente. Obsérvese la similitud de estas expresiones con las expresiones correspondientes del cálculo relacional de tuplas.

- Determinar el número de préstamo, el nombre de la sucursal y el importe de los préstamos de más de 1.200 €:

$$\{< p, s, i > \mid < p, s, i > \in \ préstamo \ \wedge \ i \ > \ 1200\}$$

- Determinar todos los números de préstamo de los préstamos por importe superior a 1.200 €:

$$\{< p > \mid \exists \, s, i \, (< p, s, i > \in \ préstamo \ \wedge \ i \ > \ 1200)\}$$

Aunque la segunda consulta tenga un aspecto muy parecido al de la que se escribió para el cálculo relacional de tuplas, hay una diferencia importante. En el cálculo de tuplas, cuando se escribe $\exists \ s$ para alguna variable tupla s, se vincula inmediatamente con una relación escribiendo $\exists \ s \ \in \ r$. Sin embargo, cuando se usa $\exists \ s$ en el cálculo de dominios, b no se refiere a una tupla, sino a un valor de dominio. Por tanto, el dominio de la variable s no está restringido hasta que la subfórmula $< p, s, i > \in \ préstamo$ restringe s a los nombres de sucursal que aparecen en la relación *préstamo*.

Ahora se mostrarán varios ejemplos de consultas del cálculo relacional de dominios.

- Determinar el nombre de todos los clientes que tienen concedido un préstamo en la sucursal de Navacerrada y determinar el importe del préstamo:

$$\begin{aligned}\{< n, i > \mid \exists \ p \, (< n, p > \in \ prestatario \\ \wedge \exists \, s \, (< p, s, i > \in \ préstamo \ \wedge \ s \ = \ \text{“Navacerrada”}))\}\end{aligned}$$

- Determinar el nombre de todos los clientes que tienen concedido un préstamo, abierta una cuenta o ambas cosas en la sucursal de Navacerrada:

$$\begin{aligned}\{< n > \mid \exists p \, (< n, p > \in \ prestatario \\ \wedge \exists \, s, i \, (< p, s, i > \in \ préstamo \ \wedge \ s \ = \ \text{“Navacerrada”})) \\ \vee \exists \, c \, (< n, c > \in \ impositor \\ \wedge \exists \, s, x \, (< c, s, x > \in \ cuenta \ \wedge \ s \ = \ \text{“Navacerrada”}))\}\end{aligned}$$

- Determinar el nombre de todos los clientes que tienen una cuenta abierta en todas las sucursales situadas en Arganzuela:

$$\begin{aligned}\{< n > \mid \exists \, d, c (< n, d, c > \in \ cliente) \wedge \\ \forall \, x, y, z \, (< x, y, z > \in \ sucursal \ \wedge \ y \ = \ \text{“Arganzuela”} \ \Rightarrow \\ \exists \, a, b \, (< a, x, b > \in \ cuenta \wedge \ < n, a > \in \ impositor))\}\end{aligned}$$

  En español la expresión anterior se interpreta como “el conjunto de todas las tuplas c (*nombre_cliente*) tales que, para todas las tuplas x, y, z (*nombre_sucursal*, *ciudad_sucursal*, *activos*), si la ciudad de la sucursal es Arganzuela, las siguientes afirmaciones son verdaderas:

  - Existe una tupla de la relación *cuenta* con número de cuenta *a* y nombre de sucursal *x*.
  - Existe una tupla de la relación *impositor* con cliente *n* y número de cuenta *a*”.

5.2.3 Seguridad de las expresiones

Ya se observó que, en el cálculo relacional de tuplas (Apartado 5.1), es posible escribir expresiones que generen relaciones infinitas. Esto llevó a definir la *seguridad* de las expresiones de cálculo relacional de tuplas. Se produce una situación parecida en el cálculo relacional de dominios. Las expresiones como:

$$\{< p, s, i > \mid \neg (< p, s, i > \in \ préstamo)\}$$

no son seguras porque permiten valores del resultado que no están en el dominio de la expresión.

En el cálculo relacional de dominios también hay que tener en cuenta la forma de las fórmulas dentro de las instrucciones "existe" y "para todo". Considérese la expresión:

$$\{< x > \mid \exists\, y\, (< x,\, y >\in\ r)\ \wedge\ \exists\, z\, (\neg(< x,\, z >\in\ r)\ \wedge\ P(x, z))\}$$

donde P es una fórmula que implica a x y a z. Se puede comprobar la primera parte de la fórmula, $\exists\, y\ (< x,\, y >\in\ r)$, tomando en consideración sólo los valores de *r*. Sin embargo, para comprobar la segunda parte de la fórmula, $\exists\, z\ (\neg\ (< x,\, z >\in\ r)\ \wedge\ P(x, z))$, hay que tomar en consideración valores de *z* que no aparecen en *r*. Dado que todas las relaciones son finitas, no aparece en *r* un número infinito de valores. Por tanto, no resulta posible en general comprobar la segunda parte de la fórmula sin tomar en consideración un número infinito de valores posibles de z. En lugar de ello se añaden restricciones para prohibir expresiones como la anterior.

En el cálculo relacional de tuplas se restringió cualquier variable cuantificada existencialmente a variar sobre una relación concreta. Dado que no se hizo así en el cálculo de dominios, se añaden reglas a la definición de seguridad para tratar casos como el del ejemplo. Se dice que la expresión:

$$\{< x_1,\ x_2, \ldots, x_n > \mid P\,(x_1,\ x_2, \ldots, x_n)\}$$

es segura si se cumplen todas las condiciones siguientes:

1. Todos los valores que aparecen en las tuplas de la expresión son valores de *dom* (*P*).
2. Para cada subfórmula "existe" de la forma $\exists\, x\ (P_1(x))$, la subfórmula es cierta si y sólo si hay un valor *x* de $dom(P_1)$ tal que $P_1(x)$ es cierto.
3. Para cada subfórmula "para todo" de la forma $\forall x\ (P_1(x))$, la subfórmula es verdadera si y sólo si $P_1(x)$ es cierto para todos los valores *x* de $dom(P_1)$.

El propósito de las reglas adicionales es garantizar que se puedan comprobar las subfórmulas "para todo" y "existe" sin tener que comprobar infinitas posibilidades. Considérese la segunda regla de la definición de seguridad. Para que $\exists\, x\ (P_1(x))$ sea cierto sólo hay que encontrar una *x* para la que $P_1(x)$ lo sea. En general habría que comprobar infinitos valores. Sin embargo, si la expresión es segura, se sabe que se puede restringir la atención a los valores de $dom(P_1)$. Esta restricción reduce las tuplas que hay que tomar en consideración a un número finito.

La situación de las subfórmulas de la forma $\forall x\ (P_1(x))$ es parecida. Para asegurar que $\forall x\ (P_1(x))$ es cierto hay que comprobar en general todos los valores posibles, por lo que es necesario examinar infinitos valores. Como antes, si se sabe que la expresión es segura, basta con comprobar $P_1(x)$ para los valores tomados de $dom(P_1)$.

Todas las expresiones del cálculo relacional de dominios que se han incluido en los ejemplos de consultas de este apartado son seguras.

5.2.4 Potencia expresiva de los lenguajes

Cuando el cálculo relacional de dominios se restringe a las expresiones seguras, es equivalente en potencia expresiva al cálculo relacional de tuplas restringido también a las expresiones seguras. Dado que ya se observó anteriormente que el cálculo relacional de tuplas restringido es equivalente al álgebra relacional, los tres lenguajes siguientes son equivalentes:

- El álgebra relacional básica (sin las operaciones extendidas del álgebra relacional).
- El cálculo relacional de tuplas restringido a las expresiones seguras.
- El cálculo relacional de dominios restringido a las expresiones seguras.

Hay que tener en cuenta que el cálculo relacional de dominios tampoco tiene equivalente para la operación agregación, pero se puede extender fácilmente para contenerla.

5.3 Query-by-Example

Query-by-Example (QBE, Consulta mediante ejemplos) es el nombre tanto de un lenguaje de manipulación de datos como de un sistema de base de datos que incluyó ese lenguaje.

El lenguaje de manipulación de datos QBE tiene dos características distintivas:

1. A diferencia de la mayor parte de los lenguajes de consultas y de programación, QBE presenta una **sintaxis bidimensional**. Las consultas *parecen* tablas. Las consultas de los lenguajes unidimensionales (como SQL) se *pueden* formular en una línea (posiblemente larga). Los lenguajes bidimensionales *necesitan* dos dimensiones para su formulación. (Existe una versión unidimensional de QBE, pero no se considerará en este estudio).
2. Las consultas en QBE se expresan "mediante ejemplos". En lugar de incluir un procedimiento para obtener la respuesta deseada, se emplea un ejemplo de lo que se desea. El sistema generaliza ese ejemplo para calcular la respuesta a la consulta.

A pesar de estas características tan poco comunes existe una estrecha correspondencia entre QBE y el cálculo relacional de dominios.

Existen dos enfoques de QBE: la versión original basada en texto y una versión gráfica desarrollada posteriormente que soporta el sistema de bases de datos Microsoft Access. En este apartado se ofrece una breve introducción a las características de manipulación de datos de ambas versiones de QBE. En primer lugar se tratarán las características de QBE basado en texto que se corresponden con la cláusula select-from-where de SQL sin agregación ni actualizaciones. Consúltense las notas bibliográficas para ver las referencias de las que se puede conseguir más información sobre la manera en que la versión basada en texto de QBE maneja la ordenación de los resultados, la agregación y la actualización. Más adelante, en el Apartado 5.3.6 se tratarán brevemente las características de la versión gráfica de QBE.

5.3.1 Esqueletos de tablas

Una consulta en QBE se expresa mediante **esqueletos de tablas**. Estas tablas presentan el esquema de la relación, como se muestra en la Figura 5.2. En lugar de llenar la pantalla con todos los esqueletos, el usuario elige los que necesita para una determinada consulta y rellena esos esqueletos con **filas de ejemplo**. Cada fila de ejemplo consiste en constantes y *elementos ejemplo*, que son variables de dominio. Para evitar confusiones entre los dos, QBE usa un carácter de subrayado (_) antes de las variables de dominio, como en _x, y permite que las constantes aparezcan sin ninguna indicación particular. Este convenio contrasta con el de la mayoría de los lenguajes, en los que las constantes se encierran entre comillas y las variables aparecen sin ninguna indicación.

5.3.2 Consultas sobre una relación

Recuperando el ejemplo bancario habitual, para determinar todos los números de préstamo de la sucursal de Navacerrada se usa el esqueleto de la relación *préstamo* y se rellena del modo siguiente:

préstamo	*número_préstamo*	*nombre_sucursal*	*importe*
	P._ x	Navacerrada	

Esta consulta indica al sistema que busque tuplas de *préstamo* que tengan "Navacerrada" como valor del atributo *nombre_sucursal*. Para cada tupla de este tipo, el sistema asigna el valor del atributo *número_préstamo* a la variable x. El valor de la variable x se "imprime" (normalmente en pantalla), debido a que el comando P. (acrónimo de *print*—imprimir en inglés) aparece en la columna *número_préstamo* junto a la variable x. Obsérvese que este resultado es parecido al que se obtendría como respuesta a la siguiente consulta del cálculo relacional de dominios:

$$\{\langle x \rangle | \exists\, s, c(\langle x, s, c \rangle \in \textit{préstamo} \wedge s = \text{"Navacerrada"})\}$$

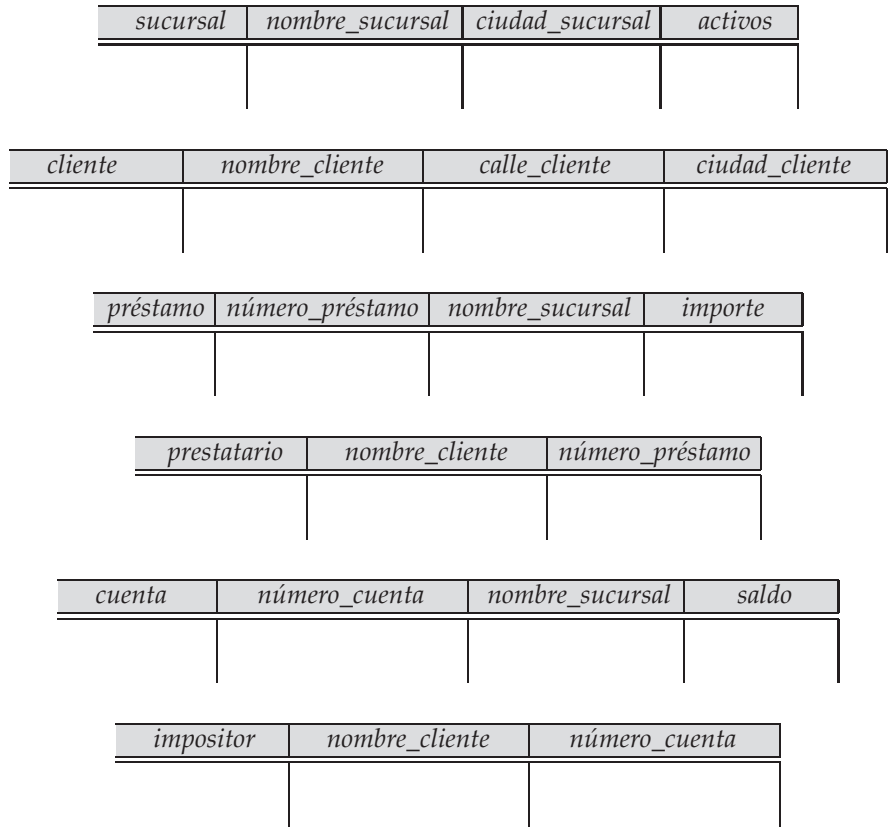


Figura 5.2 Esqueletos de tablas de QBE para el ejemplo bancario.

QBE asume que cada posición vacía de una fila contiene una variable única. En consecuencia, si alguna variable no aparece más de una vez en la consulta, se puede omitir. La consulta anterior, por tanto, puede volver a escribirse como:

préstamo	*número_préstamo*	*nombre_sucursal*	*importe*
	P.	Navacerrada	

QBE (a diferencia de SQL) realiza de manera automática la eliminación de duplicados. Para evitar la eliminación es necesario insertar el comando ALL. después del comando P.:

préstamo	*número_préstamo*	*nombre_sucursal*	*importe*
	P.ALL.	Navacerrada	

Para mostrar la relación *préstamo* completa, se puede crear una única fila con el comando P. en todos los campos. También se puede usar una notación más concisa consistente en colocar una única orden P. en la columna encabezada por el nombre de la relación:

préstamo	*número_préstamo*	*nombre_sucursal*	*importe*
P.			

QBE permite formular consultas que conlleven comparaciones aritméticas (por ejemplo, >), en lugar de las comparaciones de igualdad. Por ejemplo, "Determinar el número de todos los préstamos de aquellos préstamos de importe superior a 700 €":

préstamo	*número_préstamo*	*nombre_sucursal*	*importe*
	P.		>700

Las comparaciones sólo pueden contener una expresión aritmética en el lado derecho de la operación de comparación (por ejemplo, $> (_x + _y - 20)$). La expresión puede contener tanto variables como constantes. El lado izquierdo de la comparación debe estar vacío. Las operaciones aritméticas que son compatibles con QBE son $=$, $<$, $\leq$, $>$, $\geq$, y $\neg$.

Obsérvese que exigir que el lado izquierdo esté vacío implica que no se pueden comparar dos variables con nombres distintos. Esta dificultad se tratará en breve.

Como ejemplo adicional, considérese la consulta "Determinar el nombre de todas las sucursales que no se hallan en Arganzuela". Esta consulta se puede formular del siguiente modo:

sucursal	*nombre_sucursal*	*ciudad_sucursal*	*activos*
	P.	¬Arganzuela	

El objetivo principal de las variables en QBE es obligar a ciertas tuplas a tener el mismo valor en algunos atributos. Considérese la consulta "Determinar el número de préstamo de todos los préstamos solicitados conjuntamente por Santos y Gómez":

prestatario	*nombre_cliente*	*número_préstamo*
	Santos	P._x
	Gómez	_x

Para ejecutar esta consulta el sistema busca todos los pares de tuplas de la relación *prestatario*, que coinciden en el atributo *número_préstamo*, para los que el valor del atributo *nombre_cliente* es "Santos" para una tupla y "Gómez" para la otra. A continuación, el sistema muestra el valor del atributo *número_préstamo*.

En el cálculo relacional de dominios la consulta se podría escribir como:

$$\{\langle p \rangle \mid \exists\, x\, (\langle x, p \rangle \in prestatario \wedge x = \text{"Santos"})$$
$$\wedge \exists\, x\, (\langle x, p \rangle \in prestatario \wedge x = \text{"Gómez"})\}$$

Como ejemplo adicional, considérese la consulta "Determinar el nombre de todos los clientes que viven en la misma ciudad que Santos":

cliente	*nombre_cliente*	*calle_cliente*	*ciudad_cliente*
	P._x		_y
	Santos		_y

5.3.3 Consultas sobre varias relaciones

QBE permite formular consultas que abarquen varias relaciones diferentes (de igual forma que el producto cartesiano o la reunión natural en el álgebra relacional). Las conexiones entre las diversas relaciones se llevan a cabo a través de variables que obligan a ciertas tuplas a tomar el mismo valor en determinados atributos. Como ejemplo, supóngase que se desea determinar el nombre de todos los clientes que tienen concedido un préstamo en la sucursal de Navacerrada. Esta consulta se puede escribir como:

préstamo	*número_préstamo*	*nombre_sucursal*	*importe*
	_x	Navacerrada	

prestatario	*nombre_cliente*	*número_préstamo*
	P._y	_x

Para evaluar la consulta anterior, el sistema busca las tuplas de la relación *préstamo* con "Navacerrada" como valor del atributo *nombre_sucursal*. Para cada una de esas tuplas, el sistema busca las tuplas de la

relación *prestatario* con el mismo valor para el atributo *número_préstamo* que las tuplas de la relación *préstamo*. El sistema muestra el valor del atributo *nombre_cliente*.

Se puede usar una técnica parecida a la anterior para escribir la consulta "Determinar el nombre de todos los clientes que tienen tanto una cuenta abierta como un préstamo concedido en el banco":

impositor	*nombre_cliente*	*número_cuenta*
	P._x	

prestatario	*nombre_cliente*	*número_préstamo*
	_x	

Considérese ahora la consulta "Determinar el nombre de todos los clientes que tienen una cuenta en el banco pero que no tienen concedido ningún préstamo". En QBE las consultas que expresan negación se expresan con un signo **not** (¬) debajo del nombre de la relación y junto a una fila de ejemplo:

impositor	*nombre_cliente*	*número_cuenta*
	P._x	

prestatario	*nombre_cliente*	*número_préstamo*
¬	_x	

Compárese la consulta anterior con la formulada anteriormente: "Determinar el nombre de todos los clientes que tienen tanto una cuenta abierta como un préstamo concedido en el banco". La única diferencia es la aparición del símbolo ¬ junto a la fila de ejemplo del esqueleto de la tabla *prestatario*. Esta diferencia, no obstante, tiene un efecto importante en el procesamiento de la consulta. QBE busca todos los valores de *x* para los cuales:

1. Existe una tupla de la relación *impositor* cuyo *nombre_cliente* es la variable de dominio *x*.
2. No existe ninguna tupla de la relación *prestatario* cuyo *nombre_cliente* sea como el de la variable de dominio *x*.

El símbolo ¬ se puede leer como "no existe".

El hecho de que se haya colocado ¬ bajo el nombre de la relación, en lugar de hacerlo bajo el nombre de algún atributo es importante. El símbolo ¬ bajo el nombre de un atributo es una abreviatura de $\neq$. Por tanto, para encontrar todos los clientes que tienen al menos dos cuentas se escribe:

impositor	*nombre_cliente*	*número_cuenta*
	P._x	_y
	_x	¬ _y

La consulta anterior se lee en español "Mostrar todos los valores de *nombre_cliente* que aparecen al menos en dos tuplas, de las cuales la segunda tiene un *número_cuenta* diferente de la primera".

5.3.4 Cuadro de condiciones

Algunas veces resulta poco conveniente o imposible expresar todas las restricciones de las variables de dominio dentro de los esqueletos de tablas. Para solucionar este problema, QBE incluye la característica **cuadro de condiciones**, que permite expresar restricciones generales sobre cualquiera de las variables de dominio. QBE permite que aparezcan expresiones lógicas en los cuadros de condiciones. Los operadores lógicos son las palabras **and** (y) y **or** (o), o bien los símbolos "&" y "|".

Por ejemplo, la consulta "Determinar el número de préstamo de todos los préstamos concedidos a Santos o a Gómez (o a ambos conjuntamente)" se puede escribir como:

prestatario	*nombre_cliente*	*número_préstamo*
	_*n*	P._*x*

condiciones
_*n* = Santos **or** _*n* = Gómez

Es posible expresar esta consulta sin usar un cuadro de condiciones, usando P. en varias filas. Sin embargo, las consultas con P. en varias filas a veces resultan difíciles de entender y es mejor evitarlas.

Como ejemplo adicional, supóngase que se modifica la última consulta del Apartado 5.3.3 para que quede como: "Determinar todos los clientes cuyo nombre no sea 'Santos' y que tengan abiertas, al menos, dos cuentas". Se desea incluir en esta consulta la restricción "$x \neq$ Santos". Se consigue empleando el cuadro de condiciones e introduciéndole la restricción "$x \neq$ Santos":

condiciones
$x \neg =$ Santos

Cambiando de ejemplo, para determinar todos los números de cuenta con saldos entre 1.300 € y 1.500 €, se escribe:

cuenta	*número_cuenta*	*nombre_sucursal*	*saldo*
	P.		_*x*

condiciones
_*x* $\geq$ 1300
_*x* $\leq$ 1500

Considérese también la consulta "Determinar todas las sucursales con activos superiores a los activos de, al menos, una sucursal con sede en Arganzuela". Esta consulta se puede escribir como:

sucursal	*nombre_sucursal*	*ciudad_sucursal*	*activos*
	P._*x*		_*y*
		Arganzuela	_*z*

condiciones
_*y* > _*z*

QBE permite la aparición de expresiones aritméticas complejas dentro del cuadro de condiciones. Se puede formular la consulta "Determinar todas las sucursales que tienen activos que, como mínimo, dupliquen los activos de una de las sucursales con sede en Arganzuela" de manera parecida a como se ha formulado la consulta anterior y modificando el cuadro de condiciones:

condiciones
_*y* $\geq$ 2 * _*z*

Para obtener el número de cuenta de todas las cuentas que tienen un saldo entre 1.300 € y 2.000 €, pero que no sea exactamente igual a 1.500 € se escribirá:

cuenta	*número_cuenta*	*nombre_sucursal*	*saldo*
	P.		_x

condiciones
_x = (≥ 1300 **and** ≤ 2000 **and** ¬ 1500)

QBE usa el constructor **or** de manera poco habitual para permitir la realización de comparaciones con conjuntos de valores constantes. Para determinar todas las sucursales que se hallan en Arganzuela o en Usera, se escribirá:

sucursal	*nombre_sucursal*	*ciudad_sucursal*	*activo*
	P.	_ x	

condiciones
_x = (Arganzuela **or** Usera)

5.3.5 La relación resultado

Las consultas que se han formulado hasta ahora tienen una característica en común: los resultados aparecen en un único esquema de relación. Si el resultado de una consulta contiene atributos de varios esquemas de relación, se necesita un mecanismo para mostrar el resultado deseado en una sola tabla. Para este propósito se puede declarar una relación temporal *resultado* que incluya todos los atributos del resultado de la consulta. Se imprime el resultado deseado incluyendo el comando P. únicamente en el esqueleto de la tabla *resultado*.

Como ejemplo, considérese la consulta "Determinar *nombre_cliente*, *número_cuenta* y *saldo* de todas las cuentas de la sucursal de Navacerrada". En el álgebra relacional esta consulta se formularía de la forma siguiente:

1. Reunión de las relaciones *impositor* y *cuenta*.
2. Proyección sobre los atributos *nombre_cliente*, *número_cuenta* y *saldo*.

Para formular la misma consulta en QBE se procede del siguiente modo:

1. Se crea un esqueleto de tablas, denominado *resultado*, con los atributos *nombre_cliente*, *número _cuenta* y *saldo*. El nombre del esqueleto de tabla recién creado (es decir, *resultado*), debe ser diferente de todos los nombres de relación de la base de datos ya existentes.
2. Se escribe la consulta.

La consulta resultante es:

cuenta	*número_cuenta*	*nombre_sucursal*	*saldo*
	_y	Navacerrada	_z

impositor	*nombre_cliente*	*número_cuenta*
	_x	_y

resultado	*nombre_cliente*	*número_cuenta*	*saldo*
P.	_x	_y	_z

5.3.6 QBE en Microsoft Access

En este apartado se revisa la versión de QBE soportada por Microsoft Access. Aunque QBE originalmente se diseñó para un entorno de visualización basado en texto, QBE de Access está diseñado para un

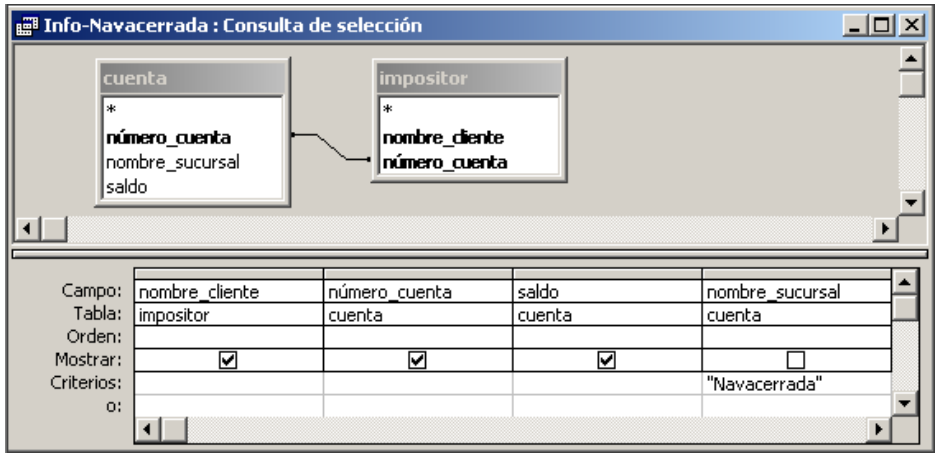


Figura 5.3 Una consulta de ejemplo en QBE de Microsoft Access.

entorno gráfico de visualización y, por tanto, se denomina **consulta gráfica mediante ejemplos (GQBE, Graphical Query-By-Example)**.

La Figura 5.3 muestra una consulta de ejemplo en GQBE. La consulta se puede describir en español como "Determinar *nombre_cliente*, *número_cuenta* y *saldo* de todas las cuentas de la sucursal de Navacerrada". En el Apartado 5.3.5 se mostró cómo se expresa en QBE.

Una pequeña diferencia en la versión de GQBE es que los atributos de las tablas se escriben uno debajo de otro, en lugar de horizontalmente. Una diferencia más significativa es que la versión gráfica de QBE emplea una línea que une los atributos de dos tablas, en lugar de una variable compartida, para especificar las condiciones de reunión.

Una característica interesante de QBE de Access es que los vínculos entre las tablas se crean automáticamente según el nombre de los atributos. En el ejemplo de la Figura 5.3 se han integrado en la consulta las tablas *cuenta* e *impositor*. El atributo *número_cuenta* se comparte entre las dos tablas seleccionadas y el sistema inserta automáticamente un vínculo entre esas dos tablas. En otras palabras, de manera predeterminada se impone una condición de reunión natural entre las tablas; el vínculo se puede borrar si no se desea que exista. El vínculo también se puede especificar para que denote una reunión externa natural, en lugar de una reunión natural.

Otra pequeña diferencia en QBE de Access es que especifica en un cuadro separado, denominado **cuadrícula de diseño**, los atributos que se van a mostrar en lugar de usar P. en la tabla. En esta cuadrícula de diseño también se especifican las selecciones según el valor de los atributos.

Las consultas que implican agrupaciones y agregaciones se pueden crear en Access como se muestra en la Figura 5.4. La consulta de la figura busca el nombre, la dirección y la ciudad de todos los clientes que tienen más de una cuenta en el banco. Los atributos y las funciones "de agregación" se marcan en la cuadrícula de diseño.

Téngase en cuenta que cuando aparece alguna condición en una columna de la cuadrícula de diseño con la fila "Total" definida para una función de agregación, la condición se aplica sobre el valor agregado; por ejemplo, en la Figura 5.4, la selección "> 1" de la columna *número_cuenta* se aplica al resultado de la función de agregación "Contar". Estas selecciones equivalen a las de las cláusulas **having** de SQL.

Las condiciones de selección se pueden aplicar a columnas de la cuadrícula de diseño que no estén ni agrupadas ni agregadas; estos atributos deben marcarse como "Dónde" en la fila "Total". Este tipo de selecciones "Dónde" se aplican antes de la agregación, y equivalen a las selecciones en las cláusulas **where** de SQL. No obstante, este tipo de columnas no puede mostrarse (no es posible marcarlas como "Mostrar"). Sólo se pueden mostrar las columnas en que la fila "Total" especifica una "agrupación" o una función de agregación.

Las consultas se crean mediante una interfaz gráfica de usuario, seleccionando en primer lugar las tablas. A continuación se pueden añadir los atributos a la cuadrícula de diseño arrastrándolos y soltándolos desde las tablas. Las condiciones de selección, agrupación y agregación se pueden especificar a continuación sobre los atributos de la cuadrícula de diseño. QBE de Access ofrece otra serie de caracte-

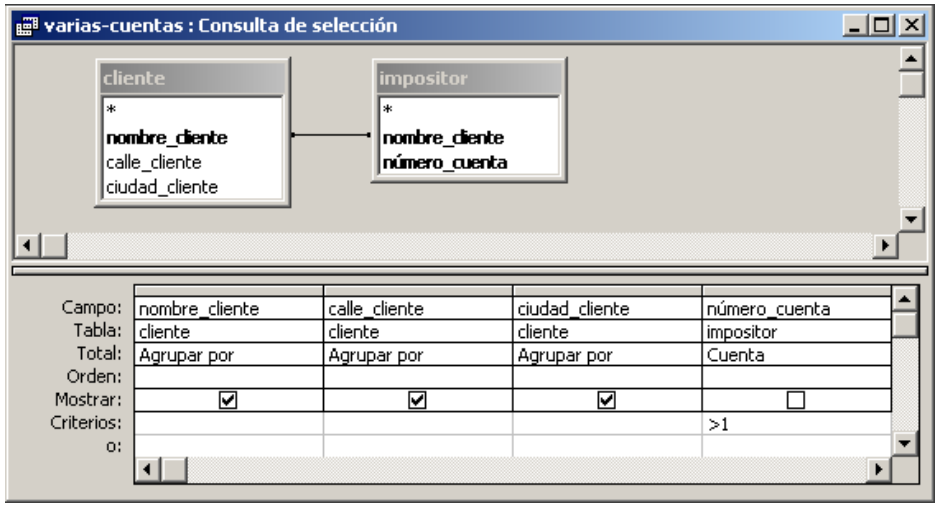


Figura 5.4 Una consulta de agregación en QBE de Microsoft Access.

rísticas, incluidas las consultas para la modificación de la base de datos mediante inserción, borrado y actualización.

5.4 Datalog

Datalog es un lenguaje de consultas no procedimental basado en el lenguaje de programación lógica Prolog. Al igual que en el cálculo relacional, el usuario describe la información deseada sin especificar el procedimiento concreto para obtener esa información. La sintaxis de Datalog recuerda la de Prolog. Sin embargo, el significado de los programas en Datalog se define de una manera puramente declarativa, a diferencia de la semántica más procedimental de Prolog, por lo que Datalog simplifica la escritura de consultas sencillas y facilita la optimización de las consultas.

5.4.1 Estructura básica

Cada programa Datalog consiste en un conjunto de **reglas**. Antes de dar una definición formal de las reglas Datalog y de su significado formal, se presentarán algunos ejemplos. Supóngase una regla Datalog para definir una vista $v1$ que contiene el número de cuenta y el saldo de las cuentas de la sucursal de Navacerrada cuyo saldo es superior a 700 €:

$$v1(C, S) \text{ :– } cuenta(C, \text{“Navacerrada”}, S), S > 700$$

Las reglas Datalog definen vistas; la regla anterior **usa** la relación *cuenta* y **define** la vista $v1$. El símbolo :– se lee “si”, y la coma que separa “$cuenta(C, \text{“Navacerrada”}, S)$” de “$S > 700$” se lee “y”. Intuitivamente, la regla se entiende del siguiente modo:

for all C, S
if $(C, \text{“Navacerrada”}, S) \in cuenta$ **and** $S > 700$
then $(C, S) \in v1$

Supóngase que la relación *cuenta* es la mostrada en la Figura 5.5. Entonces, la vista $v1$ contiene las tuplas mostradas en la Figura 5.6. Para obtener el saldo de la cuenta C-217 de esta vista se escribe:

$$? \; v1(\text{“C-217”}, S)$$

La respuesta a la consulta es:

(C-217, 750)

número_cuenta	*nombre_sucursal*	*saldo*
C-215	Becerril	700
C-101	Centro	500
C-305	Collado Mediano	350
C-201	Galapagar	900
C-217	Galapagar	750
C-222	Moralzarzal	700
C-102	Navacerrada	400

Figura 5.5 La relación *cuenta*.

Para determinar el número de cuenta y el saldo de todas las cuentas de la relación *v1* cuyo saldo es superior a 800 se puede escribir:

$$? \; v1(C, S), S > 800$$

La respuesta a la consulta es:

(C-201, 900)

En general hace falta más de una regla para definir una vista. Cada regla define un conjunto de tuplas que debe contener la vista. El conjunto de tuplas de la vista se define, por tanto, como la unión de todos esos conjuntos de tuplas. El siguiente programa Datalog especifica los tipos de interés para las cuentas:

$$\begin{aligned} &\textit{tipo_interés}(C, 5) \text{ :– } \textit{cuenta}(C, N, S), S < 10000 \\ &\textit{tipo_interés}(C, 6) \text{ :– } \textit{cuenta}(C, N, S), S >= 10000 \end{aligned}$$

El programa tiene dos reglas que definen la vista *tipo_interés*, cuyos atributos son el número de cuenta y el tipo de interés. Las reglas especifican que, si el saldo es menor de 10.000 €, el tipo de interés es el cinco por ciento y, si el saldo es igual o superior a 10.000 €, entonces el tipo de interés es el seis por ciento.

Las reglas Datalog pueden usar la negación. Las reglas siguientes definen una vista *c* que contiene el nombre de todos los clientes que tienen abierta una cuenta pero no tienen concedido ningún préstamo en el banco:

$$\begin{aligned} &c(N) \text{ :– } \textit{impositor}(N,C), \textbf{not } \textit{es_prestatario}(N) \\ &\textit{es_prestatario}(N) \text{ :– } \textit{prestatario}(N, P) \end{aligned}$$

Prolog y la mayoría de las implementaciones de Datalog reconocen los atributos de una relación por su posición y omiten el nombre de los atributos. Por tanto, las reglas Datalog son compactas en comparación con las consultas de SQL. Sin embargo, cuando las relaciones tienen gran número de atributos o el orden o el número de los atributos de la relación pueden variar, la notación posicional puede resultar engorrosa y conducir a errores. No es difícil crear una variante de la sintaxis de Datalog que reconozca los atributos por su nombre en lugar de por su posición. En un sistema de ese tipo, la regla Datalog que define *v1*, se puede escribir como:

$$\begin{aligned} &\textit{v1}(\textit{número_cuenta } C, \textit{saldo } S) \text{ :–} \\ &\quad \textit{cuenta}(\textit{número_cuenta } C, \textit{nombre_sucursal } \text{“Navacerrada”}, \textit{saldo } S), \\ &\quad S > 700 \end{aligned}$$

número_cuenta	*saldo*
C-201	900
C-217	750

Figura 5.6 La relación *v1*.

$$
\begin{aligned}
&\textit{interés}(C, I) \text{ :– } \textit{cuenta}(C, \text{“Navacerrada”}, S),\\
&\qquad\qquad \textit{tipo_interés}(C, T), I = S * T/100\\
&\textit{tipo_interés}(C, 5) \text{ :– } \textit{cuenta}(C, N, S), S < 10000\\
&\textit{tipo_interés}(C, 6) \text{ :– } \textit{cuenta}(C, N, S), S >= 10000
\end{aligned}
$$

Figura 5.7 Programa Datalog que define el interés de las cuentas de la sucursal de Navacerrada.

La traducción entre las dos variedades puede hacerse sin un esfuerzo significativo, siempre que se disponga del esquema de relación.

5.4.2 Sintaxis de las reglas Datalog

Una vez que se han explicado informalmente las reglas y las consultas, se puede definir formalmente su sintaxis; su significado se estudia en el Apartado 5.4.3. Se usan los mismos convenios que en el álgebra relacional para denotar los nombres de las relaciones, de los atributos y de las constantes (números o cadenas de caracteres entrecomilladas). Se emplean letras mayúsculas y palabras con la primera letra en mayúsculas para denotar nombres de variables, y letras minúsculas y palabras con la primera letra en minúsculas para denotar los nombres de las relaciones y de los atributos. Algunos ejemplos de constantes son 4, que es un número, y “Martín”, que es una cadena de caracteres; X y *Nombre* son variables. Un **literal positivo** tiene la forma:

$$p(t_1, t_2, \ldots, t_n)$$

donde p es el nombre de una relación con n atributos y t_1, t_2, …, t_n son constantes o variables. Un **literal negativo** tiene la siguiente forma:

$$\textbf{not } p(t_1, t_2, \ldots, t_n)$$

donde la relación p tiene n atributos. El siguiente es un ejemplo de literal:

$$\textit{cuenta}(C, \text{“Navacerrada”}, S)$$

Los literales que contienen operaciones aritméticas se tratan de un modo especial. Por ejemplo, el literal $S > 700$, aunque no tiene la sintaxis descrita, puede entenderse conceptualmente como $>(S, 700)$, que *sí* tiene la sintaxis requerida y donde $>$ es una relación.

Pero ¿qué significa esta notación para las operaciones aritméticas como “>”? La relación $>$ (conceptualmente) contiene tuplas de la forma (x, y) para todos los pares de valores x, y posibles, tales que $x > y$. Por tanto, $(2, 1)$ y $(5, -33)$ son tuplas de la relación $>$. Evidentemente, la relación $>$ es infinita. Otras operaciones aritméticas (como $>$, $=$, $+$ o $-$) se tratan también conceptualmente como relaciones. Por ejemplo, A = B + C se puede tratar conceptualmente como +(B, C, A), donde la relación $+$ contiene todas las tuplas (x, y, z) tales que $z = x + y$.

Un **hecho** se escribe de la forma:

$$p(v_1, v_2, \ldots, v_n)$$

y denota que la tupla $(v_1, v_2, \ldots, v_n)$ pertenece a la relación p. Un conjunto de hechos de una relación se puede escribir también con la notación tabular habitual. Un conjunto de hechos para las relaciones de un esquema de base de datos equivale a un ejemplar del esquema de la base de datos. Las **reglas** se construyen a partir de literales, y tienen la forma:

$$p(t_1, t_2, \ldots, t_n) \text{ :– } L_1, L_2, \ldots, L_n$$

donde cada L_i es un literal (positivo o negativo). El literal $p(t_1, t_2, \ldots, t_n)$ se denomina **cabeza** de la regla y el resto de los literales constituyen el **cuerpo** de la misma.

Un **programa Datalog** consiste en un conjunto de reglas y el orden en el que se formulen es indiferente. Como ya se ha mencionado, puede haber varias reglas que definan cada relación.

En la Figura 5.7 se muestra un ejemplo de programa Datalog que define el tipo de interés para cada cuenta de la sucursal de Navacerrada. La primera regla del programa define la vista *interés*, cuyos atributos son el número de cuenta y su tipo de interés. Usa la relación *cuenta* y la vista *tipo_interés*. Las dos últimas reglas del programa son reglas que ya se han visto.

Se dice que la vista v_1 **depende directamente de** la vista v_2 si v_2 se usa en la expresión que define a v_1. En el programa anterior, la vista *interés* depende directamente de las relaciones *tipo_interés* y *cuenta*. A su vez, la relación *tipo_interés* depende directamente de *cuenta*.

Se dice que la vista v_1 **depende indirectamente de** la vista v_2 si hay una secuencia de relaciones intermedias $i_1, i_2, \ldots, i_n$ para algún n tal que v_1 depende directamente de i_1, i_1 depende directamente de i_2, y así sucesivamente hasta i_{n-1}, que depende directamente de i_n.

En el ejemplo de la Figura 5.7, dado que hay una cadena de dependencias desde *interés*, pasando por *tipo_interés*, hasta *cuenta*, la relación *interés* también depende indirectamente de *cuenta*.

Finalmente, se dice que la vista v_1 **depende de** la vista v_2 si v_1 depende directa o indirectamente de v_2.

Se dice que la vista v es **recursiva** si depende de sí misma. Se dice que la vista que no depende de sí misma **no es recursiva**.

Considérese el programa de la Figura 5.8. En él, la vista *empl* depende de sí misma (debido a la segunda regla) y, por tanto, es recursiva. En cambio, el programa de la Figura 5.7 no es recursivo.

5.4.3 Semántica de Datalog no recursivo

A continuación se considera la semántica formal de los programas Datalog. Por ahora se analizarán únicamente los programas no recursivos. La semántica de los programas recursivos es algo más complicada y se analizará en el Apartado 5.4.6. La semántica de los programas se define empezando por la semántica de una sola regla.

5.4.3.1 Semántica de las reglas

El **ejemplar básico de una regla** es el resultado de sustituir cada variable de la regla por alguna constante. Si una variable aparece varias veces en una regla, todas las apariciones de esa variable se deben sustituir por la misma constante. Los ejemplares básicos se suelen llamar simplemente **ejemplares**.

La regla de ejemplo que define *v1* y un ejemplar de esa regla quedan como:

v1(A, B) :– *cuenta*(C, “Navacerrada”, S), $S > 700$
v1(“C-217”, 750) :– *cuenta*(“C-217”, “Navacerrada”, 750), $750 > 700$

En este ejemplo la variable C ha sido sustituida por “C-217” y la variable S por 750.

Normalmente cada regla tiene muchos ejemplares posibles. Estos ejemplares se corresponden con las diversas formas de asignar valores a cada variable de la regla.

Supóngase que se tiene la regla R:

$$p(t_1, t_2, \ldots, t_n) \text{ :– } L_1, L_2, \ldots, L_n$$

y un conjunto de hechos I asociados a las relaciones que aparecen en la regla (I también se puede considerar un ejemplar de la base de datos). Considérese cualquier ejemplar R' de la regla R:

$$p(v_1, v_2, \ldots, v_n) \text{ :– } l_1, l_2, \ldots, l_n$$

donde cada literal l_i es de la forma $q_i(v_{i,1}, v_{i,2}, \ldots, v_{i,n_i})$ o **not** $q_i(v_{i,1}, v_{i,2}, \ldots, v_{i,n_i})$, donde cada v_i y cada $v_{i,j}$ son constantes.

empl(X, Y) :– *jefe*(X, Y)
empl(X, Y) :– *jefe*(X, Z), *empl*(Z, Y)

Figura 5.8 Programa Datalog recursivo.

Se dice que el cuerpo del ejemplar R' de la regla se **satisface** en I si:

1. Para cada literal positivo $q_i(v_{i,1}, \ldots, v_{i,n_i})$ del cuerpo de R', el conjunto de hechos I contiene el hecho $q(v_{i,1}, \ldots, v_{i,n_i})$.
2. Para cada literal negativo **not** $q_j(v_{j,1}, \ldots, v_{j,n_j})$ del cuerpo de R', el conjunto de hechos I no contiene el hecho $q_j(v_{j,1}, \ldots, v_{j,n_j})$.

Se define el conjunto de hechos que se pueden **inferir** a partir de un conjunto de hechos I dado empleando la regla R como:

$$\begin{aligned} \textit{inferir}(R, I) = \{p(t_1, \ldots, t_{n_i}) \mid &\text{existe un ejemplar } R' \text{ de } R, \\ &\text{donde } p(t_1, \ldots, t_{n_i}) \text{ es la cabeza de } R', \text{ y} \\ &\text{el cuerpo de } R' \text{ se satisface en } I\}. \end{aligned}$$

Dado un conjunto de reglas $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ se define:

$$\textit{inferir}(\mathcal{R}, I) = \textit{inferir}(R_1, I) \cup \textit{inferir}(R_2, I) \cup \ldots \cup \textit{inferir}(R_n, I)$$

Supóngase un conjunto de hechos I que contiene las tuplas de la relación *cuenta* de la Figura 5.5. Un posible ejemplar de la regla de ejemplo R es

$$\textit{v1}(\text{“C-217”}, 750) \text{ :– } \textit{cuenta}(\text{“C-217”}, \text{“Navacerrada”}, 750), 750 > 700$$

El hecho *cuenta* (“C-217”, “Navacerrada”, 750) pertenece al conjunto de hechos I. Además, 750 es mayor que 700 y, por tanto, conceptualmente, $(750, 700)$ pertenece a la relación “$>$”. Por tanto, el cuerpo del ejemplar de la regla se satisface en I. Existen otros ejemplares posibles de R y, usándolos, se comprueba que *inferir*(R, I) tiene exactamente el conjunto de hechos para *v1* que se muestra en la Figura 5.9.

5.4.3.2 Semántica de los programas

Cuando una vista se define en términos de otra, el conjunto de hechos de la primera vista depende de los hechos de la segunda. En este apartado se da por supuesto que las definiciones no son recursivas: es decir, ninguna vista depende (directa o indirectamente) de sí misma. Por tanto, se pueden superponer las vistas en capas de la forma siguiente, y se puede emplear la superposición para definir la semántica del programa:

- Una relación está en la capa 1 si todas las relaciones que aparecen en los cuerpos de las reglas que la definen están almacenadas en la base de datos.
- Una relación está en la capa 2 si todas las relaciones que aparecen en los cuerpos de las reglas que la definen están almacenadas en la base de datos, o son de la capa 1.
- En general, una relación p está en la capa $i + 1$ si (1) no está en las capas $1, 2, \ldots, i$ y (2) todas las relaciones que aparecen en los cuerpos de las reglas que la definen están almacenadas en la base de datos o son de las capas $1, 2, \ldots, i$.

Considérese el programa de la Figura 5.7 con la regla adicional:

$$\textit{cuenta_navacerrada}(X, Y) \text{ :– } \textit{cuenta}(X, \text{“Navacerrada”}, Y)$$

número_cuenta	*saldo*
C-201	900
C-217	750

Figura 5.9 Resultado de *inferir*(R, I).

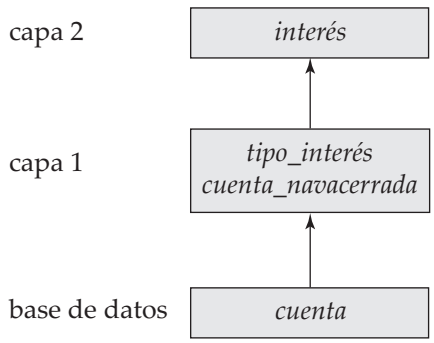


Figura 5.10 Clasificación en capas de las vistas.

La clasificación en capas de las vistas del programa se muestra en la Figura 5.10. La relación *cuenta* está en la base de datos. La relación *tipo_interés* está en la capa 1, mientras que todas las relaciones que se usan en las dos reglas que la definen están en la base de datos. La relación *cuenta_navacerrada* está también en la capa 1. Por último, la relación *interés* está en la capa 2, puesto que no está en la capa 1 y todas las relaciones que se usan en las reglas que la definen están en la base de datos o en capas inferiores a la 2.

Ahora se puede definir la semántica de los programas Datalog en términos de la clasificación en capas de las vistas. Sean las capas de un programa dado $1, 2, \ldots, n$ y $\mathcal{R}_i$ el conjunto de todas las reglas que definen vistas en la capa i.

- Se define I_0 como el conjunto de hechos almacenados en la base de datos e I_1 como

$$I_1 = I_0 \cup \mathit{inferir}(\mathcal{R}_1, I_0)$$

- Se procede de un modo análogo, definiendo I_2 en términos de I_1 y $\mathcal{R}_2$, y así sucesivamente usando la siguiente definición:

$$I_{i+1} = I_i \cup \mathit{inferir}(\mathcal{R}_{i+1}, I_i)$$

- Por último, el conjunto de hechos de las vistas definidos por el programa (también denominado **semántica del programa**) se define como el conjunto de hechos I_n de la capa más alta n.

Para el programa de la Figura 5.7, I_0 es el conjunto de hechos en la base de datos e I_1 es el conjunto de hechos de la base de datos ampliado con el conjunto de todos los hechos que se pueden inferir de I_0 usando las reglas de las relaciones *tipo_interés* y *cuenta_navacerrada*. Finalmente, I_2 contiene los hechos de I_1 junto con los hechos de la relación *interés* que se pueden inferir de los hechos en I_1 mediante la regla que define *interés*. La semántica del programa—es decir, el conjunto de hechos que están en cada vista—se define como el conjunto de hechos de I_2.

Recuérdese que en el Apartado 3.9.2 se vio la manera de definir el significado de las vistas no recursivas del álgebra relacional empleando una técnica denominada **expansión de vistas**. La expansión de vistas se puede usar también con las vistas no recursivas de Datalog; del mismo modo, la técnica de clasificación por capas aquí descrita se puede usar con las vistas del álgebra relacional.

5.4.4 Seguridad

Es posible formular reglas que generen un número infinito de respuestas. Considérese la regla

$$\mathit{mayor}(X, Y) \text{ :– } X > Y$$

Como la relación que define $>$ es infinita, esta regla generaría un número infinito de hechos para la relación *mayor*, cuyo cálculo necesitaría, lógicamente, una cantidad infinita de tiempo y de espacio.

El empleo de la negación puede causar problemas parecidos. Considérese la regla:

$$\mathit{no_en_préstamo}(P, S, I) \text{ :– } \textbf{not } \mathit{préstamo}(P, S, I)$$

La idea es que la tupla *(número_préstamo, nombre_sucursal, importe)* está en la vista *no_en_préstamo* si no pertenece a la relación *préstamo*. Sin embargo, si el conjunto de posibles números de préstamos, nombres de sucursales e importes es infinito, la relación *no_en_préstamo* también será infinita.

Por último, si existe una variable en la cabeza de la regla que no aparece en el cuerpo, se puede generar un conjunto infinito de hechos en los que la variable se asigna a distintos valores.

Con objeto de evitar estas posibilidades, se exige que las reglas Datalog cumplan las siguientes condiciones de **seguridad**:

1. Todas las variables que aparecen en la cabeza de una regla deben aparecer en un literal positivo no aritmético en el cuerpo de esa regla.
2. Todas las variables que aparecen en un literal negativo en el cuerpo de una regla deben aparecer también en algún literal positivo en el cuerpo de la misma.

Si todas las reglas de un programa Datalog no recursivo satisfacen las condiciones de seguridad anteriores, se puede probar que todas las vistas definidas en el programa son finitas, siempre que todas las relaciones de la base de datos sean finitas. Estas condiciones se pueden relajar algo para permitir que las variables de la cabeza aparezcan sólo en literales aritméticos del cuerpo en determinados casos. Por ejemplo, en la regla:

$$p(A) \text{ :- } q(B), A = B + 1$$

se puede observar que si la relación q es finita, también lo es p, por las propiedades de la suma, aunque la variable A sólo aparezca en un literal aritmético.

5.4.5 Operaciones relacionales en Datalog

Las expresiones de Datalog no recursivas sin operaciones aritméticas son equivalentes en poder expresivo a las expresiones que usan las operaciones básicas del álgebra relacional ($\cup$, $-$, $\times$, σ, Π y ρ). Esta afirmación no se probará formalmente ahora. En su lugar, se usarán ejemplos para mostrar la forma de expresar en Datalog las diversas operaciones del álgebra relacional. En todos los casos se define una vista denominada *consulta* para ilustrar las operaciones.

Ya se ha visto anteriormente la manera de llevar a cabo selecciones mediante reglas Datalog. Las proyecciones se realizan usando únicamente los atributos requeridos en la cabeza de la regla. Para proyectar el atributo *nombre_cuenta* de la relación *cuenta*, se usa:

$$\textit{consulta}(C) \text{ :- } \textit{cuenta}(C, N, S)$$

Se puede obtener el producto cartesiano de dos relaciones r_1 y r_2 en Datalog de la manera siguiente:

$$\textit{consulta}(X_1, X_2, \ldots, X_n, Y_1, Y_2, \ldots, Y_m) \text{ :- } r_1(X_1, X_2, \ldots, X_n), r_2(Y_1, Y_2, \ldots, Y_m)$$

donde r_1 es de aridad n, r_2 es de aridad m y $X_1, X_2, \ldots, X_n, Y_1, Y_2, \ldots, Y_m$ son nombres de variables, todos distintos entre sí.

La unión de dos relaciones r_1 y r_2 (ambas de aridad n), se forma del siguiente modo:

$$\begin{aligned} &\textit{consulta}(X_1, X_2, \ldots, X_n) \text{ :- } r_1(X_1, X_2, \ldots, X_n) \\ &\textit{consulta}(X_1, X_2, \ldots, X_n) \text{ :- } r_2(X_1, X_2, \ldots, X_n) \end{aligned}$$

El conjunto diferencia de dos relaciones r_1 y r_2, se calcula como sigue:

$$\textit{consulta}(X_1, X_2, \ldots, X_n) \text{ :- } r_1(X_1, X_2, \ldots, X_n), \textbf{not } r_2(X_1, X_2, \ldots, X_n)$$

Finalmente, obsérvese que, con la notación posicional usada en Datalog, el operador renombramiento ρ no es necesario. Cada relación puede aparecer más de una vez en el cuerpo de la regla pero, en lugar de renombrarla para dar nombres diferentes a sus apariciones, se emplean diferentes nombres de variables en las diferentes apariciones.

Es posible demostrar que cualquier consulta no recursiva de Datalog sin funciones aritméticas se puede expresar mediante las operaciones del álgebra relacional. Esta demostración se plantea como ejercicio para el lector. Por tanto, se puede establecer la equivalencia de las operaciones básicas del álgebra relacional y de Datalog no recursivo sin operaciones aritméticas.

Algunas extensiones de Datalog soportan las operaciones relacionales de actualización (inserción, borrado y actualización). La sintaxis para estas operaciones varía de una implementación a otra. Algunos sistemas permiten el uso de $+$ o $-$ en la cabeza de las reglas para denotar la inserción y el borrado relacionales. Por ejemplo, se pueden trasladar todas las cuentas de la sucursal de Navacerrada a la sucursal de Soto del Real ejecutando:

$$+ \textit{cuenta}(C, \text{“Soto del Real”}, S) \text{ :– } \textit{cuenta}(C, \text{“Navacerrada”}, S)$$
$$- \textit{cuenta}(C, \text{“Navacerrada”}, S) \text{ :– } \textit{cuenta}(C, \text{“Navacerrada”}, S)$$

Algunas implementaciones de Datalog también soportan la operación de agregación del álgebra relacional extendido. Una vez más, no existe sintaxis normalizada para esta operación.

5.4.6 Recursividad en Datalog

Varias aplicaciones de bases de datos manejan estructuras parecidas a los árboles de datos. Por ejemplo, considérense los empleados de una empresa. Algunos empleados son jefes. Cada jefe dirige un conjunto de personas que están bajo su mando. Pero cada una de ellas puede ser asimismo jefe y tener otras personas a su mando. Por tanto, los empleados se pueden organizar en una estructura parecida a un árbol.

Supóngase que se tiene el esquema de relación:

$$Esquema_jefe = (nombre_empleado, nombre_jefe)$$

y sea *jefe* una relación del esquema anterior.

Supóngase ahora que se desea determinar los empleados que supervisa (directa o indirectamente) un determinado jefe—por ejemplo, Santos. Es decir, si el jefe de Alández es Bariego, el jefe de Bariego es Erice y el jefe de Erice es Santos, entonces Alández, Bariego y Erice son los empleados supervisados por Santos. A menudo se escriben programas recursivos para trabajar con los árboles de datos. Usando la idea de la recursividad se puede definir el conjunto de empleados supervisados por Santos como se indica a continuación. Las personas supervisadas por Santos son (1) las personas cuyo jefe directo es Santos y (2) las personas cuyo jefe es supervisado por Santos. Obsérvese que el caso (2) es recursivo.

Se puede formular la definición recursiva anterior como una vista recursiva de Datalog, denominada *empl_santos*:

$$\textit{empl_santos}(X) \text{ :– } \textit{jefe}(X, \text{“Santos”})$$
$$\textit{empl_santos}(X) \text{ :– } \textit{jefe}(X, Y), \textit{empl_santos}(Y)$$

La primera regla corresponde al caso (1) y la segunda al caso (2). La vista *empl_santos* depende de sí misma debido a la segunda regla; por tanto, el programa Datalog anterior es recursivo. Se *da por supuesto* que los programas Datalog recursivos no contienen reglas con literales negativos. La razón se verá más adelante. Las notas bibliográficas hacen referencia a artículos que describen dónde se puede usar la negación en los programas Datalog recursivos.

```
procedure PuntoFijo-Datalog
    I = conjunto de hechos de la base de datos
    repeat
        I_Anterior = I
        I = I ∪ inferir(R, I)
    until I = I_Anterior
```

Figura 5.11 Procedimiento PuntoFijo-Datalog.

nombre_empleado	*nombre_jefe*
Alández	Bariego
Bariego	Erice
Corisco	Dalma
Dalma	Santos
Erice	Santos
Santos	Marchamalo
Rienda	Marchamalo

Figura 5.12 La relación *jefe*.

Número de iteración	Tuplas de *empl_santos*
0	
1	(Dalma), (Erice)
2	(Dalma), (Erice), (Bariego), (Corisco)
3	(Dalma), (Erice), (Bariego), (Corisco), (Alández)
4	(Dalma), (Erice), (Bariego), (Corisco), (Alández)

Figura 5.13 Subordinados de Santos en las distintas iteraciones del procedimiento PuntoFijo-Datalog.

Las vistas de los programas recursivos que contienen un conjunto de reglas $\mathcal{R}$ se definen para que contengan exactamente el conjunto de hechos I calculados por el procedimiento iterativo PuntoFijo-Datalog de la Figura 5.11. La recursividad del programa Datalog se ha transformado en la iteración del procedimiento. Al final del procedimiento, $\mathit{inferir}(\mathcal{R}, I) \cup D = I$, donde D es el conjunto de hechos de la base de datos, e I se denomina **punto fijo** del programa.

Considérese el programa que define *empl_santos* con la relación *jefe*, como en la Figura 5.12. El conjunto de hechos calculado para la vista *empl_santos* en cada iteración aparece en la Figura 5.13. En cada iteración el programa calcula otro nivel de empleados bajo el mando de Santos y los añade al conjunto *empl_santos*. El procedimiento termina cuando no se produce ningún cambio en el conjunto *empl_santos*, lo cual detecta el sistema al descubrir que $I = I_Anterior$. Como el conjunto de empleados y jefes es finito, se tiene que alcanzar este punto fijo. En la relación *jefe* dada, el procedimiento PuntoFijo-Datalog termina después de la cuarta iteración, al detectar que no se ha inferido ningún hecho nuevo.

Conviene comprobar que, al final de la iteración, la vista *empl_santos* contiene exactamente los empleados que trabajan bajo la supervisión de Santos. Para obtener los nombres de los empleados supervisados por Santos definidos por la vista se puede usar la consulta:

$$? \mathit{empl_santos}(N)$$

Para comprender el procedimiento PuntoFijo-Datalog hay que recordar que cada regla infiere hechos nuevos a partir de un conjunto de hechos dado. La iteración comienza con un conjunto de hechos I que coincide con los hechos de la base de datos. Se sabe que todos estos hechos son ciertos, pero puede haber otros hechos que también sean ciertos[1]. A continuación se usa el conjunto de reglas $\mathcal{R}$ del programa Datalog para inferir los hechos que son ciertos, dado que los hechos de I lo son. Los hechos inferidos se añaden a I y se vuelven a usar las reglas para hacer más inferencias. Este proceso se repite hasta que no se puedan inferir hechos nuevos.

Se puede demostrar para los programas Datalog seguros que existe un punto a partir del cual no se pueden obtener más hechos nuevos; es decir, para algún k, $I_{k+1} = I_k$. En ese punto, por tanto, se tiene el conjunto final de hechos ciertos. Además, dado un programa Datalog y una base de datos, el procedimiento de punto fijo infiere todos los hechos que se puede inferir que son ciertos.

1. La palabra "hecho" se usa en un sentido técnico para denotar la pertenencia de una tupla a una relación. Así, en el sentido de Datalog para "hecho", un hecho puede ser cierto (la tupla está realmente en la relación) o falso (la tupla no está en la relación).

Si un programa recursivo contiene una regla con un literal negativo, puede surgir el problema siguiente. Recuérdese que cuando se hace una inferencia empleando el ejemplar básico de una regla, por cada literal negativo **not** q en el cuerpo de la regla hay que comprobar que q no esté presente en el conjunto de hechos I. Esta comprobación da por supuesto que q no se puede inferir posteriormente. Sin embargo, en la iteración de punto fijo, el conjunto de hechos I crece en cada iteración y, aunque q no esté presente en I en una iteración dada, puede aparecer más tarde. Por tanto, se puede haber hecho una inferencia en una iteración que ya no se pueda hacer en una iteración anterior, y la inferencia sería incorrecta. Se exige que los programas recursivos no contengan literales negativos para evitar estos problemas.

En lugar de crear una vista para los empleados supervisados por un jefe concreto, por ejemplo Santos, se puede crear una vista más general, *empl*, que contenga todas las tuplas (X, Y) tales que X sea directa o indirectamente supervisado por Y, usando el siguiente programa (que también puede verse en la Figura 5.8):

$$\begin{aligned} &\textit{empl}(X, Y) \ \text{:–}\ \textit{jefe}(X, Y) \\ &\textit{empl}(X, Y) \ \text{:–}\ \textit{jefe}(X, Z), \textit{empl}(Z, Y) \end{aligned}$$

Para determinar los subordinados directos o indirectos de Santos, se usa simplemente la consulta:

$$?\ \textit{empl}(X, \text{“Santos”})$$

que devuelve para X el mismo conjunto de valores que la vista *empl_santos.* La mayoría de las implementaciones de Datalog cuenta con sofisticados optimizadores de consultas y motores de evaluación que pueden ejecutar la consulta anterior aproximadamente a la misma velocidad que evaluarían la vista *empl_santos.* La vista *empl* definida anteriormente se denomina **cierre transitivo** de la relación *jefe.* Si se sustituyera la relación *jefe* por cualquier otra relación binaria R, el programa anterior definiría el cierre transitivo de R.

5.4.7 La potencia de la recursividad

Datalog con recursividad tiene mayor potencia expresiva que Datalog sin recursividad. En otras palabras, existen consultas a la base de datos que se pueden resolver usando recursividad, pero que no se pueden resolver sin usarla. Por ejemplo, en Datalog no se puede expresar el cierre transitivo sin usar la recursividad (como, por cierto, en SQL o QBE sin recursividad). Considérese el cierre transitivo de la relación *jefe*. Intuitivamente, un número fijo de reuniones sólo puede determinar los empleados que están un número fijo (diferente) de niveles por debajo de cualquier jefe (no se intentará demostrar esta afirmación aquí). Como cualquier consulta no recursiva tiene un número fijo de reuniones, existe un límite al número de niveles de empleados que la consulta puede determinar. Si el número de niveles de empleados de la relación *jefe* es mayor que el límite de la consulta, la consulta no encontrará algún nivel de empleados. Por tanto, los programas Datalog no recursivos no pueden expresar el cierre transitivo.

Una alternativa a la recursividad es usar un mecanismo externo, como SQL incorporado, para iterar por las consultas no recursivas. La iteración implementa el bucle del algoritmo de punto fijo de la Figura 5.11. De hecho, así es como se implementa ese tipo de consultas en los sistemas de bases de datos que no permiten la recursividad. Sin embargo, la formulación de estas consultas empleando la iteración es más complicada que si se usa la recursividad, y la evaluación usando recursividad puede optimizarse para que se ejecute más rápidamente que la evaluación mediante iteración.

La potencia expresiva proporcionada por la recursividad debe usarse con cuidado. Resulta relativamente sencillo escribir programas recursivos que generen un número infinito de hechos, como se muestra en este programa:

$$\begin{aligned} &\textit{número}(0) \\ &\textit{número}(A) \ \text{:–}\ \textit{número}(B), A = B + 1 \end{aligned}$$

El programa genera $número(n)$ para todos los enteros n positivos, que es claramente un conjunto infinito y no termina nunca. La segunda regla del programa no cumple la condición de seguridad descrita en

el Apartado 5.4.4. Los programas que cumplen la condición de seguridad terminan, aunque sean recursivos, siempre que todas las relaciones de la base de datos sean finitas. Para programas de este tipo, las tuplas de las vistas sólo pueden contener constantes de la base de datos y, por ello, las vistas son finitas. Lo opuesto no es cierto; es decir, hay programas que no cumplen la condición de seguridad, pero que finalizan.

El procedimiento PuntoFijo-Datalog usa de manera iterativa la función *inferir*$(\mathcal{R}, I)$ para calcular los hechos que son ciertos dado un programa Datalog recursivo. Aunque sólo se ha considerado el caso de los programas Datalog sin literales negativos, el procedimiento se puede emplear también en vistas definidas en otros lenguajes, como SQL o el álgebra relacional, siempre que las vistas cumplan las condiciones que se describen a continuación. Independientemente del lenguaje empleado para definir una vista V, esa vista se puede considerar definida por una expresión E_V que, dado un conjunto de hechos I, devuelve un conjunto de hechos $E_V(I)$ para la vista V. Dado un conjunto de definiciones de vistas $\mathcal{R}$ (en cualquier lenguaje) se puede definir la función *inferir* $(\mathcal{R}, I)$ que devuelva $I \cup \bigcup_{V \in \mathcal{R}} E_V(I)$. La función anterior es parecida a la función *inferir* de Datalog.

Se dice que una vista V es **monótona** si, dado cualquier par de conjuntos de hechos I_1 e I_2 tales que $I_1 \subseteq I_2$, entonces $E_V(I_1) \subseteq E_V(I_2)$, donde E_V es la expresión usada para definir V. Análogamente, se dice que la función *inferir* es monótona si:

$$I_1 \subseteq I_2 \Rightarrow \textit{inferir}(\mathcal{R}, I_1) \subseteq \textit{inferir}(\mathcal{R}, I_2)$$

Por tanto, si *inferir* es monótona, dado un conjunto de hechos I_0 que es un subconjunto de los hechos ciertos, se puede asegurar que todos los hechos de *inferir*$(\mathcal{R}, I_0)$ también son ciertos. Usando el mismo razonamiento del Apartado 5.4.6 se puede demostrar que el procedimiento PuntoFijo-Datalog es correcto (es decir, sólo calcula hechos ciertos) siempre que la función *inferir* sea monótona.

Las expresiones del álgebra relacional que sólo usan los operadores $\Pi, \sigma, \times, \bowtie, \cup, \cap,$ o ρ son monótonas. Se pueden definir vistas recursivas usando estas expresiones.

Sin embargo, las expresiones relacionales que usan el operador $-$ no son monótonas. Por ejemplo, sean $jefe_1$ y $jefe_2$ relaciones con el mismo esquema que la relación *jefe*. Sea

$$\begin{aligned} I_1 = \{\ & jefe_1(\text{“Alández”, “Bariego”}), jefe_1(\text{“Bariego”, “Erice”}), \\ & jefe_2(\text{“Alández”, “Bariego”})\ \} \end{aligned}$$

y sea

$$\begin{aligned} I_2 = \{\ & jefe_1(\text{“Alández”, “Bariego”}), jefe_1(\text{“Bariego”, “Erice”}), \\ & jefe_2(\text{“Alández”, “Bariego”}), jefe_2(\text{“Bariego”, “Erice”})\} \end{aligned}$$

Considérese la expresión $jefe_1 - jefe_2$. El resultado de la expresión anterior sobre I_1 es (“Bariego”, “Erice”), mientras que el resultado de la expresión sobre I_2 es la relación vacía. Pero, como $I_1 \subseteq I_2$, la expresión no es *monótona*. Las expresiones que usan el operador de agrupación del álgebra relacional extendido tampoco son monótonas.

La técnica del punto fijo no funciona sobre las vistas recursivas definidas con expresiones no monótonas. No obstante, existen situaciones en las que ese tipo de vistas es útil, especialmente para la definición de agregaciones en las relaciones “objeto–componente”. Ese tipo de relaciones define los componentes que constituyen cada objeto. Los componentes pueden estar formados, a su vez, por muchos otros componentes, y así sucesivamente; por tanto, las relaciones, como la relación $jefe$, tienen una estructura recursiva natural. Un ejemplo de consulta de agregación sobre una estructura de ese tipo es el cálculo del número total de componentes de cada objeto. Escribir esta consulta en Datalog o en SQL (sin las extensiones procedimentales) exigiría el empleo de una vista recursiva sobre una expresión no monótona. Las notas bibliográficas contienen referencias sobre la investigación acerca de la definición de este tipo de vistas.

Es posible definir algunos tipos de consultas recursivas sin usar vistas. Por ejemplo, se han propuesto operaciones relacionales extendidas para definir el cierre transitivo, y extensiones de la sintaxis de SQL para especificar el cierre transitivo (generalizado). Sin embargo, las definiciones de vistas recursivas proporcionan una mayor potencia expresiva que las demás formas de consultas recursivas.

5.5 Resumen

- El **cálculo relacional de tuplas** y el **cálculo relacional de dominios** son lenguajes no procedimentales que representan la potencia básica necesaria en un lenguaje de consultas relacionales. El álgebra relacional básica es un lenguaje procedimental que es equivalente en potencia a ambas formas del cálculo relacional cuando se limitan a expresiones seguras.
- Los cálculos relacionales son lenguajes rígidos y formales que no resultan adecuados para los usuarios ocasionales de los sistemas de bases de datos. Los sistemas comerciales de bases de datos, por tanto, usan lenguajes con más "azúcar sintáctico". Se han considerado dos lenguajes de consultas: QBE y Datalog.
- QBE está basado en un paradigma visual: las consultas tienen un aspecto muy parecido a tablas.
- QBE y sus variantes se han hecho populares entre los usuarios poco expertos de bases de datos, debido a la simplicidad intuitiva del paradigma visual. El muy usado sistema de bases de datos Access de Microsoft soporta una versión gráfica de QBE denominada GQBE.
- Datalog procede de Prolog pero, a diferencia de éste, tiene una semántica declarativa que hace que las consultas sencillas sean más fáciles de formular y que la evaluación de las consultas resulte más fácil de optimizar.
- La definición de las vistas resulta especialmente sencilla en Datalog, y las vistas recursivas que permite Datalog hacen posible la formulación de consultas, tales como el cierre transitivo, que no podrían formularse sin usar la recursividad o la iteración. Sin embargo, en Datalog no hay normas aceptadas para características importantes como la agrupación y la agregación. Datalog sigue siendo, principalmente, un lenguaje de investigación.

Términos de repaso

- Cálculo relacional de tuplas.
- Cálculo relacional de dominios.
- Seguridad de las expresiones.
- Potencia expresiva de los lenguajes.
- Query-by-Example (QBE, consulta mediante ejemplos).
- Sintaxis bidimensional.
- Esqueletos de tablas.
- Filas de ejemplo.
- Cuadro de condiciones.
- Relación resultado.
- Microsoft Access.
- Graphical Query-By-Example (GQBE, consulta gráfica mediante ejemplos).
- Cuadrícula de diseño.
- Datalog.
- Reglas.
- Usa.
- Define.
- Literal positivo.
- Literal negativo.
- Hecho.
- Regla:
  - ☐ Cabeza.
  - ☐ Cuerpo.
- Programa Datalog.
- Depende:
  - ☐ Directamente.
  - ☐ Indirectamente.
- Vista recursiva.
- Vista no recursiva.
- Ejemplares:
  - ☐ Básicos.
  - ☐ Satisfacer.
- Inferir.
- Semántica:
  - ☐ De las reglas.
  - ☐ De los programas.
- Seguridad.
- Punto fijo.
- Cierre transitivo.
- Definición de vistas monótonas.

Ejercicios prácticos

5.1 Sean los siguientes esquemas de relaciones:

$$R = (A, B, C)$$
$$S = (D, E, F)$$

Sean las relaciones *r*(*R*) y *s*(*S*). Dese una expresión del cálculo relacional de tuplas que sea equivalente a cada una de las siguientes:

a. $\Pi_A(r)$
b. $\sigma_{B=17}(r)$
c. $r \times s$
d. $\Pi_{A,F}(\sigma_{C=D}(r \times s))$

5.2 Sea $R = (A, B, C)$ y sean r_1 y r_2 relaciones del esquema R. Dese una expresión del cálculo relacional de dominios que sea equivalente a cada una de las siguientes:

a. $\Pi_A(r_1)$
b. $\sigma_{B=17}(r_1)$
c. $r_1 \cup r_2$
d. $r_1 \cap r_2$
e. $r_1 - r_2$
f. $\Pi_{A,B}(r_1) \bowtie \Pi_{B,C}(r_2)$

5.3 Sean $R = (A, B)$ y $S = (A, C)$ y sean $r(R)$ y $s(S)$ relaciones. Escríbanse expresiones de QBE y de Datalog para cada una de las consultas siguientes:

a. $\{< a > \mid \exists\, b\, (< a, b > \in r \wedge b = 7)\}$
b. $\{< a, b, c > \mid < a, b > \in r \wedge < a, c > \in s\}$
c. $\{< a > \mid \exists\, c\, (< a, c > \in s \wedge \exists\, b_1, b_2\, (< a, b_1 > \in r \wedge < c, b_2 > \in r \wedge b_1 > b_2))\}$

5.4 Considérese la base de datos relacional de la Figura 5.14, en la que se han subrayado las claves primarias. Proporciónese una expresión de Datalog para cada una de las siguientes consultas:

a. Determinar todos los empleados que trabajan (directa o indirectamente) a las órdenes del jefe "Santos".
b. Determinar las ciudades de residencia de todos los empleados que trabajan (directa o indirectamente) a las órdenes del jefe "Santos".
c. Determinar todas las parejas de empleados que tienen un jefe (directo o indirecto) en común.
d. Determinar todas las parejas de empleados que tienen un jefe (directo o indirecto) en común y que están el mismo número de niveles por debajo de ese jefe.

5.5 Descríbase cómo una regla Datalog arbitraria puede expresarse como una vista del álgebra relacional extendida.

Ejercicios

5.6 Considérese la base de datos de empleados de la Figura 5.14. Proporciónense expresiones del cálculo relacional de tuplas para cada una de las consultas siguientes:

a. Determinar el nombre de todos los empleados que trabajan en el Banco Importante.
b. Determinar el nombre y la ciudad de residencia de todos los empleados que trabajan en el Banco Importante.
c. Determinar el nombre, la dirección y la ciudad de residencia de todos los empleados que trabajan en el Banco Importante y ganan más de 10.000 € anuales.
d. Determinar todos los empleados que viven en la ciudad en la que se ubica la empresa para la que trabajan.
e. Determinar todos los empleados que viven en la misma ciudad y en la misma calle que su jefe.
f. Determinar todos los empleados de la base de datos que no trabajan en el Banco Importante.
g. Determinar todos los empleados que ganan más que cualquier empleado del Banco Pequeño.

empleado (*nombre_empleado*, *calle*, *ciudad*)
trabaja (*nombre_empleado*, *nombre_empresa*, *sueldo*)
empresa (*nombre_empresa*, *ciudad*)
jefe (*nombre_empleado*, *nombre_jefe*)

Figura 5.14 Base de datos de empleados.

h. Supóngase que las empresas pueden tener sede en varias ciudades. Determinar todas las empresas con sede en todas las ciudades en las que tiene sede el Banco Pequeño.

5.7 Sea $R = (A, B)$ y $S = (A, C)$ y sean $r(R)$ y $s(S)$ relaciones. Escríbanse expresiones del álgebra relacional equivalentes a cada una de las expresiones siguientes del cálculo relacional de dominios:

a. $\{< a > \mid \exists\, b\, (< a, b > \in r \wedge b = 17)\}$
b. $\{< a, b, c > \mid < a, b > \in r \wedge < a, c > \in s\}$
c. $\{< a > \mid \exists\, b\, (< a, b > \in r) \vee \forall\, c\, (\exists\, d\, (< d, c > \in s) \Rightarrow < a, c > \in s)\}$
d. $\{< a > \mid \exists\, c\, (< a, c > \in s \wedge \exists\, b_1, b_2\, (< a, b_1 > \in r \wedge < c, b_2 > \in r \wedge b_1 > b_2))\}$

5.8 Repítase el Ejercicio 5.7 escribiendo consultas SQL en lugar de expresiones del álgebra relacional.

5.9 Sea $R = (A, B)$ y $S = (A, C)$ y sean $r(R)$ y $s(S)$ relaciones. Usando la constante especial *nulo*, escríbanse expresiones del cálculo relacional de tuplas equivalentes a cada una de las siguientes:

a. $r \bowtie\!\!\!\!\sqsubset s$
b. $r \sqsupset\!\!\!\!\bowtie\!\!\!\!\sqsubset s$
c. $r \sqsupset\!\!\!\!\bowtie s$

5.10 Considérese la base de datos de seguros de la Figura 5.15, en la que las claves primarias se han subrayado. Formúlense las siguientes consultas en GQBE para esta base de datos relacional:

a. Determinar el número total de personas que poseen coches que se hayan visto involucrados en accidentes en 2005.
b. Determinar el número de accidentes en los cuales se ha visto involucrado algún coche perteneciente a "Martín Gómez".

5.11 Dese una expresión del cálculo relacional de tuplas para determinar el valor máximo de $r(A)$.

5.12 Repítase el ejercicio 5.6 empleando QBE y Datalog.

5.13 Sea $R = (A, B, C)$ y sean r_1 y r_2 relaciones del esquema R. Dense expresiones en QBE y Datalog equivalentes a cada una de las consultas siguientes:

a. $r_1 \cup r_2$
b. $r_1 \cap r_2$
c. $r_1 - r_2$
d. $\Pi_{AB}(r_1) \bowtie \Pi_{BC}(r_2)$

5.14 Escríbase una vista del álgebra relacional extendida equivalente a la regla Datalog

$$p(A, C, D) \text{ :- } q1(A, B), q2(B, C), q3(4, B), D = B + 1$$

persona (*número_carné*, *nombre*, *dirección*)
coche (*matrícula*, *modelo*, *año*)
accidente (*número_parte*, *fecha*, *lugar*)
es_dueño (*número_carné*, *matrícula*)
participó (*número_carné*, *coche*, *número_parte*, *importe_daños*)

Figura 5.15 Base de datos de seguros.

Notas bibliográficas

La definición original del cálculo relacional de tuplas aparece en Codd [1972]. Hay una prueba formal de la equivalencia del cálculo relacional de tuplas y del álgebra relacional en Codd [1972]. Se han propuesto varias extensiones del cálculo relacional de tuplas. Klug [1982] y Escobar-Molano et al. [1993] describen extensiones para funciones de agregación escalares.

El sistema de bases de datos QBE se desarrolló en el Centro de Investigación T. J. Watson de IBM en los primeros años 1970. El lenguaje de manipulación de datos de QBE se usó más tarde en el Servicio de Gestión de Consultas (Query Management Facility, QMF) de IBM. La versión original de Query-by-Example se describe en Zloof [1977]. Otras implementaciones de QBE son Access de Microsoft y Paradox de Borland (que ya no tiene soporte técnico).

Ullman [1988] y Ullman [1989] ofrecen amplias explicaciones teóricas sobre los lenguajes lógicos de consultas y sus técnicas de implementación. Ramakrishnan y Ullman [1995] ofrecen un resumen más actual de las bases de datos deductivas.

A los programas Datalog que tienen tanto recursividad como negación se les puede asignar una semántica sencilla si las negaciones están "estratificadas"—es decir, si no hay ninguna recursividad que afecte a las negaciones. Chandra y Harel [1982] y Apt y Pugin [1987] estudian las negaciones estratificadas. Una extensión importante, denominada *semántica de la estratificación modular*, que maneja una clase de programas recursivos con literales negativos, se estudia en Ross [1990]; Ramakrishnan et al. [1992] describe una técnica de evaluación para este tipo de programas.

Herramientas

QBE de Access de Microsoft es actualmente la implementación de QBE más usada. Las ediciones QMF y Everywhere de DB2 de IBM también soportan QBE.

El sistema Coral de la Universidad de Wisconsin–Madison (www.cs.wisc.edu/coral) es una implementación de Datalog. El sistema XSB de la Universidad Estatal de Nueva York (SUNY, State University of New York) en Stony Brook (xsb.sourceforge.net) es una implementación de Prolog muy usada que soporta consultas a las bases de datos; recuérdese que Datalog es un subconjunto no procedimental de Prolog.

PARTE 2

Diseño de bases de datos

Los sistemas de bases de datos están diseñados para gestionar grandes cantidades de información. Estas grandes cantidades de información no se encuentran aisladas. Forman parte del funcionamiento de alguna empresa cuyo producto final puede ser la información obtenida de la base de datos o algún producto o servicio para el que la base de datos sólo desempeña un papel secundario.

Los dos primeros capítulos de esta parte se centran en el diseño de los esquemas de las bases de datos. El modelo entidad-relación (E-R) descrito en el Capítulo 6 es un modelo de datos de alto nivel. En lugar de representar todos los datos en tablas, distingue entre los objetos básicos, denominados *entidades*, y las *relaciones* entre esos objetos. Suele utilizarse como un primer paso en el diseño de los esquemas de las bases de datos.

El diseño de las bases de datos relacionales—el diseño del esquema relacional—se trató informalmente en los capítulos anteriores. No obstante, existen criterios para distinguir los buenos diseños de bases de datos de los malos. Éstos se formalizan mediante varias "formas normales" que ofrecen diferentes compromisos entre la posibilidad de inconsistencias y la eficiencia de determinadas consultas. El Capítulo 7 describe el diseño formal de los esquemas de las relaciones.

El diseño de un entorno completo de aplicaciones para bases de datos, que responda a las necesidades de la empresa que se modela, exige prestar atención a un conjunto de aspectos más amplio, muchos de los cuales se tratan en el Capítulo 8. Este capítulo describe las interfaces para bases de datos basadas en Web y amplía el estudio de capítulos anteriores sobre la integridad de los datos y la seguridad.

CAPÍTULO 6

Diseño de bases de datos y el modelo E-R

Hasta este punto se ha dado por supuesto un determinado esquema de la base de datos y se ha estudiado la manera de expresar las consultas y las actualizaciones. Ahora se va a considerar en primer lugar la manera de diseñar el esquema de la base de datos. En este capítulo nos centraremos en el modelo de datos entidad-relación (E-R), que ofrece una manera de identificar las entidades que se van a representar en la base de datos y el modo en que se relacionan entre sí. Finalmente, el diseño de la base de datos se expresará en términos del diseño de bases de datos relacionales y del conjunto de restricciones asociado. En este capítulo se mostrará la manera en que el diseño E-R puede transformarse en un conjunto de esquemas de relación y el modo en que se pueden incluir algunas de las restricciones en ese diseño. Luego, en el Capítulo 7, se considerará en detalle si el conjunto de esquemas de relación representa un diseño de la base de datos bueno o malo y se estudiará el proceso de creación de buenos diseños usando un conjunto de restricciones más amplio. Estos dos capítulos tratan los conceptos fundamentales del diseño de bases de datos.

6.1 Visión general del proceso de diseño

La tarea de creación de aplicaciones de bases de datos es una labor compleja, que implica varias fases, como el diseño del esquema de la base de datos, el diseño de los programas que tienen acceso a los datos y los actualizan y el diseño del esquema de seguridad para controlar el acceso a los datos. Las necesidades de los usuarios desempeñan un papel central en el proceso de diseño. En este capítulo nos centraremos en el diseño del esquema de la base de datos, aunque se esbozarán brevemente algunas de las otras tareas más adelante en este capítulo.

El diseño de un entorno completo de aplicaciones de bases de datos que responda a las necesidades de la empresa que se está modelando exige prestar atención a un amplio conjunto de consideraciones. Estos aspectos adicionales del uso esperado de la base de datos influyen en gran variedad de opciones de diseño en los niveles físico, lógico y de vistas.

6.1.1 Fases del diseño

Para aplicaciones pequeñas puede resultar factible para un diseñador de bases de datos que comprenda los requisitos de la aplicación decidir directamente sobre las relaciones que hay que crear, sus atributos y las restricciones sobre las relaciones. Sin embargo, un proceso de diseño tan directo resulta difícil para las aplicaciones reales, ya que a menudo son muy complejas. Frecuentemente no existe una sola persona que comprenda todas las necesidades de datos de la aplicación. El diseñador de la base de datos debe interactuar con los usuarios para comprender las necesidades de la aplicación, realizar una representación de alto nivel de esas necesidades que pueda ser comprendida por los usuarios y luego traducir esos requisitos a niveles inferiores del diseño. Los modelos de datos de alto nivel sirven a los diseñadores de bases de datos ofreciéndoles un marco conceptual en el que especificar de forma

sistemática los requisitos de datos de los usuarios de la base de datos y una estructura para la base de datos que satisfaga esos requisitos.

- La fase inicial del diseño de las bases de datos es la caracterización completa de las necesidades de datos de los posibles usuarios de la base de datos. El diseñador de la base de datos debe interactuar intensamente con los expertos y los usuarios del dominio para realizar esta tarea. El resultado de esta fase es una especificación de requisitos del usuario. Aunque existen técnicas para representar en diagramas los requisitos de los usuarios, en este capítulo sólo se describirán textualmente esos requisitos, que se ilustrarán posteriormente en el Apartado 6.8.2.
- A continuación, el diseñador elige el modelo de datos y, aplicando los conceptos del modelo de datos elegido, traduce estos requisitos en un esquema conceptual de la base de datos. El esquema desarrollado en esta fase de **diseño conceptual** proporciona una visión detallada de la empresa. Se suele emplear el modelo entidad-relación, que se estudiará en el resto de este capítulo, para representar el diseño conceptual. En términos del modelo entidad-relación, el esquema conceptual especifica las entidades que se representan en la base de datos, sus atributos, las relaciones entre ellas y las restricciones que las afectan. Generalmente, la fase de diseño conceptual da lugar a la creación de un diagrama entidad-relación que ofrece una representación gráfica del esquema.

  El diseñador revisa el esquema para confirmar que realmente se satisfacen todos los requisitos y que no entran en conflicto entre sí. También puede examinar el diseño para eliminar características redundantes. Su atención en este momento se centra en describir los datos y sus relaciones, más que en especificar los detalles del almacenamiento físico.
- Un esquema conceptual completamente desarrollado indica también los requisitos funcionales de la empresa. En la **especificación de requisitos funcionales** los usuarios describen los tipos de operaciones (o transacciones) que se llevarán a cabo sobre los datos. Algunos ejemplos de operaciones son la modificación o actualización de datos, la búsqueda y recuperación de datos concretos y el borrado de datos. En esta fase de diseño conceptual el diseñador puede revisar el esquema para asegurarse de que satisface los requisitos funcionales.
- El proceso de paso desde el modelo abstracto de datos a la implementación de la base de datos se divide en de dos fases de diseño finales.
  - En la **fase de diseño lógico** el diseñador traduce el esquema conceptual de alto nivel al modelo de datos de la implementación del sistema de bases de datos que se va a usar. El modelo de implementación de los datos suele ser el modelo relacional, y este paso suele consistir en la traducción del esquema conceptual definido mediante el modelo entidad-relación en un esquema de relación.
  - Finalmente, el diseñador usa el esquema de base de datos resultante propio del sistema en la siguiente **fase de diseño físico**, en la que se especifican las características físicas de la base de datos. Entre estas características están la forma de organización de los archivos y las estructuras de almacenamiento interno; se estudian en el Capítulo 11.

El esquema físico de la base de datos puede modificarse con relativa facilidad una vez creada la aplicación. Sin embargo, las modificaciones del esquema lógico suelen resultar más difíciles de llevar a cabo, ya que pueden afectar a varias consultas y actualizaciones dispersas por todo el código de la aplicación. Por tanto, es importante llevar a cabo con cuidado la fase de diseño de la base de datos antes de crear el resto de la aplicación de bases de datos.

6.1.2 Alternativas de diseño

Una parte importante del proceso de diseño de las bases de datos consiste en decidir la manera de representar en el diseño los diferentes tipos de "cosas", como personas, lugares, productos y similares. Se usa el término *entidad* para hacer referencia a cualquiera de esos elementos claramente identificables. Las diferentes entidades tienen algunos aspectos en común y algunas diferencias. Se desea aprovechar los aspectos en común para tener un diseño conciso, fácilmente comprensible, pero hay que conservar la suficiente flexibilidad como para representar las diferencias entre las diferentes entidades existentes en

el momento del diseño o que se puedan materializar en el futuro. Las diferentes entidades se relacionan entre sí de diversas maneras, todas las cuales deben incluirse en el diseño de la base de datos.

Al diseñar el esquema de una base de datos hay que asegurarse de que se evitan dos peligros importantes:

1. **Redundancia**. Un mal diseño puede repetir información. En el ejemplo bancario que se ha venido usando, se tiene una relación con la información sobre los clientes y una relación separada con la información sobre las cuentas. Supóngase que, en lugar de eso, se repitiera toda la información sobre los clientes (nombre, dirección, etc.) una vez por cada cuenta o préstamo que tuviera cada cliente. Evidentemente, sería redundante. Lo ideal sería que la información apareciera exactamente en un solo lugar.
2. **Incompletitud**. Un mal diseño puede hacer que determinados aspectos de la empresa resulten difíciles o imposibles de modelar. Por ejemplo, supóngase que se usa un diseño de base de datos para el escenario bancario que almacena la información del nombre y de la dirección del cliente con cada cuenta y con cada préstamo, pero que no tiene una relación separada para los clientes. Resultaría imposible introducir el nombre y la dirección de los clientes nuevos a menos que ya tuvieran abierta una cuenta o concedido un préstamo. Se podría intentar salir del paso con este diseño problemático almacenando valores nulos para la información de las cuentas o de los préstamos, como puede ser el número de cuenta o el importe del préstamo. Este parche no sólo resulta poco atractivo, sino que puede evitarse mediante restricciones de clave primaria.

Evitar los malos diseños no es suficiente. Puede haber gran número de buenos diseños entre los que haya que escoger. Como ejemplo sencillo, considérese un cliente que compra un producto. ¿La venta de este producto es una relación entre el cliente y el producto? Dicho de otra manera, ¿es la propia venta una entidad que está relacionada con el cliente y con el producto? Esta elección, aunque simple, puede suponer una importante diferencia en cuanto a los aspectos de la empresa que se pueden modelar bien. Considerando la necesidad de tomar decisiones como ésta para el gran número de entidades y de relaciones que hay en las empresas reales, no es difícil ver que el diseño de bases de datos puede constituir un problema arduo. En realidad, se verá que exige una combinación de conocimientos y de "buen gusto".

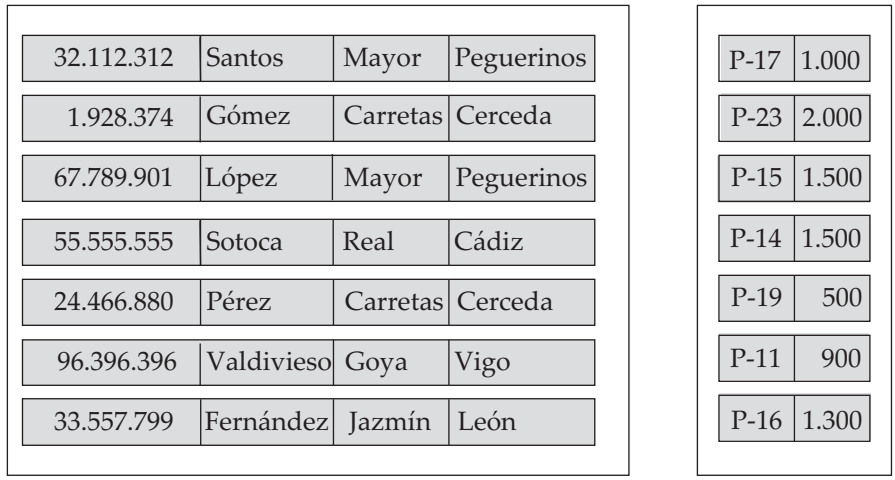
6.2 El modelo entidad-relación

El modelo de datos **entidad–relación (E-R)** se desarrolló para facilitar el diseño de bases de datos permitiendo la especificación de un *esquema de la empresa* que representa la estructura lógica global de la base de datos. El modelo de datos E-R es uno de los diferentes modelos de datos semánticos; el aspecto semántico del modelo radica en la representación del significado de los datos. El modelo E-R resulta muy útil para relacionar los significados e interacciones de las empresas reales con el esquema conceptual. Debido a esta utilidad, muchas herramientas de diseño de bases de datos se basan en los conceptos del modelo E-R. El modelo de datos E-R emplea tres conceptos básicos: los conjuntos de entidades, los conjuntos de relaciones y los atributos.

6.2.1 Conjuntos de entidades

Una **entidad** es una "cosa" u "objeto" del mundo real que es distinguible de todos los demás objetos. Por ejemplo, cada persona de una empresa es una entidad. Una entidad tiene un conjunto de propiedades, y los valores de algún conjunto de propiedades pueden identificar cada entidad de forma unívoca. Por ejemplo, el DNI 67.789.901 identifica unívocamente una persona concreta de la empresa. Análogamente, los préstamos se pueden considerar entidades, y el número de préstamo P-15 de la sucursal de Navacerrada identifica unívocamente una entidad de préstamo. Las entidades pueden ser concretas, como las personas o los libros, o abstractas, como los préstamos, las vacaciones o los conceptos.

Un **conjunto de entidades** es un conjunto de entidades del mismo tipo que comparten las mismas propiedades, o atributos. El conjunto de todas las personas que son clientes en un banco dado, por



32.112.312	Santos	Mayor	Peguerinos
1.928.374	Gómez	Carretas	Cerceda
67.789.901	López	Mayor	Peguerinos
55.555.555	Sotoca	Real	Cádiz
24.466.880	Pérez	Carretas	Cerceda
96.396.396	Valdivieso	Goya	Vigo
33.557.799	Fernández	Jazmín	León

cliente

P-17	1.000
P-23	2.000
P-15	1.500
P-14	1.500
P-19	500
P-11	900
P-16	1.300

préstamo

Figura 6.1 Conjuntos de entidades *cliente* y *préstamo*.

ejemplo, se puede definir como el conjunto de entidades *cliente*. Análogamente, el conjunto de entidades *préstamo* puede representar el conjunto de todos los préstamos concedidos por un banco concreto. Cada una de las entidades que constituyen un conjunto se denomina *extensión* de ese conjunto de entidades. Por tanto, todos los clientes de un banco son la extensión del conjunto de entidades *cliente*.

Los conjuntos de entidades no son necesariamente disjuntos. Por ejemplo, es posible definir el conjunto de entidades de todos los empleados de un banco (*empleado*) y el conjunto de entidades de todos los clientes del banco (*cliente*). Una entidad *persona* puede ser una entidad *empleado*, una entidad *cliente*, ambas cosas, o ninguna.

Cada entidad se representa mediante un conjunto de **atributos**. Los atributos son propiedades descriptivas que posee cada miembro de un conjunto de entidades. La designación de un atributo para un conjunto de entidades expresa que la base de datos almacena información parecida relativa a cada entidad del conjunto de entidades; sin embargo, cada entidad puede tener su propio valor para cada atributo. Posibles atributos del conjunto de entidades *cliente* son *id_cliente*, *nombre_cliente*, *calle_cliente* y *ciudad_cliente.* En la vida real habría más atributos, como el número de la calle, el número del piso la provincia, el código postal, y el país, pero se omiten para no complicar el ejemplo. Posibles atributos del conjunto de entidades *préstamo* son *número_préstamo* e *importe*.

Cada entidad tiene un **valor** para cada uno de sus atributos. Por ejemplo, una entidad *cliente* concreta puede tener el valor 32.112.312 para *id_cliente*, el valor Santos para *nombre_cliente*, el valor Mayor para *calle_cliente* y el valor Peguerinos para *ciudad_cliente*.

El atributo *id_cliente* se usa para identificar unívocamente a los clientes, dado que puede haber más de un cliente con el mismo nombre, calle y ciudad. En Estados Unidos, muchas empresas consideran adecuado usar el número *seguridad_social* de cada persona[1] como atributo cuyo valor identifica unívocamente a esa persona. En general la empresa tendría que crear y asignar un identificador unívoco a cada cliente.

Por tanto, las bases de datos incluyen una serie de conjuntos de entidades, cada una de las cuales contiene cierto número de entidades del mismo tipo. La Figura 6.1 muestra parte de una base de datos de un banco que consta de dos conjuntos de entidades, *cliente* y *préstamo*.

Las bases de datos para entidades bancarias pueden incluir diferentes conjuntos de entidades. Por ejemplo, además del seguimiento de los clientes y de los préstamos, el banco también ofrece cuentas, que se representan mediante el conjunto de entidades *cuenta* con los atributos *número_cuenta* y *saldo*. Además, si el banco tiene varias sucursales, se puede guardar información acerca de todas las sucursales del

1. En España se asigna a cada ciudadano un número único, denominado número del documento nacional de identidad (DNI) para identificarla unívocamente. Se supone que cada persona tiene un único DNI y que no hay dos personas con el mismo DNI.

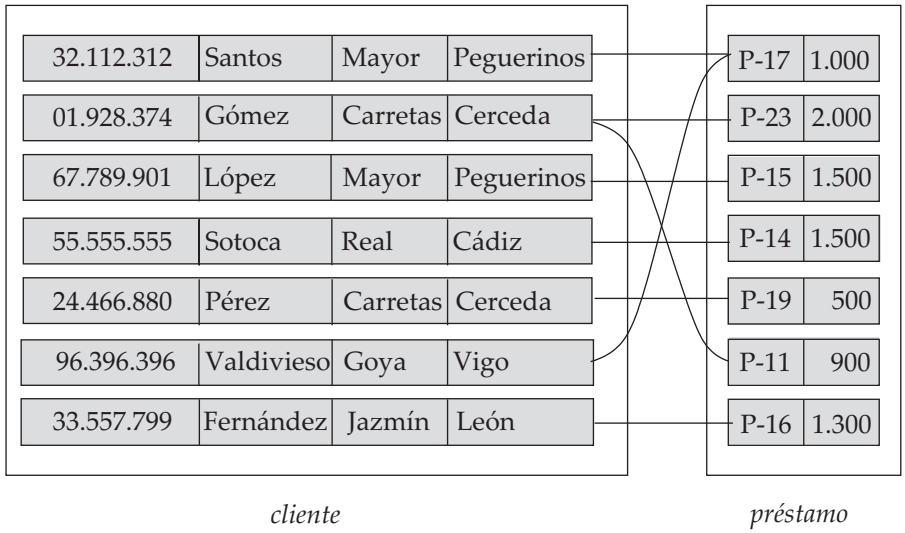


Figura 6.2 Conjunto de relaciones *prestatario*.

banco. Cada conjunto de entidades *sucursal* se puede describir mediante los atributos *nombre_sucursal*, *ciudad_sucursal* y *activos*.

6.2.2 Conjuntos de relaciones

Una **relación** es una asociación entre varias entidades. Por ejemplo, se puede definir una relación que asocie al cliente López con el préstamo P-15. Esta relación especifica que López es un cliente con el préstamo número P-15.

Un **conjunto de relaciones** es un conjunto de relaciones del mismo tipo. Formalmente es una relación matemática con $n \geq 2$ de conjuntos de entidades (posiblemente no distintos). Si $E_1,\ E_2, \ldots, E_n$ son conjuntos de entidades, entonces un conjunto de relaciones R es un subconjunto de:

$$\{(e_1, e_2, \ldots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$$

donde $(e_1, e_2, \ldots, e_n)$ es una relación.

Considérense las dos entidades *cliente* y *préstamo* de la Figura 6.1. Se define el conjunto de relaciones *prestatario* para denotar la asociación entre los clientes y los préstamos bancarios que tienen concedidos. La Figura 6.2 muestra esta asociación.

Como ejemplo adicional, considérense los dos conjuntos de entidades *préstamo* y *sucursal*. Se puede definir el conjunto de relaciones *sucursal_préstamo* para denotar la asociación entre un préstamo y la sucursal en que se concede ese préstamo.

La asociación entre conjuntos de entidades se conoce como *participación*; es decir, los conjuntos de entidades $E_1,\ E_2, \ldots, E_n$ **participan** en el conjunto de relaciones R. Un **ejemplar de la relación** de un esquema E-R representa una asociación entre las entidades citadas en la empresa real que se está modelando. Como ilustración, la entidad *cliente* López, que tiene el identificador de cliente 67.789.901, y la entidad *préstamo* P-15 participan en un ejemplar de la relación *prestatario*. Este ejemplar de relación representa que, en la empresa real, la persona llamada López que tiene el *id_cliente* 67.789.901 tiene concedido el préstamo que está numerado como P-15.

La función que desempeña una entidad en una relación se denomina **rol** de esa entidad. Como los conjuntos de entidades que participan en un conjunto de relaciones, generalmente, son distintos, los roles están implícitos y no se suelen especificar. Sin embargo, resultan útiles cuando el significado de una relación necesita aclaración. Tal es el caso cuando los conjuntos de entidades de una relación no son distintos; es decir, el mismo conjunto de entidades participa en un conjunto de relaciones más de una vez, con diferentes roles. En este tipo de conjunto de relaciones, que se denomina a veces conjunto de relaciones **recursivo**, son necesarios los nombres explícitos para los roles para especificar la manera

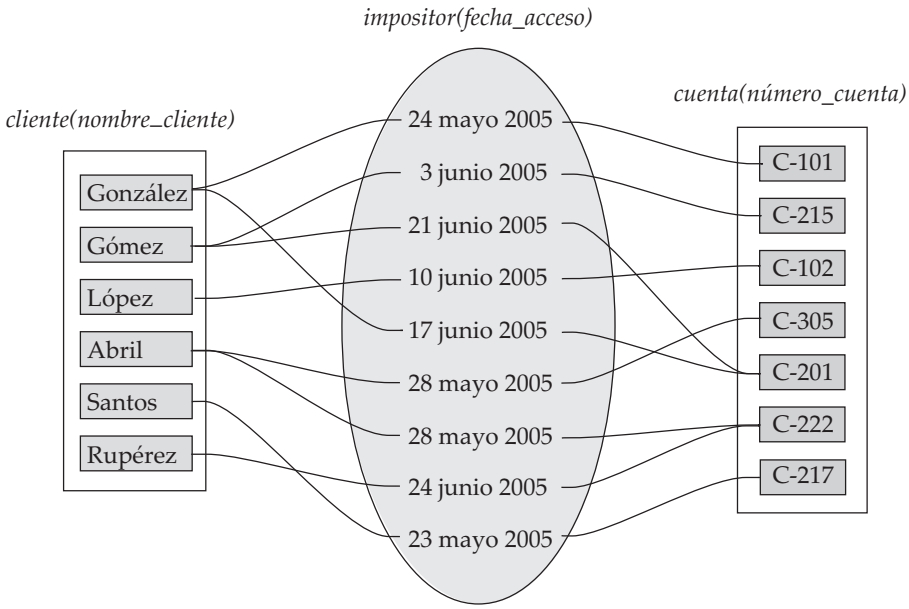


Figura 6.3 *Fecha_acceso* como atributo del conjunto de entidades *impositor*.

en que cada entidad participa en cada ejemplar de la relación. Por ejemplo, considérese un conjunto de entidades *empleado* que almacena información sobre todos los empleados del banco. Se puede tener un conjunto de relaciones *trabaja_para* que se modela mediante pares ordenados de entidades *empleado*. El primer empleado de cada par adopta el rol de *trabajador*, mientras el segundo desempeña el rol de *jefe*. De esta manera, todas las relaciones *trabaja_para* se caracterizan mediante pares (trabajador, jefe); los pares (jefe, trabajador) quedan excluidos.

Una relación puede también tener atributos denominados **atributos descriptivos**. Considérese el conjunto de relaciones *impositor* con los conjuntos de entidades *cliente* y *cuenta*. Se puede asociar el atributo *fecha_acceso* con esta relación para especificar la fecha más reciente de acceso del cliente a la cuenta. La relación *impositor* entre las entidades correspondientes al cliente Santos y la cuenta C-217 tiene el valor "23 mayo 2005" para el atributo *fecha_acceso*, lo que significa que el último día en que Santos accedió a la cuenta C-217 fue el 23 de mayo de 2005.

La Figura 6.3 muestra el conjunto de relaciones *impositor* con el atributo descriptivo *fecha_acceso*; en aras de la sencillez, sólo se muestran algunos atributos de los dos conjuntos de entidades.

Como ejemplo adicional de los atributos descriptivos de las relaciones, supóngase que se tienen los conjuntos de entidades *estudiante* y *asignatura* que participan en el conjunto de relaciones *matriculado*. Puede que se desee almacenar el atributo descriptivo *créditos* con la relación, para registrar si el estudiante se ha matriculado en la asignatura para obtener créditos o sólo como oyente.

Cada ejemplar de una relación de un conjunto de relaciones determinado debe identificarse unívocamente a partir de sus entidades participantes, sin usar los atributos descriptivos. Para comprender esto, supóngase que deseemos representar todas las fechas en las que un cliente ha tenido acceso a una cuenta. El atributo monovalorado *fecha_acceso* sólo puede almacenar una fecha de acceso. No se pueden representar varias fechas de acceso mediante varios ejemplares de la relación entre el mismo cliente y la misma cuenta, ya que los ejemplares de la relación no quedarían identificados unívocamente usando solamente las entidades participantes. La forma correcta de tratar este caso es crear el atributo multivalorado *fechas_acceso*, que puede almacenar todas las fechas de acceso.

Sin embargo, puede haber más de un conjunto de relaciones que implique a los mismos conjuntos de entidades. En nuestro ejemplo, los conjuntos de entidades *cliente* y *préstamo* participan en el conjunto de relaciones *prestatario*. Además, supóngase que cada préstamo deba tener otro cliente que sirva como avalista del préstamo. Entonces los conjuntos de entidades *cliente* y *préstamo* pueden participar en otro conjunto de relaciones: *avalista*.

Los conjuntos de relaciones *prestatario* y *sucursal_préstamo* proporcionan un ejemplo de conjunto de relaciones **binario**—es decir, uno que implica dos conjuntos de entidades. La mayor parte de los conjuntos de relaciones de los sistemas de bases de datos son binarios. A veces, no obstante, los conjuntos de relaciones implican a más de dos conjuntos de entidades.

Por ejemplo, considérense los conjuntos de entidades *empleado*, *sucursal* y *trabajo*. Ejemplos de entidades *trabajo* pueden ser director, cajero, interventor, etc. Las entidades trabajo pueden tener los atributos *puesto* y *nivel*. El conjunto de relaciones *trabaja_en* entre *empleado*, *sucursal* y *trabajo* es un ejemplo de relación ternaria. Una relación ternaria entre Santos, Navacerrada y director indica que Santos trabaja de director de la sucursal de Navacerrada. Santos también podría actuar como interventor de la sucursal de Centro, lo que estaría representado por otra relación. Podría haber otra relación entre Gómez, Centro y cajero, que indicaría que Gómez trabaja de cajero en la sucursal de Centro.

El número de conjuntos de entidades que participan en un conjunto de relaciones es también el **grado** de ese conjunto de relaciones. Los conjuntos de relaciones binarios tienen grado 2; los conjuntos de relaciones ternarios tienen grado 3.

6.2.3 Atributos

Para cada atributo hay un conjunto de valores permitidos, denominados **dominio** o **conjunto de valores** de ese atributo. El dominio del atributo *nombre_cliente* puede ser el conjunto de todas las cadenas de texto de una cierta longitud. Análogamente, el dominio del atributo *número_préstamo* puede ser el conjunto de todas las cadenas de caracteres de la forma "P-n", donde n es un entero positivo.

Formalmente, cada atributo de un conjunto de entidades es una función que asigna el conjunto de entidades a un dominio. Dado que el conjunto de entidades puede tener varios atributos, cada entidad se puede describir mediante un conjunto de pares (atributo,valor), un par por cada atributo del conjunto de entidades. Por ejemplo, una entidad *cliente* concreta se puede describir mediante el conjunto (*id_cliente*, 67.789.901), (*nombre_cliente*, López), (*calle_cliente*, Mayor), (*ciudad_cliente*, Peguerinos), lo que significa que esa entidad describe a una persona llamada López que tiene el DNI número 67.789.901 y que reside en la calle Mayor de Peguerinos. Ahora se puede ver que existe una integración del esquema abstracto con la empresa real que se está modelando. Los valores de los atributos que describen cada entidad constituyen una parte significativa de los datos almacenados en la base de datos.

Cada atributo, tal y como se usa en el modelo E-R, se puede caracterizar por los siguientes tipos de atributo.

- Atributos **simples** y **compuestos**. En los ejemplos considerados hasta ahora los atributos han sido simples; es decir, no estaban divididos en subpartes. Los atributos **compuestos**, en cambio, se pueden dividir en subpartes (es decir, en otros atributos). Por ejemplo, el atributo *nombre* puede estar estructurado como un atributo compuesto consistente en *nombre*, *primer_apellido* y *segundo_apellido*. Usar atributos compuestos en un esquema de diseño es una buena elección si el usuario desea referirse a un atributo completo en algunas ocasiones y, en otras, solamente a algún componente del atributo. Supóngase que hubiera que sustituir para el conjunto de entidades *cliente* los atributos *calle_cliente* y *ciudad_cliente* por el atributo compuesto *dirección*, con los atributos *calle*, *ciudad*, *provincia*, y *código_postal*[2]. Los atributos compuestos ayudan a agrupar atributos relacionados, lo que hace que los modelos sean más claros.

  Obsérvese también que los atributos compuestos pueden aparecer como una jerarquía. En el atributo compuesto dirección, el componente *calle* puede dividirse a su vez en *número_calle*, *nombre_calle* y *piso*. La Figura 6.4 muestra estos ejemplos de atributos compuestos para el conjunto de entidades *cliente*.

- Atributos **monovalorados** y **multivalorados**. Todos los atributos que se han especificado en los ejemplos anteriores tienen un único valor para cada entidad concreta. Por ejemplo, el atributo *número_préstamo* para una entidad préstamo específica hace referencia a un único número de préstamo. Se dice que estos atributos son **monovalorados**. Puede haber ocasiones en las que

2. Se asume el formato de *calle_cliente* y *dirección* usado en España, que incluye un código postal numérico llamado "código postal".

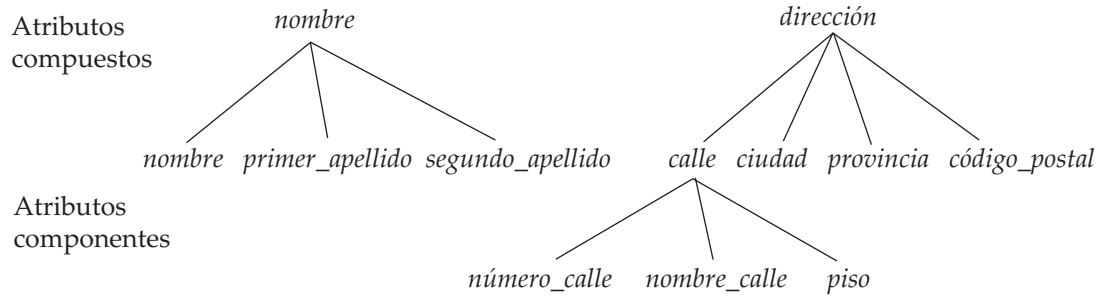


Figura 6.4 Atributos compuestos *nombre* y *dirección*.

un atributo tenga un conjunto de valores para una entidad concreta. Considérese un conjunto de entidades *empleado* con el atributo *número_teléfono.* Cada empleado puede tener cero, uno o varios números de teléfono, y empleados diferentes pueden tener diferente cantidad de teléfonos. Se dice que este tipo de atributo es **multivalorado**. Como ejemplo adicional, el atributo *nombre_subordinado* del conjunto de entidades *empleado* es multivalorado, ya que cada empleado podría tener cero, uno o más subordinados.

Si resulta necesario, se pueden establecer apropiadamente límites inferior y superior al número de valores en el atributo **multivalorado**. Por ejemplo, el banco puede limitar a dos el número de números de teléfono que se guardan por cliente. El establecimiento de límites en este caso expresa que el atributo *número_teléfono* del conjunto de entidades *cliente* puede tener entre cero y dos valores.

- Atributos **derivados**. El valor de este tipo de atributo se puede obtener a partir del valor de otros atributos o entidades relacionados. Por ejemplo, supóngase que el conjunto de entidades *cliente* que tiene un atributo *préstamos* que representa el número de préstamos que cada cliente tiene concedidos en el banco. Ese atributo se puede obtener contando el número de entidades *préstamo* asociadas con cada cliente.

  Como ejemplo adicional, supóngase que el conjunto de entidades *cliente* tiene el atributo *edad*, que indica la edad del cliente. Si el conjunto de entidades *cliente* tiene también un atributo *fecha_de_nacimiento*, se puede calcular *edad* a partir de *fecha_de_nacimiento* y de la fecha actual. Por tanto, *edad* es un atributo derivado. En este caso, *fecha_de_nacimiento* puede considerarse un atributo *básico*, o *almacenado*. El valor de los atributos derivados no se almacena, sino que se calcula cada vez que hace falta.

Los atributos toman valores **nulos** cuando las entidades no tienen ningún valor para ese atributo. El valor *nulo* también puede indicar "no aplicable"—es decir, que el valor no existe para esa entidad. Por ejemplo, una persona puede no tener un segundo nombre de pila. *Nulo* puede también designar que el valor del atributo es desconocido. Un valor desconocido puede ser *falta* (el valor existe pero no se tiene esa información) o *desconocido* (no se sabe si ese valor existe realmente o no).

Por ejemplo, si el valor *nombre* de un *cliente* dado es *nulo*, se da por supuesto que el valor es falta, ya que todos los clientes deben tener nombre. Un valor *nulo* para el atributo *piso* puede significar que la dirección no incluye un piso (no aplicable), que existe el valor piso pero no se conoce cuál es (falta), o que no se sabe si el valor piso forma parte o no de la dirección del cliente (desconocido).

6.3 Restricciones

Un esquema de desarrollo E-R puede definir ciertas restricciones a las que el contenido de la base de datos se debe adaptar. En este apartado se examinan la correspondencia de cardinalidades, las restricciones de claves y las restricciones de participación.

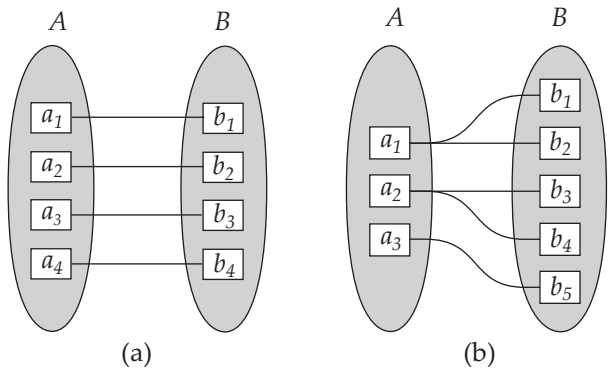


Figura 6.5 Correspondencia de cardinalidades. (a) Uno a uno. (b) Uno a varios.

6.3.1 Correspondencia de cardinalidades

La **correspondencia de cardinalidades**, o razón de cardinalidad, expresa el número de entidades a las que otra entidad se puede asociar mediante un conjunto de relaciones.

La correspondencia de cardinalidades resulta muy útil para describir conjuntos de relaciones binarias, aunque pueda contribuir a la descripción de conjuntos de relaciones que impliquen más de dos conjuntos de entidades. En este apartado se centrará la atención únicamente en los conjuntos de relaciones binarias.

Para un conjunto de relaciones binarias R entre los conjuntos de entidades A y B, la correspondencia de cardinalidades debe ser una de las siguientes:

- **Uno a uno** Cada entidad de A se asocia, *a lo sumo*, con una entidad de B, y cada entidad en B se asocia, *a lo sumo*, con una entidad de A (véase la Figura 6.5a).
- **Uno a varios** Cada entidad de A se asocia con cualquier número (cero o más) de entidades de B. Cada entidad de B, sin embargo, se puede asociar, *a lo sumo*, con una entidad de A (véase la Figura 6.5b).
- **Varios a uno** Cada entidad de A se asocia, *a lo sumo*, con una entidad de B. Cada entidad de B, sin embargo, se puede asociar con cualquier número (cero o más) de entidades de A (véase la Figura 6.6a).
- **Varios a varios** Cada entidad de A se asocia con cualquier número (cero o más) de entidades de B, y cada entidad de B se asocia con cualquier número (cero o más) de entidades de A (véase la Figura 6.6b).

La correspondencia de cardinalidades adecuada para un conjunto de relaciones dado depende, obviamente, de la situación del mundo real que el conjunto de relaciones modele.

Como ilustración, considérese el conjunto de relaciones *prestatario*. Si, en un banco dado, cada préstamo sólo puede pertenecer a un cliente y cada cliente puede tener varios préstamos, entonces el conjunto de relaciones de *cliente* a *préstamo* es uno a varios. Si cada préstamo puede pertenecer a varios clientes (como los préstamos solicitados conjuntamente por varios socios de un negocio) el conjunto de relaciones es varios a varios. La Figura 6.2 muestra este tipo de relación.

6.3.2 Claves

Es necesario tener una forma de especificar la manera de distinguir las entidades pertenecientes a un conjunto de entidades dado. Conceptualmente cada entidad es distinta; desde el punto de vista de las bases de datos, sin embargo, la diferencia entre ellas se debe expresar en términos de sus atributos.

Por lo tanto, los valores de los atributos de cada entidad deben ser tales que permitan *identificar unívocamente* a esa entidad. En otras palabras, no se permite que ningún par de entidades de un conjunto de entidades tenga exactamente el mismo valor en todos sus atributos.

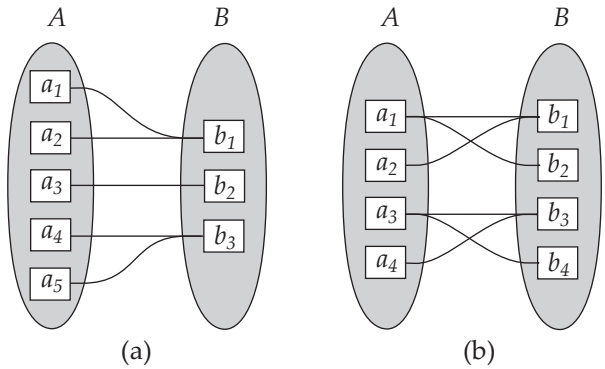


Figura 6.6 Correspondencia de cardinalidades. (a) Varios a uno. (b) Varios a varios.

Las *claves* permiten identificar un conjunto de atributos que resulta suficiente para distinguir las entidades entre sí. Las claves también ayudan a identificar unívocamente las relaciones y, por tanto, a distinguir las relaciones entre sí.

6.3.2.1 Conjuntos de entidades

Una **superclave** es un conjunto de uno o más atributos que, tomados conjuntamente, permiten identificar de forma unívoca una entidad del conjunto de entidades. Por ejemplo, el atributo *id_cliente* del conjunto de entidades *cliente* es suficiente para distinguir una entidad *cliente* de las demá. Por tanto, *id_cliente* es una superclave. Análogamente, la combinación de *nombre_cliente* e *id_cliente* es una superclave del conjunto de entidades *cliente*. El atributo *nombre_cliente* de *cliente* no es una superclave, ya que varias personas pueden tener el mismo nombre.

El concepto de superclave no es suficiente para lo que aquí se propone ya que, como se ha visto, las superclaves pueden contener atributos innecesarios. Si *C* es una superclave, entonces también lo es cualquier superconjunto de *C*. A menudo interesan las superclaves para las que ningún subconjunto propio es superclave. Esas superclaves mínimas se denominan **claves candidatas**.

Es posible que conjuntos distintos de atributos puedan servir como clave candidata. Supóngase que una combinación de *nombre_cliente* y *calle_cliente* sea suficiente para distinguir entre los miembros del conjunto de entidades *cliente*. Entonces, tanto *id_cliente* como *nombre_cliente*, *calle_cliente* son claves candidatas. Aunque los atributos *id_cliente* y *nombre_cliente* juntos puedan diferenciar las entidades *cliente*, su combinación no forma una clave candidata, ya que el atributo *id_cliente* por sí solo es una clave candidata.

Se usa el término **clave primaria** para denotar la clave candidata elegida por el diseñador de la base de datos como elemento principal de identificación de las entidades pertenecientes a un conjunto de entidades. Las claves (primarias, candidatas y superclaves) son propiedades del conjunto de entidades, más que de cada una de las entidades. Dos entidades cualesquiera del conjunto no pueden tener el mismo valor de los atributos de su clave al mismo tiempo. La designación de una clave representa una restricción de la empresa real que se está modelando.

Las claves candidatas se deben escoger con cuidado. Como se ha indicado, el nombre de cada persona es obviamente insuficiente, ya que puede haber muchas personas con el mismo nombre. En España, el atributo DNI de cada persona puede ser una clave candidata. Como los no ciudadanos de en España no tienen DNI, las empresas con trabajadores extranjeros deben generar sus propios identificadores únicos. Una alternativa es usar como clave alguna combinación única de otros atributos.

La clave primaria se debe escoger de manera que sus atributos nunca, o muy raramente, cambien. Por ejemplo, el campo dirección de cada persona no debe formar parte de la clave primaria, ya que es probable que cambie. El número de DNI, por otra parte, es seguro que no cambiará. Los identificadores únicos generados por las empresas generalmente no cambian, excepto si se fusionan dos empresas; en tal caso, las dos empresas pueden haber emitido el mismo identificador y será necesario reasignar los identificadores para asegurarse de que sean únicos.

6.3.2.2 Conjuntos de relaciones

La clave primaria de cada conjunto de entidades permite distinguir entre las diferentes entidades del conjunto. Se necesita un mecanismo parecido para distinguir entre las diferentes relaciones de cada conjunto de relaciones.

Sea R un conjunto de relaciones que involucra los conjuntos de entidades $E_1,\ E_2, \ldots, E_n$. Sea *clave_primaria* (E_i) el conjunto de atributos que forman la clave primaria del conjunto de entidades E_i. Por ahora se dará por supuesto que los nombres de los atributos de todas las claves primarias son únicos y que cada conjunto de entidades participa sólo una vez en la relación. La composición de la clave primaria de un conjunto de relaciones depende del conjunto de atributos asociado con el conjunto de relaciones R.

Si el conjunto de relaciones R no tiene atributos asociados, entonces el conjunto de atributos

$$\textit{clave_primaria}(E_1) \cup \textit{clave_primaria}(E_2) \cup \cdots \cup \textit{clave_primaria}(E_n)$$

describe una relación concreta del conjunto R.

Si el conjunto de relaciones R tiene asociados los atributos $a_1, a_2, \ldots, a_m$, entonces el conjunto de atributos

$$\textit{clave_primaria}(E_1) \cup \textit{clave_primaria}(E_2) \cup \cdots \cup \textit{clave_primaria}(E_n)\ \cup \{a_1, a_2, \ldots, a_m\}$$

describe una relación concreta del conjunto R.

En ambos casos, el conjunto de atributos

$$\textit{clave_primaria}(E_1) \cup \textit{clave_primaria}(E_2) \cup \cdots \cup \textit{clave_primaria}(E_n)$$

forma una superclave del conjunto de relaciones.

En el caso de que los nombres de los atributos de las claves primarias no sean únicos en todos los conjuntos de entidades, hay que renombrar los atributos para distinguirlos; el nombre del conjunto de entidades combinado con el del atributo formará un nombre único. En el caso de que un conjunto de entidades participe más de una vez en el conjunto de relaciones (como en la relación *trabaja_para* del Apartado 6.2.2), se usa el nombre del rol en lugar del nombre del conjunto de entidades para formar un nombre de atributo único.

La estructura de la clave primaria para el conjunto de relaciones depende de la correspondencia de cardinalidades del conjunto de relaciones. Como ejemplo, considérense los conjuntos de entidades *cliente* y *cuenta* y el conjunto de relaciones *impositor*, con el atributo *fecha_acceso* del Apartado 6.2.2. Supóngase que el conjunto de relaciones es varios a varios. Entonces, la clave primaria de *impositor* consiste en la unión de las claves primarias de *cliente* y de *cuenta*. Sin embargo, si un cliente sólo puede tener una cuenta—es decir, si la relación *impositor* es varios a uno de *cliente* a *cuenta*—entonces la clave primaria de *impositor* es simplemente la clave primaria de *cliente*. Análogamente, si la relación es varios a uno de *cuenta* a *cliente*—es decir, cada cuenta pertenece, a lo sumo, a un cliente— entonces la clave primaria de *impositor* es simplemente la clave primaria de *cuenta*. Para relaciones uno a uno se puede usar cualquiera de las claves primarias.

Para las relaciones no binarias, si no hay restricciones de cardinalidad, la superclave formada como se ha descrito anteriormente en este apartado es la única clave candidata, y se elige como clave primaria. La elección de la clave primaria resulta más complicada si hay restricciones de cardinalidad. Dado que no se ha estudiado la manera de especificar las restricciones de cardinalidad en relaciones no binarias, no se estudiará más este aspecto en este capítulo. Se considerará este aspecto con más detalle en el Apartado 7.4.

6.3.3 Restricciones de participación

Se dice que la participación de un conjunto de entidades E en un conjunto de relaciones R es **total** si cada entidad de E participa, al menos, en una relación de R. Si sólo algunas entidades de E participan en relaciones de R, se dice que la participación del conjunto de entidades E en la relación R es **parcial**. Por ejemplo, se puede esperar que cada entidad *préstamo* esté relacionada, al menos, con un cliente mediante

la relación *prestatario*. Por tanto, la participación de *préstamo* en el conjunto de relaciones *prestatario* es total. En cambio, un individuo puede ser cliente del banco tenga o no tenga concedido algún préstamo en el banco. Por tanto, es posible que sólo algunas de las entidades *cliente* estén relacionadas con el conjunto de entidades *préstamo* mediante la relación *prestatario*, y la participación de *cliente* en la relación *prestatario* es, por tanto, parcial.

6.4 Diagramas entidad-relación

Como se vio brevemente en el Apartado 1.3.3, los **diagramas E-R** pueden expresar gráficamente la estructura lógica general de las bases de datos. Los diagramas E-R son sencillos y claros—cualidades que pueden ser responsables en gran parte de la popularidad del modelo E-R. Estos diagramas constan de los siguientes componentes principales:

- **Rectángulos**, que representan conjuntos de entidades.
- **Elipses**, que representan atributos.
- **Rombos**, que representan conjuntos de relaciones.
- **Líneas**, que unen los atributos con los conjuntos de entidades y los conjuntos de entidades con los conjuntos de relaciones.
- **Elipses dobles**, que representan atributos multivalorados.
- **Elipses discontinuas**, que denotan atributos derivados.
- **Líneas dobles**, que indican participación total de una entidad en un conjunto de relaciones.
- **Rectángulos dobles**, que representan conjuntos de entidades débiles (se describirán posteriormente en el Apartado 6.6).

Considérese el diagrama entidad-relación de la Figura 6.7, que consiste en dos conjuntos de entidades, *cliente* y *préstamo*, relacionados mediante el conjunto de relaciones binarias *prestatario*. Los atributos asociados con *cliente* son *id_cliente*, *nombre_cliente*, *calle_cliente*, y *ciudad_cliente*. Los atributos asociados con *préstamo* son *número_préstamo* e *importe*. En la Figura 6.7 los atributos del conjunto de entidades que son miembros de la clave primaria están subrayados.

El conjunto de relaciones *prestatario* puede ser varios a varios, uno a varios, varios a uno o uno a uno. Para distinguir entre estos tipos, se dibuja o una línea dirigida (→) o una línea no dirigida (—) entre el conjunto de relaciones y el conjunto de entidades en cuestión.

- Una línea dirigida desde el conjunto de relaciones *prestatario* al conjunto de entidades *préstamo* especifica que *prestatario* es un conjunto de relaciones uno a uno o varios a uno desde *cliente* a *préstamo*; *prestatario* no puede ser un conjunto de relaciones varios a varios ni uno a varios desde *cliente* a *préstamo*.

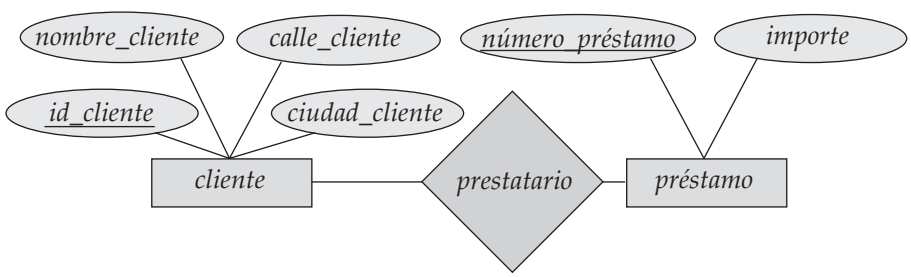


Figura 6.7 Diagrama E-R correspondiente a clientes y préstamos.

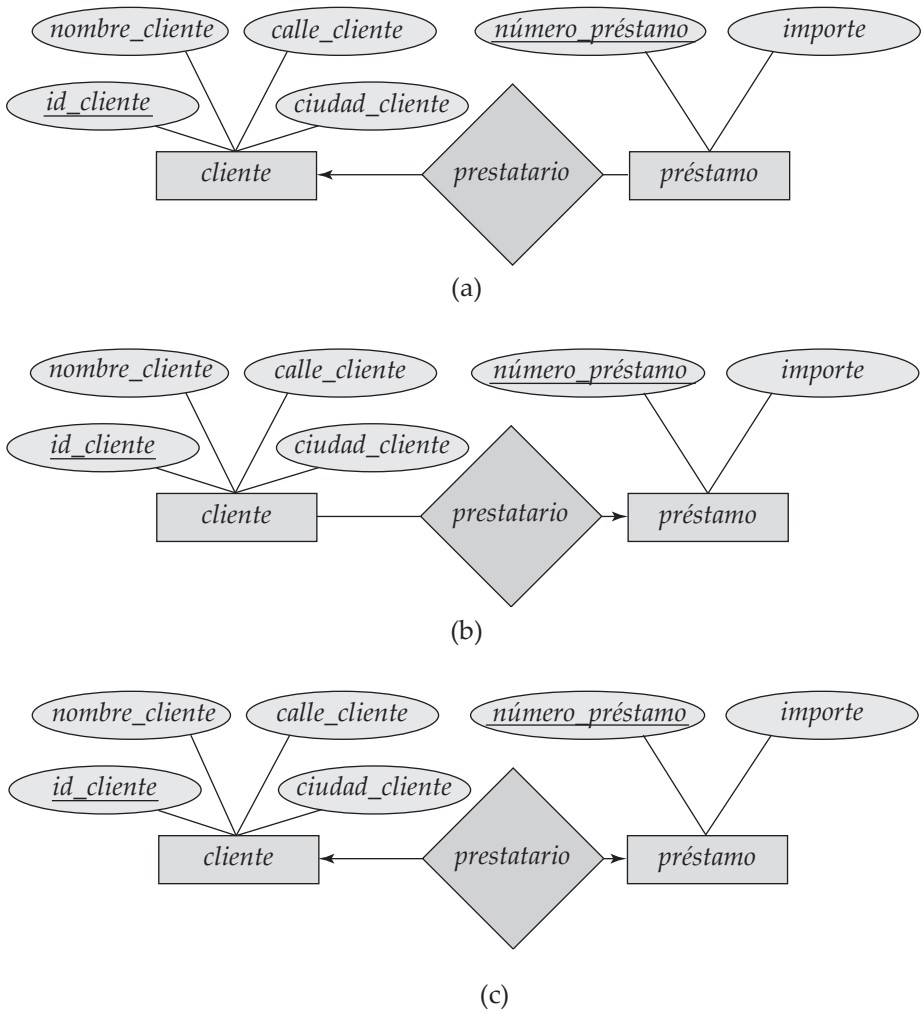


Figura 6.8 Relaciones. (a) Uno a varios. (b) Varios a uno. (c) Uno a uno.

- Una línea no dirigida desde el conjunto de relaciones *prestatario* al conjunto de relaciones *préstamo* especifica que *prestatario* es un conjunto de relaciones varios a varios o uno a varios desde *cliente* a *préstamo*.

Volviendo al diagrama E-R de la Figura 6.7, puede verse que el conjunto de relaciones *prestatario* es varios a varios. Si el conjunto de relaciones *prestatario* fuera uno a varios, desde *cliente* a *préstamo*, entonces la línea desde *prestatario* a *cliente* sería dirigida, con una flecha que apuntaría al conjunto de

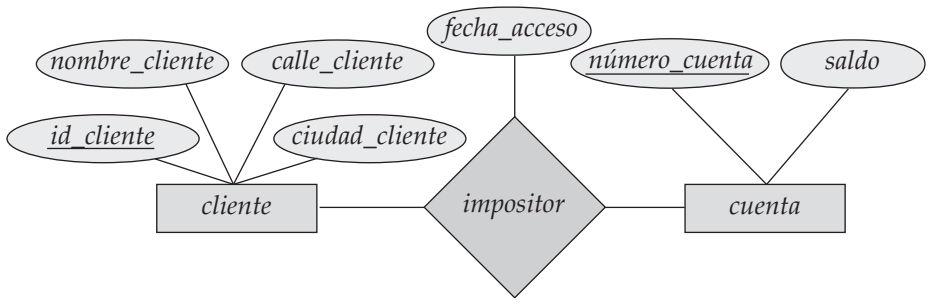


Figura 6.9 Diagrama E-R con un atributo unido a un conjunto de relaciones.

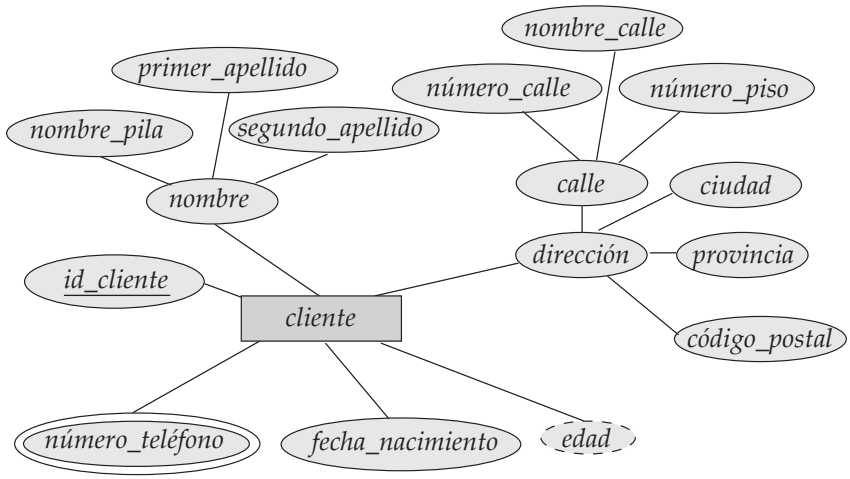


Figura 6.10 Diagrama E-R con atributos compuestos, multivalorados y derivados.

entidades *cliente* (Figura 6.8a). Análogamente, si el conjunto de relaciones *prestatario* fuera varios a uno desde *cliente* a *préstamo*, entonces la línea desde *prestatario* a *préstamo* tendría una flecha que apuntaría al conjunto de entidades *préstamo* (Figura 6.8b). Finalmente, si el conjunto de relaciones *prestatario* fuera uno a uno, entonces las dos líneas que salen de *prestatario* tendrían flecha: una que apuntaría al conjunto de entidades *préstamo* y otra que apuntaría al conjunto de entidades *cliente* (Figura 6.8c).

Si un conjunto de relaciones también tiene asociados algunos atributos, entonces esos atributos se unen con el conjunto de relaciones. Por ejemplo, en la Figura 6.9, se tiene el atributo descriptivo *fecha_acceso* unido al conjunto de relaciones *impositor* para especificar la fecha del último acceso del cliente a esa cuenta.

La Figura 6.10 muestra cómo se pueden representar los atributos compuestos en la notación E-R. En este caso, el atributo compuesto *nombre*, con los atributos componentes *nombre_pila*, *primer_apellido* y *segundo_apellido* sustituye al atributo simple *nombre_cliente* de *cliente*. Además, el atributo compuesto *dirección*, cuyos atributos componentes son *calle*, *ciudad*, *provincia* y *código_postal*, sustituye a los atributos *calle_cliente* y *ciudad_cliente* de *cliente*. El atributo *calle* es por sí mismo un atributo compuesto cuyos atributos componentes son *número_calle*, *nombre_calle* y *número_piso*.

La Figura 6.10 también muestra un atributo multivalorado, *número_teléfono*, indicado por una elipse doble, y un atributo derivado, *edad*, indicado por una elipse discontinua.

En los diagramas E-R los roles se indican mediante etiquetas en las líneas que unen los rombos con los rectángulos. La Figura 6.11 muestra los indicadores de roles *director* y *trabajador* entre el conjunto de entidades *empleado* y el conjunto de relaciones *trabaja_para*.

Los conjuntos de relaciones no binarias se pueden especificar fácilmente en los diagramas E-R. La Figura 6.12 consta de tres conjuntos de entidades *empleado*, *trabajo* y *sucursal*, relacionados mediante el conjunto de relaciones *trabaja_en*.

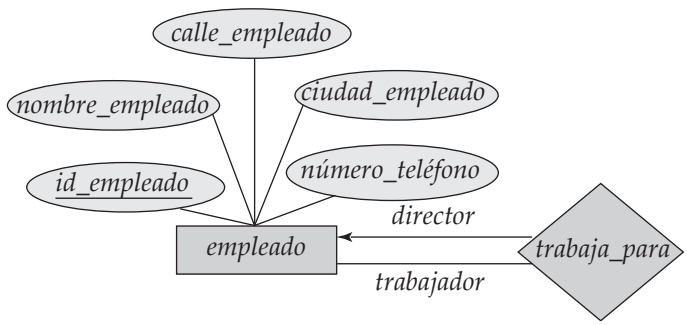


Figura 6.11 Diagrama E-R con indicadores de roles.

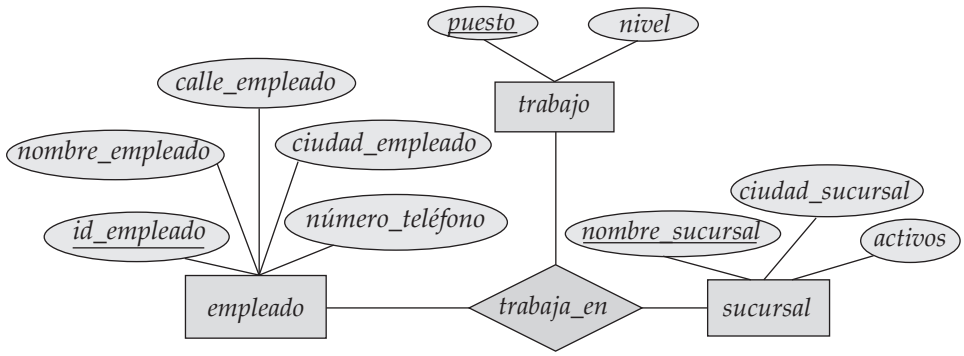


Figura 6.12 Diagrama E-R con una relación ternaria.

En el caso de conjuntos de relaciones no binarias se pueden especificar algunos tipos de relaciones varios a uno. Supóngase que un empleado puede tener, a lo sumo, un trabajo en cada sucursal (por ejemplo, Santos no puede ser director e interventor en la misma sucursal). Esta restricción se puede especificar mediante una flecha que apunte a *trabajo* en el arco de *trabaja_en*.

Como máximo se permite una flecha desde cada conjunto de relaciones, ya que los diagramas E-R con dos o más flechas salientes de cada conjunto de relaciones no binarias se pueden interpretar de dos formas. Supónganse que hay un conjunto de relaciones *R* entre los conjuntos de entidades $A_1, A_2, \ldots, A_n$ y que las únicas flechas están en los arcos de los conjuntos de entidades $A_{i+1}, A_{i+2}, \ldots, A_n$. Entonces, las dos interpretaciones posibles son:

1. Una combinación concreta de entidades de $A_1, A_2, \ldots, A_i$ se puede asociar, a lo sumo, con una combinación de entidades de $A_{i+1}, A_{i+2}, \ldots, A_n$. Por tanto, la clave primaria de la relación *R* se puede crear mediante la unión de las claves primarias de $A_1, A_2, \ldots, A_i$.
2. Para cada conjunto de entidades A_k, $i < k \leq n$, cada combinación de las entidades de los otros conjuntos de entidades se puede asociar, a lo sumo, con una entidad de A_k. Cada conjunto $\{A_1, A_2, \ldots, A_{k-1}, A_{k+1}, \ldots, A_n\}$, para $i < k \leq n$, forma, entonces, una clave candidata.

Cada una de estas interpretaciones se ha usado en diferentes libros y sistemas. Para evitar confusiones, sólo se permite una flecha saliente de cada conjunto de relaciones, en cuyo caso las dos interpretaciones son equivalentes. En el Capítulo 7 (Apartado 7.4) se estudia el concepto de *dependencia funcional*, que permite especificar cualquiera de estas dos interpretaciones sin ambigüedad.

En los diagramas E-R se usan líneas dobles para indicar que la participación de un conjunto de entidades en un conjunto de relaciones es total; es decir, cada entidad del conjunto de entidades aparece, al menos, en una relación de ese conjunto de relaciones. Por ejemplo, considérese la relación *prestatario* entre clientes y préstamos. Una línea doble de *préstamo* a *prestatario*, como en la Figura 6.13, indica que cada préstamo debe tener, al menos, un cliente asociado.

Los diagramas E-R también ofrecen una manera de indicar restricciones más complejas sobre el número de veces que cada entidad participa en las relaciones de un conjunto de relaciones. Un segmento entre un conjunto de entidades y un conjunto de relaciones binarias puede tener unas cardinalidades

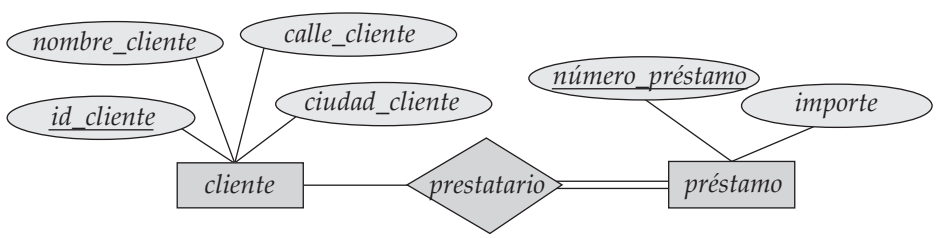


Figura 6.13 Participación total de un conjunto de entidades en un conjunto de relaciones.

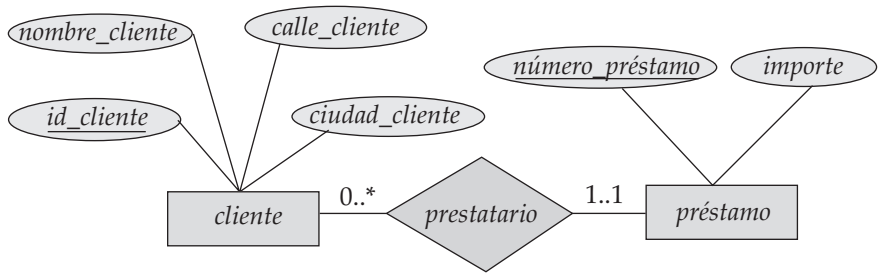


Figura 6.14 Límites de cardinalidad en los conjuntos de relaciones.

mínima y máxima asociadas , que se muestran de la forma $min..max$, donde min es la cardinalidad mínima y max es la máxima. Un valor mínimo de 1 indica una participación total del conjunto de entidades en el conjunto de relaciones. Un valor máximo de 1 indica que la entidad participa, a lo sumo, en una relación, mientras que un valor máximo de $*$ indica que no hay límite. Téngase en cuenta que una etiqueta $1..*$ en un segmento es equivalente a una línea doble.

Por ejemplo, considérese la Figura 6.14. El segmento entre *préstamo* y *prestatario* tiene una restricción de cardinalidad de $1..1$, que significa que tanto la cardinalidad mínima como la máxima son 1. Es decir, cada préstamo debe tener exactamente un cliente asociado. El límite $0..*$ en el segmento de *cliente* a *prestatario* indica que cada cliente puede no tener ningún préstamo o tener varios. Por tanto, la relación *prestatario* es uno a varios de *cliente* a *préstamo* y, además, la participación de *préstamo* en *prestatario* es total.

Es fácil malinterpretar $0..*$ en el segmento entre *cliente* y *prestatario* y pensar que la relación *prestatario* es varios a uno de *cliente* a *préstamo*—esto es exactamente lo contrario a la interpretación correcta.

Si los dos segmentos de una relación binaria tienen un valor máximo de 1, la relación es uno a uno. Si se hubiese especificado un límite de cardinalidad de $1..*$ en el segmento entre *cliente* y *prestatario*, se estaría diciendo que cada cliente debe tener, al menos, un préstamo.

6.5 Aspectos del diseño entidad-relación

Los conceptos de conjunto de entidades y de conjunto de relaciones no son precisos, y es posible definir el conjunto de entidades y las relaciones entre ellas de diferentes formas. En este apartado se examinan aspectos básicos del diseño de esquemas de bases de datos E-R. El Apartado 6.7.4 trata el proceso de diseño con más detalle.

6.5.1 Uso de conjuntos de entidades y de atributos

Considérese el conjunto de entidades *empleado* con los atributos *nombre_empleado* y *número_teléfono*. Se puede argumentar que un *teléfono* es una entidad en sí misma con los atributos *número_teléfono* y *ubicación*; la ubicación puede ser la sucursal o el domicilio donde el teléfono está instalado, y se pueden representar los teléfonos móviles (celulares) mediante el valor "móvil". Si se adopta este punto de vista, hay que redefinir el conjunto de entidades *empleado* como:

- El conjunto de entidades *empleado* con los atributos *id_empleado* y *nombre_empleado*.
- El conjunto de entidades *teléfono* con los atributos *número_teléfono* y *ubicación*.
- La relación *teléfono_empleado*, que denota la asociación entre los empleados y sus teléfonos.

Estas alternativas se muestran en la Figura 6.15.

¿Cuál es, entonces, la diferencia principal entre estas dos definiciones de empleado? Tratar el teléfono como el atributo *número_teléfono* implica que cada empleado tiene exactamente un número de teléfono. Tratar el teléfono como la entidad *teléfono* permite que los empleados tengan varios números de telé-

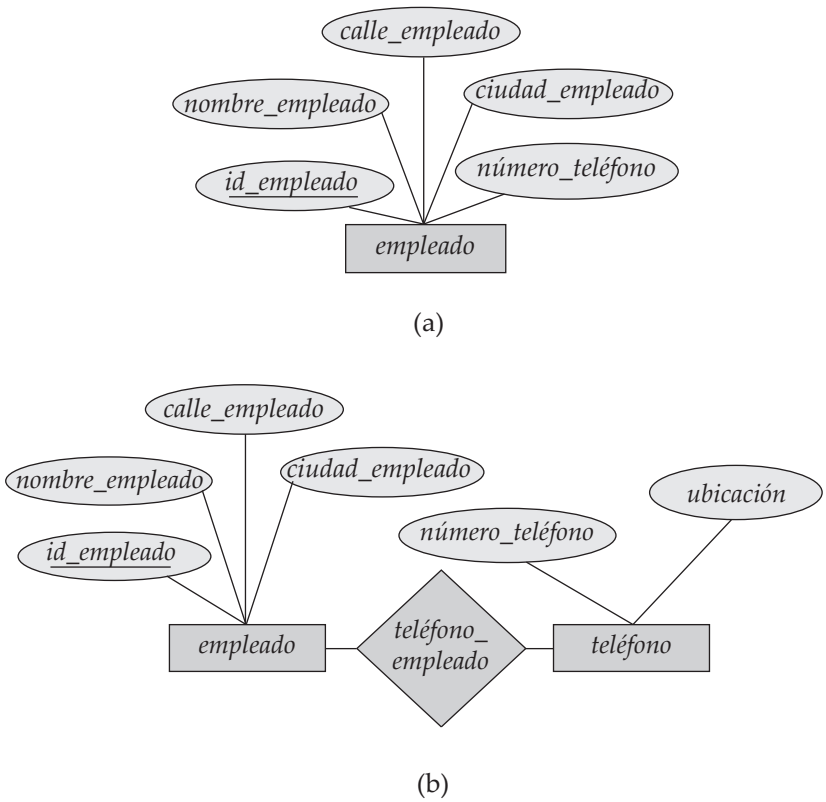


Figura 6.15 Alternativas para *empleado* y *teléfono*.

fono (o ninguno) asociados. Sin embargo, se puede definir en su lugar *número_teléfono* como atributo multivalorado para permitir varios teléfonos por empleado.

La diferencia principal es que tratar el teléfono como entidad modela mejor una situación en la que se puede querer guardar información extra sobre el teléfono, como su ubicación, su tipo (móvil, videoteléfono o fijo) o las personas que lo comparten. Por tanto, tratar el teléfono como entidad es más general que tratarlo como atributo y resulta adecuado cuando la generalidad pueda ser útil.

En cambio, no sería apropiado tratar el atributo *nombre_empleado* como entidad; es difícil argumentar que *nombre_empleado* sea una entidad en sí misma (a diferencia de lo que ocurre con *teléfono*). Así pues, resulta adecuado tener *nombre_empleado* como atributo del conjunto de entidades *empleado*.

Por tanto, se suscitan naturalmente dos preguntas: ¿qué constituye un atributo? y ¿qué constituye un conjunto de entidades? Desafortunadamente no hay respuestas sencillas. Las distinciones dependen principalmente de la estructura de la empresa real que se esté modelando y de la semántica asociada con el atributo en cuestión.

Un error común es usar la clave primaria de un conjunto de entidades como atributo de otro conjunto de entidades en lugar de usar una relación. Por ejemplo, es incorrecto modelar *id_cliente* como atributo de *préstamo*, aunque cada préstamo tenga sólo un cliente. La relación *prestatario* es la forma correcta de representar la conexión entre los préstamos y los clientes, ya que hace explícita su conexión, en lugar de dejarla implícita mediante un atributo.

Otro error relacionado con éste que se comete a veces es escoger los atributos de clave primaria de los conjuntos de entidades relacionados como atributos del conjunto de relaciones. Por ejemplo, *número_préstamo* (el atributo de clave primaria de *préstamo*) e *id_cliente* (la clave primaria de *cliente*) no deben aparecer como atributos de la relación *prestatario*. Esto no es adecuado, ya que los atributos de clave primaria están implícitos en el conjunto de relaciones[3].

3. Cuando se crea un esquema de relación a partir del esquema E-R los atributos pueden aparecer en una tabla creada a partir del conjunto de relaciones *prestatario*, como se verá más adelante; no obstante, no deben aparecer en el conjunto de relaciones *prestatario*.

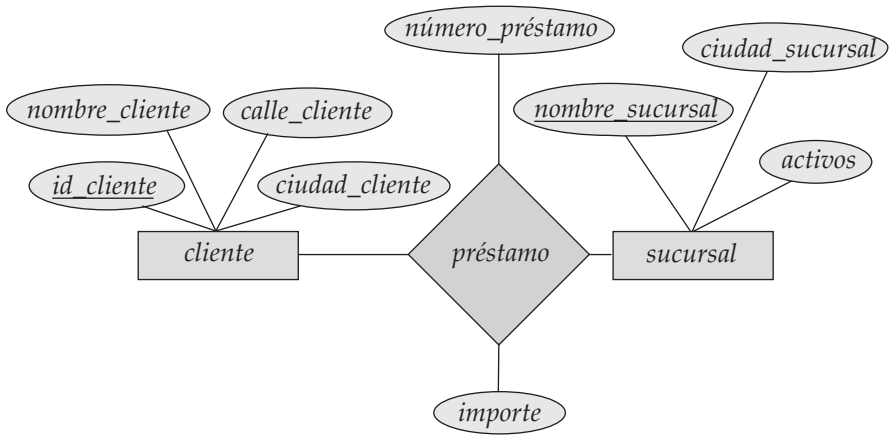


Figura 6.16 *préstamo* como conjunto de relaciones.

6.5.2 Uso de los conjuntos de entidades y de los conjuntos de relaciones

No siempre está claro si es mejor expresar un objeto mediante un conjunto de entidades o mediante un conjunto de relaciones. En el Apartado 6.2.1 se dio por supuesto que los préstamos bancarios se modelaban como entidades. Una alternativa es modelar los préstamos no como entidades, sino como relaciones entre los clientes y las sucursales, con *número_préstamo* e *importe* como atributos descriptivos, como puede verse en la Figura 6.16. Cada préstamo se representa mediante una relación entre un cliente y una sucursal.

Si cada préstamo se concede exactamente a un cliente y se asocia exactamente con una sucursal, se puede encontrar satisfactorio el diseño en el que cada préstamo se representa como una relación. Sin embargo, con este diseño, no se puede representar convenientemente una situación en la que un préstamo se conceda a varios clientes conjuntamente. Para tratar esta situación se debe definir otra relación para cada prestatario de ese préstamo conjunto. Entonces habrá que replicar los valores de los atributos descriptivos *número_préstamo* e *importe* en cada una de esas relaciones. Cada una de esas relaciones debe, por supuesto, tener el mismo valor para los atributos descriptivos *número_préstamo* e *importe*.

Se plantean dos problemas como resultado de esta réplica: (1) los datos se almacenan varias veces, lo que hace que se desperdicie espacio de almacenamiento, y (2) las actualizaciones pueden dejar los datos en un estado inconsistente, en el que los valores de atributos que se supone que tienen el mismo valor difieran. El problema de cómo evitar esta réplica se trata formalmente mediante la *teoría de la normalización*, que se aborda en el Capítulo 7.

El problema de la réplica de los atributos *número_préstamo* e *importe* no aparece en el diseño original del Apartado 6.4, ya que allí *préstamo* es un conjunto de entidades.

Un criterio para determinar si se debe usar un conjunto de entidades o un conjunto de relaciones puede ser escoger un conjunto de relaciones para describir las acciones que ocurran entre entidades. Este enfoque también puede ser útil para decidir si ciertos atributos se pueden expresar mejor como relaciones.

6.5.3 Conjuntos de relaciones binarias y n-arias

Las relaciones en las bases de datos suelen ser binarias. Puede que algunas relaciones que no parecen ser binarias se puedan representar mejor mediante varias relaciones binarias. Por ejemplo, se puede crear la relación ternaria *padres*, que relaciona a cada hijo con su padre y con su madre. Sin embargo, esa relación se puede representar mediante dos relaciones binarias, *padre* y *madre*, que relacionan a cada hijo con su padre y con su madre por separado. El uso de las dos relaciones *padre* y *madre* permite el registro de la madre del niño aunque no se conozca la identidad del padre; si se usara en la relación ternaria *padres* se necesitaría un valor nulo. En este caso es preferible usar conjuntos de relaciones binarias.

De hecho, siempre es posible sustituir los conjuntos de relaciones no binarias ($n$aria, para $n > 2$) por varios conjuntos de relaciones binarias. Por simplificar, considérense el conjunto de relaciones abstracto ternario ($n = 3$) R y los conjuntos de entidades A, B, y C. Se sustituye el conjunto de relaciones R por un conjunto de entidades E y se crean tres conjuntos de relaciones como se muestra en la Figura 6.17:

- P_A, que relaciona E y A
- P_B, que relaciona E y B
- P_C, que relaciona E y C

Si el conjunto de relaciones R tiene atributos, éstos se asignan al conjunto de entidades E; además, se crea un atributo de identificación especial para E (ya que se debe poder distinguir las diferentes entidades de cada conjunto de entidades con base en los valores de sus atributos). Para cada relación (a_i, b_i, c_i) del conjunto de relaciones R se crea una nueva entidad e_i del conjunto de entidades E. Luego, en cada uno de los tres nuevos conjuntos de relaciones, se inserta una relación del modo siguiente:

- (e_i, a_i) en R_A
- (e_i, b_i) en R_B
- (e_i, c_i) en R_C

Este proceso se puede generalizar de forma directa a los conjuntos de relaciones n-arias. Por tanto, conceptualmente, se puede restringir el modelo E-R para que sólo incluya conjuntos de relaciones binarias. Sin embargo, esta restricción no siempre es deseable.

- Es posible que sea necesario crear un atributo de identificación para que el conjunto de entidades represente el conjunto de relaciones. Este atributo, junto con los conjuntos de relaciones adicionales necesarios, incrementa la complejidad del diseño y (como se verá en el Apartado 6.9) los requisitos globales de almacenamiento.
- Un conjunto de relaciones n-arias muestra más claramente que varias entidades participan en una sola relación.
- Puede que no haya forma de traducir las restricciones a la relación ternaria en restricciones a las relaciones binarias. Por ejemplo, considérese una restricción que dice que R es varios a uno de A, B a C; es decir, cada par de entidades de A y de B se asocia, a lo sumo, con una entidad de C. Esta restricción no se puede expresar mediante restricciones de cardinalidad sobre los conjuntos de relaciones R_A, R_B y R_C.

Considérese el conjunto de relaciones *trabaja_en* del Apartado 6.2.2, que relaciona *empleado*, *sucursal* y *trabajo*. No se puede dividir directamente *trabaja_en* en relaciones binarias entre *empleado* y *sucursal* y entre *empleado* y *trabajo*. Si se hiciese, se podría registrar que Santos es director e interventor y que Santos trabaja en Navacerrada y en Centro; sin embargo, no se podría registrar que Santos es director de Navacerrada e interventor de Centro, pero que no es interventor de Navacerrada ni director de Centro.

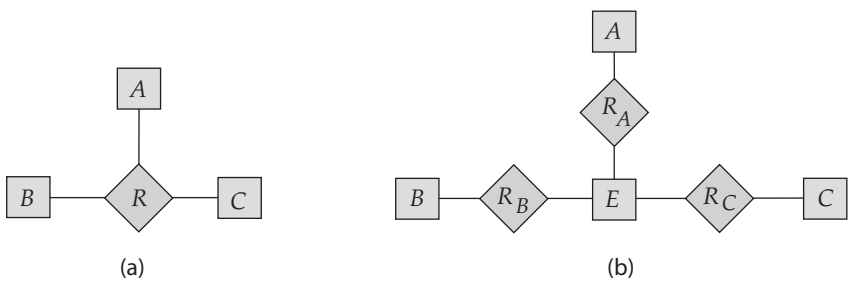


Figura 6.17 Una relación ternaria y tres relaciones binarias.

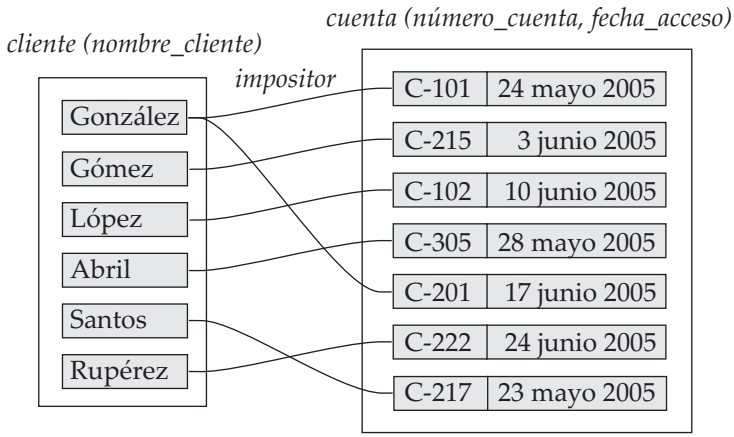


Figura 6.18 *fecha_acceso* como atributo del conjunto de entidades *cuenta*.

El conjunto de relaciones *trabaja_en* se puede dividir en relaciones binarias mediante la creación de nuevos conjuntos de entidades como se ha descrito anteriormente. Sin embargo, no sería muy natural.

6.5.4 Ubicación de los atributos de las relaciones

La razón de cardinalidad de una relación puede afectar a la ubicación de sus atributos. Por tanto, los atributos de los conjuntos de relaciones uno a uno o uno a varios pueden estar asociados con uno de los conjuntos de entidades participantes, en lugar de con el conjunto de relaciones. Por ejemplo, especifiquemos que *impositor* es un conjunto de relaciones uno a varios tal que cada cliente puede tener varias cuentas, pero cada cuenta sólo puede tener un cliente como titular. En este caso, el atributo *fecha_acceso*, que especifica la fecha en que el cliente tuvo acceso a la cuenta por última vez, podría estar asociado con el conjunto de entidades *cuenta*, como se describe en la Figura 6.18; para simplificar la figura sólo se muestran algunos de los atributos de los dos conjuntos de entidades. Dado que cada entidad *cuenta* participa en una relación con un ejemplar de *cliente*, como máximo, hacer esta designación de atributos tendría el mismo significado que colocar *fecha_acceso* en el conjunto de relaciones *impositor*. Los atributos de un conjunto de relaciones uno a varios sólo se pueden recolocar en el conjunto de entidades de la parte "varios" de la relación. Por otra parte, para los conjuntos de entidades uno a uno, los atributos de la relación se pueden asociar con cualquiera de las entidades participantes.

La decisión de diseño sobre la ubicación de los atributos descriptivos en estos casos—como atributo de la relación o de la entidad—debe reflejar las características de la empresa que se modela. El diseñador puede elegir mantener *fecha_acceso* como atributo de *impositor* para expresar explícitamente que se produce un acceso en el punto de interacción entre los conjuntos de entidades *cliente* y *cuenta*.

La elección de la ubicación del atributo es más sencilla para los conjuntos de relaciones varios a varios. Volviendo al ejemplo, especifiquemos el caso, quizá más realista, de que *impositor* sea un conjunto de relaciones varios a varios que expresa que cada cliente puede tener una o más cuentas, y que cada cuenta puede tener a varios clientes como titulares. Si hay que expresar la fecha en que un cliente dado tuvo acceso por última vez a una cuenta concreta, *fecha_acceso* debe ser atributo del conjunto de relaciones *impositor*, en lugar de serlo de cualquiera de las entidades participantes. Si *fecha_acceso* fuese un atributo de *cuenta*, por ejemplo, no se podría determinar qué cliente llevó a cabo el acceso más reciente a una cuenta conjunta. Cuando un atributo se determina mediante la combinación de los conjuntos de entidades participantes, en lugar de por cada entidad por separado, ese atributo debe estar asociado con el conjunto de relaciones varios a varios. La Figura 6.3 muestra la ubicación de *fecha_acceso* como atributo de la relación; de nuevo, para simplificar la figura, sólo se muestran algunos de los atributos de los dos conjuntos de entidades.

6.6 Conjuntos de entidades débiles

Puede que un conjunto de entidades no tenga suficientes atributos para formar una clave primaria. Ese conjunto de entidades se denomina **conjunto de entidades débiles**. Los conjuntos de entidades que tienen una clave primaria se denominan **conjuntos de entidades fuertes**.

Como ilustración, considérese el conjunto de entidades *pago*, que tiene tres atributos: *número_pago*, *fecha_pago* e *importe_pago*. Los números de pago suelen ser números secuenciales, a partir de 1, generados independientemente para cada préstamo. Por tanto, aunque cada entidad *pago* es distinta, los pagos de diferentes préstamos pueden compartir el mismo número de pago. Así, este conjunto de entidades no tiene clave primaria; es un conjunto de entidades débiles.

Para que un conjunto de entidades débiles tenga sentido, debe estar asociado con otro conjunto de entidades, denominado **conjunto de entidades identificadoras** o **propietarias**. Cada entidad débil debe estar asociada con una entidad identificadora; es decir, se dice que el conjunto de entidades débiles **depende existencialmente** del conjunto de entidades identificadoras. Se dice que el conjunto de entidades identificadoras **posee** el conjunto de entidades débiles al que identifica. La relación que asocia el conjunto de entidades débiles con el conjunto de entidades identificadoras se denomina **relación identificadora**. La relación identificadora es varios a uno del conjunto de entidades débiles al conjunto de entidades identificadoras y la participación del conjunto de entidades débiles en la relación es total.

En nuestro ejemplo, el conjunto de entidades identificador de *pago* es *préstamo*, y la relación *pago_préstamo* que asocia las entidades *pago* con sus correspondientes entidades *préstamo* es la relación identificadora.

Aunque los conjuntos de entidades débiles no tienen clave primaria, hace falta un medio para distinguir entre todas las entidades del conjunto de entidades débiles que dependen de una entidad fuerte concreta. El **discriminante** de un conjunto de entidades débiles es un conjunto de atributos que permite que se haga esta distinción. Por ejemplo, el discriminante del conjunto de entidades débiles *pago* es el atributo *número_pago* ya que, para cada préstamo, el número de pago identifica de forma única cada pago de ese préstamo. El discriminante del conjunto de entidades débiles se denomina *clave parcial* del conjunto de entidades.

La clave primaria de un conjunto de entidades débiles se forma con la clave primaria del conjunto de entidades identificadoras y el discriminante del conjunto de entidades débiles. En el caso del conjunto de entidades *pago*, su clave primaria es {*número_préstamo*, *número_pago*}, donde *número_préstamo* es la clave primaria del conjunto de entidades identificadoras, es decir, *préstamo*, y *número_pago* distingue las entidades *pago* de un mismo préstamo.

El conjunto de entidades identificadoras no deber tener atributos descriptivos, ya que cualquier atributo requerido puede estar asociado con el conjunto de entidades débiles (véase la discusión sobre el traslado de los atributos del conjunto de relaciones a los conjuntos de entidades participantes en el Apartado 6.5.4).

Los conjuntos de entidades débiles pueden participar en otras relaciones aparte de la relación identificadora. Por ejemplo, la entidad *pago* puede participar en una relación con el conjunto de entidades *cuenta*, identificando la cuenta desde la que se ha realizado el pago. Los conjuntos de entidades débiles pueden participar como propietario de una relación identificadora con otro conjunto de entidades débiles. También es posible tener conjuntos de entidades débiles con más de un conjunto de entidades identificadoras. Cada entidad débil se identificaría mediante una combinación de entidades, una de cada conjunto de entidades identificadoras. La clave primaria de la entidad débil consistiría de la unión de las claves primarias de los conjuntos de entidades identificadoras y el discriminante del conjunto de entidades débiles.

En los diagramas E-R los rectángulos con líneas dobles indican conjuntos de entidades débiles, mientras que un rombo con líneas dobles indica la correspondiente relación de identificación. En la Figura 6.19, el conjunto de entidades débiles *pago* depende del conjunto de entidades fuertes *préstamo* mediante el conjunto de relaciones *pago_préstamo*.

La figura también ilustra el uso de líneas dobles para indicar *participación total*—la participación del conjunto de entidades (débiles) *pago* en la relación *pago_préstamo* es total, lo que significa que cada pago debe estar relacionando mediante *pago_préstamo* con alguna cuenta. Finalmente, la flecha de *pago_prés-*

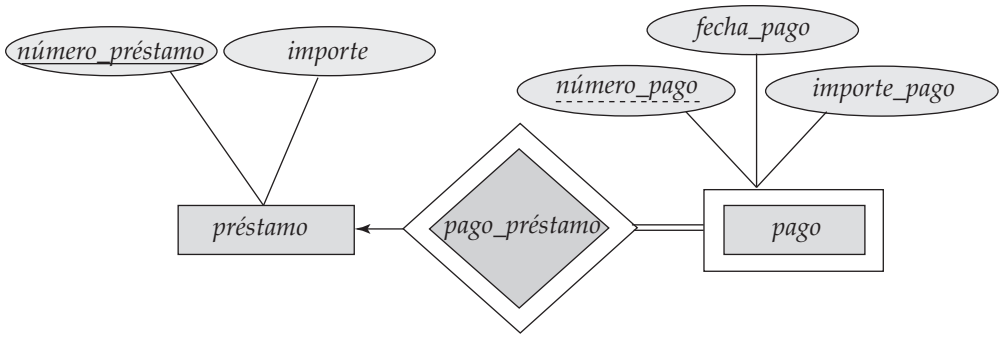


Figura 6.19 Diagrama E-R con un conjunto de entidades débiles.

tamo a *préstamo* indica que cada pago es para un solo préstamo. Los discriminantes de los conjuntos de entidades débiles también se subrayan, pero con una línea discontinua, en lugar de con una continua.

En algunos casos, puede que el diseñador de la base de datos decida expresar un conjunto de entidades débiles como atributo compuesto multivalorado del conjunto de entidades propietarias. En el ejemplo, esta alternativa exigiría que el conjunto de entidades *préstamo* tuviera el atributo compuesto y multivalorado *pago*, que constara de *número_pago*, *fecha_pago* e *importe_pago*. Los conjuntos de entidades débiles se pueden modelar mejor como atributos si sólo participan en la relación identificadora y tienen pocos atributos. A la inversa, las representaciones de los conjuntos de entidades débiles modelarán mejor las situaciones en las que esos conjuntos participen en otras relaciones aparte de la relación identificadora y tengan muchos atributos.

Como ejemplo adicional de conjunto de entidades que se puede modelar como conjunto de entidades débiles, considérese las ofertas de asignaturas en una universidad. La misma asignatura se puede ofrecer en diferentes cursos y, dentro de cada curso, puede haber varios grupos para la misma asignatura. Por tanto, se puede crear el conjunto de entidades débiles *oferta_asignatura*, que depende existencialmente de *asignatura*; las diferentes ofertas de la misma asignatura se identifican mediante *curso* y *número_grupo*, que forman un discriminante pero no una clave primaria.

6.7 Características del modelo E-R extendido

Aunque los conceptos básicos del modelo E-R pueden modelar la mayor parte de las características de las bases de datos, algunos aspectos de las bases de datos se pueden expresar mejor mediante ciertas extensiones del modelo E-R básico. En este apartado se estudian las características E-R extendidas de especialización, generalización, conjuntos de entidades de nivel superior e inferior, herencia de atributos y agregación.

6.7.1 Especializacion

Los conjuntos de entidades pueden incluir subgrupos de entidades que se diferencian de alguna forma de las demás entidades del conjunto. Por ejemplo, un subconjunto de entidades de un conjunto de entidades puede tener atributos que no sean compartidos por todas las entidades del conjunto de entidades. El modelo E-R ofrece un medio de representar estos grupos de entidades diferentes.

Como ejemplo, considérese el conjunto de entidades *persona* con los atributos *id_persona, nombre*, *calle* y *ciudad*. Cada persona puede clasificarse además en una de las categorías siguientes:

- *cliente*
- *empleado*

Cada uno de estos tipos de persona se describen mediante un conjunto de atributos que incluye todos los atributos del conjunto de entidades *persona* más otros posibles atributos adicionales. Por ejemplo, las entidades *cliente* se pueden describir además mediante el atributo *calificación_crediticia*, mientras que

las entidades *empleado* se pueden describir además mediante el atributo *sueldo*. El proceso de establecimiento de subgrupos dentro del conjunto de entidades se denomina **especialización**. La especialización de *persona* permite distinguir entre las personas basándonos en si son empleados o clientes: en general, cada persona puede ser empleado, cliente, las dos cosas o ninguna de ellas.

Como ejemplo adicional, supóngase que el banco desea dividir las cuentas en dos categorías: cuentas corrientes y cuentas de ahorro. Las cuentas de ahorro necesitan un saldo mínimo, pero el banco puede establecer diferentes tasas de interés para cada cliente y ofrecer mejores tasas a los clientes preferentes. Las cuentas corrientes tienen una tasa de interés fija, pero permiten los descubiertos; hay que registrar el importe de los descubiertos de las cuentas corrientes. Cada uno de estos tipos de cuenta se describe mediante un conjunto de atributos que incluye todos los atributos del conjunto de entidades *cuenta* más otros atributos adicionales.

El banco puede crear dos especializaciones de *cuenta*, por ejemplo, *cuenta_ahorro* y *cuenta_corriente*. Como ya se ha visto, las entidades *cuenta* se describen mediante los atributos *número_cuenta* y *saldo*. El conjunto de entidades *cuenta_ahorro* tendría todos los atributos de *cuenta* y el atributo adicional *tasa_interés*. El conjunto de entidades *cuenta_corriente* tendría todos los atributos de *cuenta* y el atributo adicional *importe_descubierto*.

La especialización se puede aplicar repetidamente para refinar el esquema de diseño. Por ejemplo, los empleados del banco se pueden clasificar también en alguna de las categorías siguientes:

- *oficial*
- *cajero*
- *secretaria*

Cada uno de estos tipos de empleado se describe mediante un conjunto de atributos que incluye todos los atributos del conjunto de entidades *empleado* y otros adicionales. Por ejemplo, las entidades *oficial* se pueden describir además por el atributo *número_despacho*, las entidades *cajero* por los atributos *número_caja* y *horas_semana*, y las entidades *secretaria* por el atributo *horas_semana*. Además, las entidades *secretaria* pueden participar en la relación *secretaria_de*, que identifica a los empleados a los que ayuda cada secretaria.

Cada conjunto de entidades se puede especializar en más de una característica distintiva. En este ejemplo, la característica distintiva entre las entidades *empleado* es el trabajo que desempeña cada empleado. Otra especialización coexistente se puede basar en si cada persona es un trabajador temporal o fijo, lo que da lugar a los conjuntos de entidades *empleado_temporal* y *empleado_fijo*. Cuando se forma más de una especialización en un conjunto de entidades, cada entidad concreta puede pertenecer a varias especializaciones. Por ejemplo, un empleado dado puede ser un empleado temporal y una secretaria.

En términos de los diagramas E-R, la especialización se representa mediante un componente *triangular* etiquetado **ES**, como muestra la Figura 6.20. La etiqueta ES representa, por ejemplo, que cada cliente "es" una persona. La relación ES también se puede denominar relación **superclase-subclase**. Los conjuntos de entidades de nivel superior e inferior se representan como conjuntos de entidades regulares—es decir, como rectángulos que contienen el nombre del conjunto de entidades.

6.7.2 Generalización

El refinamiento a partir del conjunto de entidades inicial en sucesivos niveles de subgrupos de entidades representa un proceso de diseño **descendente** en el que las distinciones se hacen explícitas. El proceso de diseño también puede proceder de forma **ascendente**, en la que varios conjuntos de entidades se sintetizan en un conjunto de entidades de nivel superior basado en características comunes. El diseñador de la base de datos puede haber identificado primero el conjunto de entidades *cliente* con los atributos *id_cliente, nombre_cliente*, *calle_cliente*, *ciudad_cliente* y *calificación_crediticia*, y el conjunto de entidades *empleado* con los atributos *id_empleado, nombre_empleado*, *calle_empleado*, *ciudad_empleado* y *sueldo_empleado*.

Existen analogías entre el conjunto de entidades *cliente* y el conjunto de entidades *empleado* en el sentido de que tienen varios atributos que, conceptualmente, son iguales en los dos conjuntos de entidades: los atributos para el identificador, el nombre, la calle y la ciudad. Esta similitud se puede expresar mediante la **generalización**, que es una relación de contención que existe entre el conjunto de entidades de

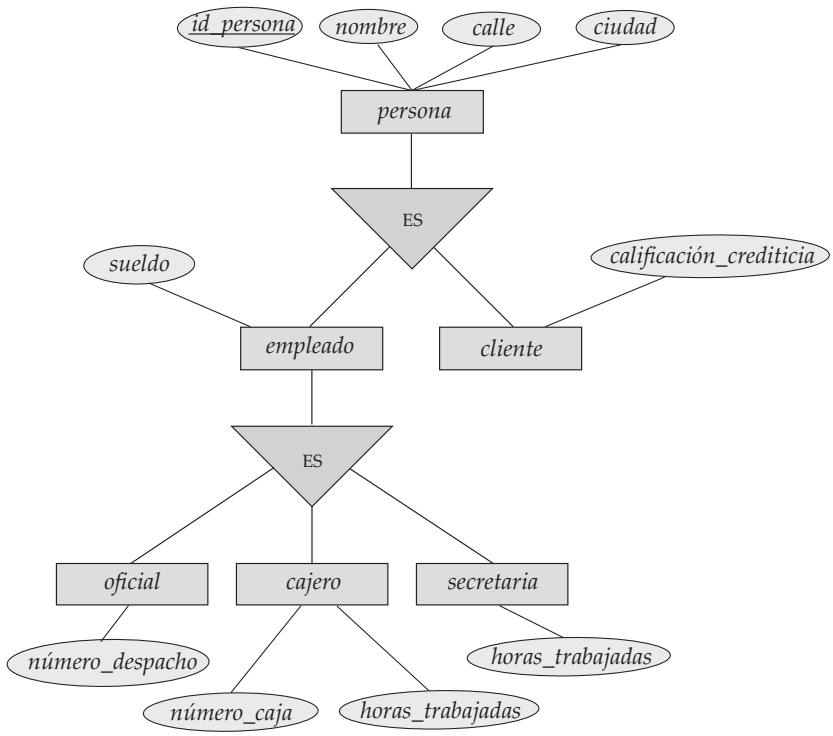


Figura 6.20 Especialización y generalización.

nivel superior y uno o varios conjuntos de entidades de *nivel inferior*. En este ejemplo, *persona* es el conjunto de entidades de nivel superior y *cliente* y *empleado* son conjuntos de entidades de nivel inferior. En este caso, los atributos que son conceptualmente iguales tienen nombres diferentes en los dos conjuntos de entidades de nivel inferior. Para crear generalizaciones los atributos deben tener un nombre común y representarse mediante la entidad de nivel superior *persona.* Se pueden usar los nombres de atributos *id_persona*, *nombre*, *calle* y *ciudad*, como se vio en el ejemplo del Apartado 6.7.1.

Los conjuntos de entidades de nivel superior e inferior también se pueden denominar con los términos **superclase** y **subclase**, respectivamente. El conjunto de entidades *persona* es la superclase de las subclases *cliente* y *empleado*.

A todos los efectos prácticos, la generalización es una inversión simple de la especialización. Se aplicarán ambos procesos, combinados, en el transcurso del diseño del esquema E-R de una empresa. En términos del propio diagrama E-R no se distingue entre especialización y generalización. Los niveles nuevos de representación de las entidades se distinguen (especialización) o sintetizan (generalización) cuando el esquema de diseño llega a expresar completamente la aplicación de la base de datos y los requisitos del usuario de la base de datos. Las diferencias entre los dos enfoques se pueden caracterizar mediante su punto de partida y su objetivo global.

La especialización parte de un único conjunto de entidades; destaca las diferencias entre las entidades del conjunto mediante la creación de diferentes conjuntos de entidades de nivel inferior. Esos conjuntos de entidades de nivel inferior pueden tener atributos o participar en relaciones que no se aplican a todas las entidades del conjunto de entidades de nivel inferior. Realmente, la razón de que el diseñador aplique la especialización es poder representar esas características distintivas. Si *cliente* y *empleado* no tuvieran atributos que no tuvieran las entidades *persona* ni participaran en relaciones diferentes de las relaciones en las que participan las entidades *persona*, no habría necesidad de especializar el conjunto de entidades *persona*.

La generalización parte del reconocimiento de que varios conjuntos de entidades comparten algunas características comunes (es decir, se describen mediante los mismos atributos y participan en los mismos conjuntos de relaciones). Con base en esas similitudes, la generalización sintetiza esos conjuntos de entidades en un solo conjunto de nivel superior. La generalización se usa para destacar las similitudes

entre los conjuntos de entidades de nivel inferior y para ocultar las diferencias; también permite una economía de representación, ya que no se repiten los atributos compartidos.

6.7.3 Herencia de los atributos

Una propiedad crucial de las entidades de nivel superior e inferior creadas mediante la especialización y la generalización es la **herencia de los atributos**. Se dice que los atributos de los conjuntos de entidades de nivel superior son **heredados** por los conjuntos de entidades de nivel inferior. Por ejemplo, *cliente* y *empleado* heredan los atributos de *persona*. Así, *cliente* se describe mediante sus atributos *nombre*, *calle* y *ciudad* y, adicionalmente, por el atributo *id_cliente*; *empleado* se describe mediante sus atributos *nombre*, *calle* y *ciudad* y, adicionalmente, por los atributos *id_empleado* y *sueldo*.

Los conjuntos de entidades de nivel inferior (o subclases) también heredan la participación en los conjuntos de relaciones en los que participa su entidad de nivel superior (o superclase). Los conjuntos de entidades *oficial*, *cajero* y *secretaria* pueden participar en el conjunto de relaciones *trabaja_para*, ya que la superclase *empleado* participa en la relación *trabaja_para*. La herencia de los atributos se aplica a todas las capas de conjuntos de entidades de nivel inferior. Los conjuntos de entidades anteriores pueden participar en cualquier relación en la que participe el conjunto de entidades *persona*.

Tanto si se llega a una porción dada del modelo E-R mediante la especialización como si se hace mediante la generalización, el resultado es básicamente el mismo:

- Un conjunto de entidades de nivel superior con los atributos y las relaciones que se aplican a todos sus conjuntos de entidades de nivel inferior.
- Conjuntos de entidades de nivel inferior con características distintivas que sólo se aplican en un conjunto dado de entidades de nivel inferior.

En lo que sigue, aunque a menudo sólo se haga referencia a la generalización, las propiedades que se estudian corresponden completamente a ambos procesos.

La Figura 6.20 describe una **jerarquía** de conjuntos de entidades. En la figura, *empleado* es un conjunto de entidades de nivel inferior de *persona* y un conjunto de entidades de nivel superior de los conjuntos de entidades *oficial*, *cajero* y *secretaria*. En las jerarquías un conjunto de entidades dado sólo puede estar implicado como conjunto de entidades de nivel inferior en una relación ES; es decir, los conjuntos de entidades de este diagrama sólo tienen **herencia única**. Si un conjunto de entidades es un conjunto de entidades de nivel inferior en más de una relación ES, el conjunto de entidades tiene **herencia múltiple**, y la estructura resultante se denomina *retículo*.

6.7.4 Restricciones a las generalizaciones

Para modelar una empresa con más precisión, el diseñador de la base de datos puede decidir imponer ciertas restricciones sobre una generalización concreta. Un tipo de restricción implica la determinación de las entidades que pueden formar parte de un conjunto de entidades de nivel inferior dado. Esa pertenencia puede ser una de las siguientes:

- **Definida por la condición**. En los conjuntos de entidades de nivel inferior definidos por la condición, la pertenencia se evalúa en función del cumplimiento de una condición o predicado explícito por la entidad. Por ejemplo, supóngase que el conjunto de entidades de nivel superior *cuenta* tiene el atributo *tipo_cuenta*. Todas las entidades *cuenta* se evalúan según el atributo *tipo_cuenta* que las define. Sólo las entidades que satisfacen la condición *tipo_cuenta* = "cuenta de ahorro" pueden pertenecer al conjunto de entidades de nivel inferior *cuenta_ahorro*. Todas las entidades que satisfacen la condición *tipo_cuenta* = "cuenta corriente" se incluyen en *cuenta_corriente*. Dado que todas las entidades de nivel inferior se evalúan en función del mismo atributo (en este caso, *tipo_cuenta*), se dice que este tipo de generalización está **definida por el atributo**.
- **Definida por el usuario**. Los conjuntos de entidades de nivel inferior definidos por el usuario no están restringidos por una condición de pertenencia; más bien, el usuario de la base de datos asigna las entidades a un conjunto de entidades dado. Por ejemplo, supóngase que, después de tres meses de trabajo, los empleados del banco se asignan a uno de los cuatro grupos de trabajo.

En consecuencia, los grupos se representan como cuatro conjuntos de entidades de nivel inferior del conjunto de entidades de nivel superior *empleado*. No se asigna cada empleado a una entidad grupo concreta automáticamente de acuerdo con una condición explícita que lo defina. En vez de eso, la asignación al grupo la lleva a cabo el usuario que toma persona a persona. La asignación se implementa mediante una operación que añade cada entidad a un conjunto de entidades.

Un segundo tipo de restricciones tiene relación con la pertenencia de las entidades a más de un conjunto de entidades de nivel inferior de la generalización. Los conjuntos de entidades de nivel inferior pueden ser de uno de los tipos siguientes:

- **Disjuntos**. La *restricción sobre la condición de disjunción* exige que cada entidad no pertenezca a más de un conjunto de entidades de nivel inferior. En el ejemplo, cada entidad *cuenta* sólo puede cumplir una condición del atributo *tipo_cuenta*; cada entidad puede ser una cuenta de ahorro o una cuenta corriente, pero no ambas cosas a la vez.
- **Solapados**. En las *generalizaciones solapadas* la misma entidad puede pertenecer a más de un conjunto de entidades de nivel inferior de la generalización. Como ilustración, considérese el ejemplo del grupo de trabajo de empleados y supóngase que algunos directores participan en más de un grupo de trabajo. Cada empleado, por tanto, puede aparecer en más de uno de los conjuntos de entidades grupo que son conjuntos de entidades de nivel inferior de *empleado*. Por tanto, la generalización es solapada.

  Como ejemplo adicional, supóngase que la generalización aplicada a los conjuntos de entidades *cliente* y *empleado* conduce a un conjunto de entidades de nivel superior *persona*. La generalización es solapada si los empleados también pueden ser clientes.

El solapamiento de las entidades de nivel inferior es el caso predeterminado; la restricción sobre la condición de disjunción se debe imponer explícitamente a la generalización (o especialización). La restricción sobre condición de disjunción se puede denotar en los diagramas E-R añadiendo la palabra *disjunta* junto al símbolo del triángulo.

Una última restricción, la **restricción de completitud** sobre una generalización o especialización, especifica si una entidad del conjunto de entidades de nivel superior debe pertenecer, al menos, a uno de los conjuntos de entidades de nivel inferior de la generalización o especialización. Esta restricción puede ser de uno de los tipos siguientes:

- **Generalización** o **especialización total**. Cada entidad de nivel superior debe pertenecer a un conjunto de entidades de nivel inferior.
- **Generalización** o **especialización parcial**. Puede que alguna entidad de nivel superior no pertenezca a ningún conjunto de entidades de nivel inferior.

La generalización parcial es la predeterminada. Se puede especificar la generalización total en los diagrama E-R usando una línea doble para conectar el rectángulo que representa el conjunto de entidades de nivel superior con el símbolo del triángulo. (Esta notación es parecida a la usada para la participación total en una relación).

La generalización de *cuenta* es total: todas las entidades *cuenta* deben ser cuentas de ahorro o cuentas corrientes. Como el conjunto de entidades de nivel superior al que se llega mediante la generalización suele estar compuesto únicamente de entidades de los conjuntos de entidades de nivel inferior, la restricción de completitud para los conjuntos de entidades de nivel superior generalizados suele ser total. Cuando la restricción es parcial, las entidades de nivel superior no están limitadas a aparecer en los conjuntos de entidades de nivel inferior. Los conjuntos de entidades grupo de trabajo ilustran una especialización parcial. Como los empleados sólo se asignan a cada grupo después de llevar tres meses en el trabajo, puede que algunas entidades *empleado* no pertenezcan a ninguno de los conjuntos de entidades grupo de nivel inferior.

Los conjuntos de entidades equipo se pueden caracterizar mejor como especialización de *empleado* parcial y solapada. La generalización de *cuenta_corriente* y *cuenta_ahorro* en *cuenta* es una generalización total y disjunta. Las restricciones de completitud y sobre la condición de disjunción, sin embargo, no

dependen una de la otra. Las características de las restricciones también pueden ser parcial—disjunta y total—solapada.

Es evidente que algunos requisitos de inserción y de borrado son consecuencia de las restricciones que se aplican a una generalización o especialización dada. Por ejemplo, cuando se impone una restricción de completitud total, las entidades insertadas en un conjunto de entidades de nivel superior se deben insertar, al menos, en uno de los conjuntos de entidades de nivel inferior. Con una restricción de definición por condición, todas las entidades de nivel superior que cumplen la condición se deben insertar en ese conjunto de entidades de nivel inferior. Finalmente, las entidades que se borren de los conjuntos de entidades de nivel superior, se deben borrar también de todos los conjuntos de entidades de nivel inferior asociados a los que pertenezcan.

6.7.5 Agregación

Una limitación del modelo E-R es que no es posible expresar relaciones entre las relaciones. Para ilustrar la necesidad de estos constructores, considérese la relación ternaria *trabaja_en*, que se ya se ha visto anteriormente, entre *empleado*, *sucursal* y *trabajo* (véase la Figura 6.12). Supóngase ahora que se desea registrar el director responsable de las tareas realizadas por cada empleado de cada sucursal; es decir, se desea registrar al director responsable de las combinaciones (*empleado*, *sucursal*, *trabajo*). Supóngase que existe un conjunto de entidades *director*.

Una alternativa para representar esta relación es crear una relación cuaternaria *dirige* entre *empleado*, *sucursal*, *trabajo* y *director* (se necesita una relación cuaternaria—una relación binaria entre *director* y *empleado* no permitiría representar las combinaciones (*sucursal*, *trabajo*) de cada empleado que son responsabilidad de cada director). Mediante los constructores de modelado básicos del modelo E-R se obtiene el diagrama E-R de la Figura 6.21 (por simplificar se han omitido los atributos de los conjuntos de entidades).

Parece que los conjuntos de relaciones *trabaja_en* y *dirige* se pueden combinar en un solo conjunto de relaciones. No obstante, no se deben combinar en una sola relación, ya que puede que algunas combinaciones *empleado*, *sucursal*, *trabajo* no tengan director.

No obstante, hay información redundante en la figura obtenida, ya que cada combinación *empleado*, *sucursal*, *trabajo* de *dirige* también está en *trabaja_en*. Si el director fuese un valor en lugar de una entidad *director*, se podría hacer que *director* fuese un atributo multivalorado de la relación *trabaja_en*. Pero eso dificulta (tanto lógicamente como en coste de ejecución) encontrar, por ejemplo, las tripletas empleado-sucursal-trabajo de las que es responsable cada director. Como el director es una entidad *director*, esta alternativa queda descartada en cualquier caso.

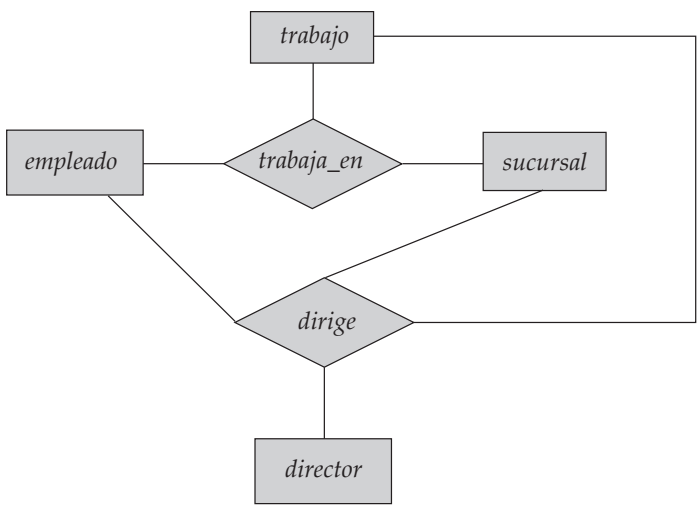


Figura 6.21 Diagrama E-R con relaciones redundantes.

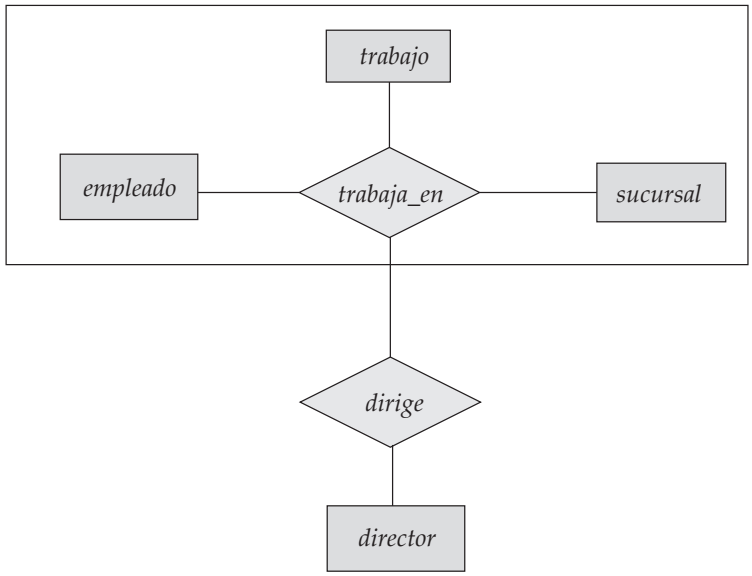


Figura 6.22 Diagrama E-R con agregación.

La mejor forma de modelar una situación como la descrita es usar la agregación. La **agregación** es una abstracción a través de la cual las relaciones se tratan como entidades de nivel superior. Así, para este ejemplo, se considera el conjunto de relaciones *trabaja_en* (que relaciona los conjuntos de entidades *empleado*, *sucursal* y *trabajo*) como el conjunto de entidades de nivel superior denominado *trabaja_en*. Ese conjunto de entidades se trata de la misma forma que cualquier otro conjunto de entidades. Se puede crear entonces la relación binaria *dirige* entre *trabaja_en* y *director* para representar al responsable de cada tarea. La Figura 6.22 muestra una notación para la agregación que se usa habitualmente para representar esta situación.

6.7.6 Notaciones E-R alternativas

La Figura 6.23 resume el conjunto de símbolos que se ha usado en los diagramas E-R. No hay una norma universal para la notación de los diagramas E-R, y cada libro y cada programa informático de diagramas E-R usa notaciones diferentes.

La Figura 6.24 indica alguna de las notaciones alternativas que más se usan. Los conjuntos de entidades se pueden representar como rectángulos con el nombre por fuera, y los atributos relacionados unos debajo de los otros dentro del rectángulo. Los atributos clave primaria se indican relacionándolos en la parte superior, con una línea que los separa de los demás atributos.

Las restricciones de cardinalidad se pueden indicar de varias formas, como se muestra en la Figura 6.24. Las etiquetas $*$ y 1 en los segmentos que salen de las relaciones se usan a menudo para denotar relaciones varios a varios, uno a uno y varios a uno, como muestra la figura. El caso de uno a varios es simétrico con el de varios a uno y no se muestra. En otra notación alternativa de la figura los conjuntos de relaciones se representan mediante líneas entre los conjuntos de entidades, sin rombos; por tanto, sólo se podrán modelar relaciones binarias. Las restricciones de cardinalidad en esta notación se muestran mediante la notación "pata de gallo", como en la figura.

Desafortunadamente no existe una notación E-R normalizada. La notación que se usa en este libro, con rectángulos, rombos y elipses se denomina notación de Chen, y la usó Chen en el artículo que introdujo el concepto de modelado E-R. El Instituto Nacional de EEUU para Normalización y Tecnología (U.S. National Institute for Standards and Technology) definió una norma denominada IDEF1X en 1993, que usa la notación de pata de gallo. IDEF1X también incluye gran variedad de notaciones diferentes que no se han mostrado, incluidas las barras verticales en los segmentos de las relaciones para indicar participación total y los círculos vacíos para denotar participación parcial. Hay gran variedad de herramientas para la construcción de diagramas E-R, cada una de las cuales tiene sus propias variantes en cuanto a la notación. Véanse las referencias en las notas bibliográficas para obtener más información.

6.8 Diseño de una base de datos para un banco

Ahora se centrará la atención en los requisitos de diseño de la base de datos de una entidad bancaria con más detalle y se desarrollará un diseño más realista, aunque también más complicado, de lo que se ha visto en los ejemplos anteriores. No obstante, no se intentará modelar cada aspecto del diseño de la base de datos para el banco; se considerarán sólo unos cuantos aspectos para ilustrar el proceso de diseño de bases de datos.

Se aplicarán las dos fases iniciales del diseño de bases de datos, es decir, la recopilación de los requisitos de datos y el diseño del esquema conceptual, al ejemplo de entidad bancaria. Se usará el modelo de datos E-R para traducir los requisitos de los usuarios a un esquema de diseño conceptual que se representará como diagrama E-R.

Finalmente, el resultado del proceso de diseño E-R será el esquema de una base de datos relacional. En el Apartado 6.9 se considerará el proceso de generación del diseño relacional a partir de un diseño E-R.

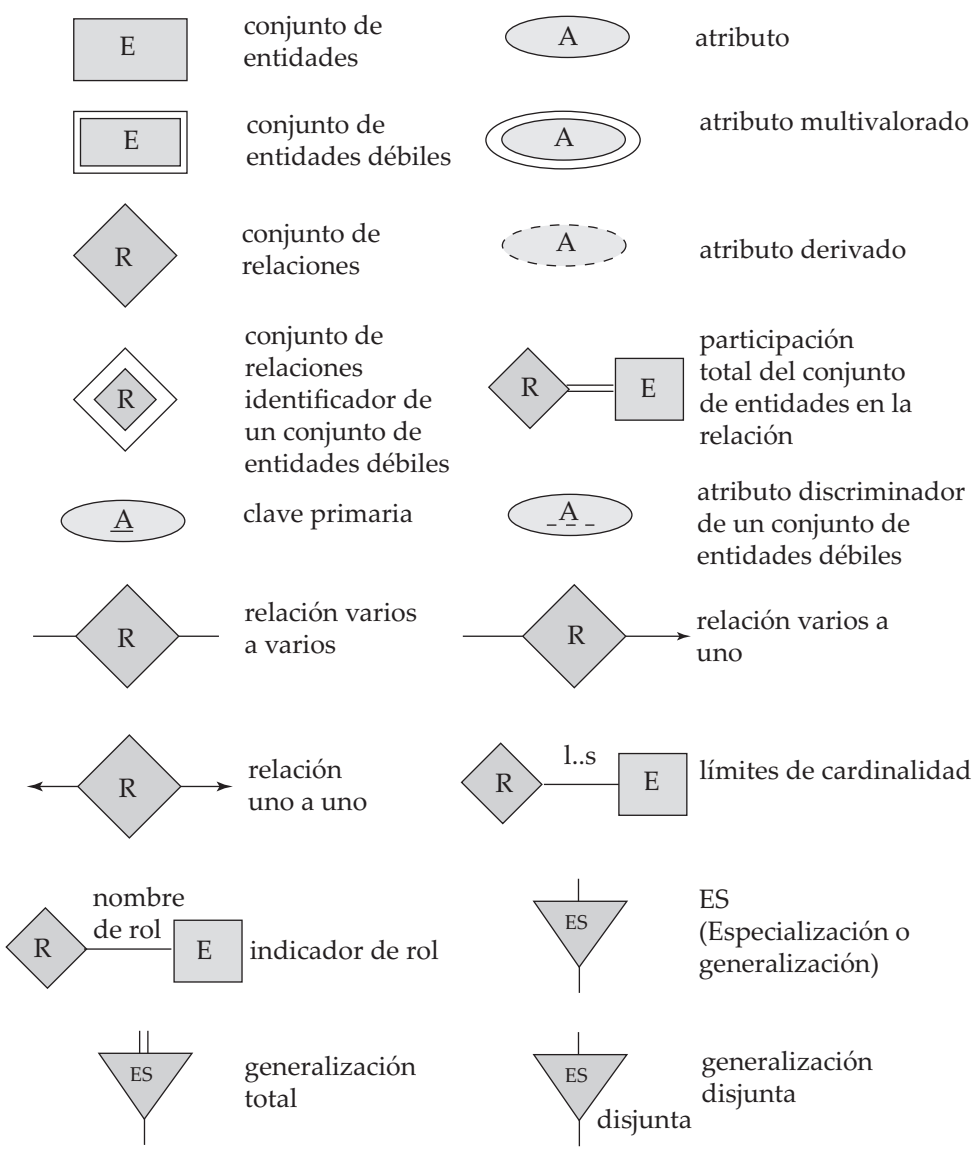


Figura 6.23 Símbolos usados en la notación E-R.

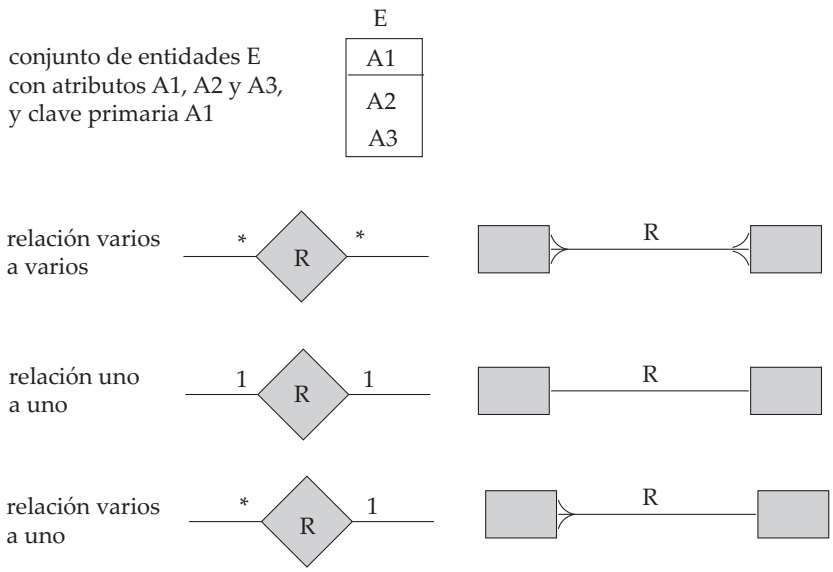


Figura 6.24 Notaciones E-R alternativas.

Antes de comenzar con el diseño de la base de datos de la entidad bancaria se describirán brevemente las alternativas de diseño E-R entre las que pueden escoger los diseñadores de bases de datos.

6.8.1 Alternativas de diseño E-R

El modelo de datos E-R permite gran flexibilidad en el diseño de los esquemas de las bases de datos para modelar una empresa dada. A continuación se sugiere la manera en que el diseñador de bases de datos puede escoger entre una amplia gama de alternativas. Entre las decisiones que debe tomar el diseñador están:

- Si usar atributos o conjuntos de entidades para representar los objetos (se ha estudiado en el Apartado 6.5.1).
- Si los conceptos del mundo real se expresan mejor mediante conjuntos de entidades o mediante conjuntos de relaciones (Apartado 6.5.2).
- Si usar relaciones ternarias o pares de relaciones binarias (Apartado 6.5.3).
- Si usar conjuntos de entidades fuertes o débiles (Apartado 6.6); cada conjunto de entidades fuertes y sus conjuntos de entidades débiles se puede considerar como un solo "objeto" de la base de datos, ya que las entidades débiles dependen existencialmente de la entidad fuerte.
- Si es adecuado usar la generalización (Apartado 6.7.2); la generalización, o la jerarquía de relaciones ES, contribuye a la modularidad al permitir que los atributos comunes de entidades parecidas se representen en un solo sitio del diagrama E-R.
- Si es adecuado usar la agregación (Apartado 6.7.5); la agregación agrupa parte del diagrama E-R en un solo conjunto de entidades, lo que permite tratar el conjunto de entidades agregadas como una sola unidad sin necesidad de preocuparse por los detalles de su estructura interna.

Se puede ver que los diseñadores de bases de datos necesitan tener una buena comprensión de la empresa que se modela para poder adoptar las diferentes decisiones de diseño necesarias.

6.8.2 Requisitos de datos de la base de datos bancaria

La especificación inicial de los requisitos de los usuarios se puede basar en entrevistas con los usuarios de la base de datos y en el análisis propio del diseñador de la empresa. La descripción que surge de esta

fase de diseño sirve de base para concretar la estructura conceptual de la base de datos. La siguiente lista describe las principales características de la entidad bancaria.

- El banco está organizado en sucursales. Cada sucursal está ubicada en una ciudad concreta y se identifica con un nombre único. El banco supervisa los activos de cada sucursal.
- Los clientes del banco se identifican mediante su valor de *id_cliente*. El banco almacena cada nombre de cliente, y la calle y la ciudad donde vive cada cliente. Los clientes pueden tener cuentas y pueden solicitar préstamos. Cada cliente puede estar asociado con un empleado del banco concreto, que puede actuar como responsable de préstamos o como asesor personal de ese cliente.
- Los empleados del banco se identifican mediante su valor de *id_empleado*. La administración del banco almacena el nombre y el número de teléfono de cada empleado, el nombre de los subordinados de cada empleado, y el número *id_empleado* del jefe de cada empleado. El banco también mantiene un registro de la fecha de incorporación a la empresa del empleado y, por tanto, de su antigüedad.
- El banco ofrece dos tipos de cuentas: cuentas de ahorro y cuentas corrientes. Las cuentas pueden tener como titular a más un cliente, y cada cliente puede tener más de una cuenta. Cada cuenta tiene asignado un número de cuenta único. El banco mantiene un registro del saldo de cada cuenta y de la fecha más reciente en que cada titular de la cuenta tuvo acceso a esa cuenta. Además, cada cuenta de ahorro tiene un tipo de interés y para cada cuenta corriente se registran los descubiertos generados.
- Cada préstamo se genera en una sucursal concreta y pueden solicitarlo uno o más clientes. Cada préstamo se identifica mediante un número de préstamo único. Para cada préstamo el banco mantiene un registro del importe del préstamo y de los pagos realizados y pendientes. Aunque los números de los pagos del préstamo no identifican de forma unívoca cada pago entre los de todos los préstamos del banco, el número de pago sí que identifica cada pago de un préstamo concreto. De cada pago se registran la fecha y el importe.

En las entidades bancarias reales, el banco realiza un seguimiento de los abonos y cargos realizados en las cuentas de ahorros y en las cuentas corrientes, igual que mantiene un registro de los pagos de los préstamos. Debido a que los requisitos del modelo para ese seguimiento son parecidos, y con objeto de mantener las aplicaciones de ejemplo con un tamaño reducido, en este modelo no se realiza el seguimiento de esos abonos y cargos.

6.8.3 Conjuntos de entidades de la base de datos bancaria

La especificación de los requisitos de datos sirve como punto de partida para la construcción del esquema conceptual de la base de datos. A partir de la especificación que aparece en el Apartado 6.8.2 se comienzan a identificar los conjuntos de entidades y sus atributos:

- El conjunto de entidades *sucursal*, con los atributos *nombre_sucursal*, *ciudad_sucursal* y *activos*.
- El conjunto de entidades *cliente*, con los atributos *id_cliente*, *nombre_cliente*, *calle_cliente* y *ciudad_cliente*. Un posible atributo adicional es *nombre_asesor*.
- El conjunto de entidades *empleado*, con los atributos *id_empleado, nombre_empleado, número_teléfono, sueldo y jefe*. Algunas características descriptivas adicionales son el atributo multivalorado *nombre_subordinado*, el atributo básico *fecha_contratación* y el atributo derivado *antigüedad*.
- Dos conjuntos de entidades cuenta—*cuenta_ahorro* y *cuenta_corriente*—con los atributos comunes *número_cuenta* y *saldo*; además, *cuenta_ahorro* tiene el atributo *tipo_interés* y *cuenta_corriente* tiene el atributo *descubierto*.
- El conjunto de entidades *préstamo*, con los atributos *número_préstamo*, *importe* y *sucursal_origen*.
- El conjunto de entidades débiles *pago_préstamo*, con los atributos *número_pago*, *fecha_pago* e *importe_pago*.

6.8.4 Conjuntos de relaciones de la base de datos bancaria

Volviendo ahora al esquema de diseño rudimentario del Apartado 6.8.3 se pueden especificar los conjuntos de relaciones y correspondencias de cardinalidades siguientes. En el proceso también se perfeccionan algunas de las decisiones tomadas anteriormente en relación con los atributos de los conjuntos de entidades.

- *prestatario*, un conjunto de relaciones varios a varios entre *cliente* y *préstamo*.
- *sucursal_préstamo*, un conjunto de relaciones varios a uno que indica la sucursal en que se ha originado un préstamo. Obsérvese que este conjunto de relaciones sustituye al atributo *sucursal _origen* del conjunto de entidades *préstamo.*
- *pago_préstamo*, un conjunto de relaciones uno a varios de *préstamo* a *pago*, que documenta que se ha realizado un pago de un préstamo.
- *impositor*, con el atributo de relación *fecha_acceso*, un conjunto de relaciones varios a varios entre *cliente* y *cuenta*, que indica que un cliente posee una cuenta.
- *asesor_personal*, con el atributo de relación *tipo*, un conjunto de relaciones varios a uno que expresa que un cliente puede ser asesorado por un empleado del banco, y que un empleado del banco puede asesorar a uno o más clientes. Obsérvese que este conjunto de relaciones ha sustituido al atributo *nombre_asesor* del conjunto de entidades *cliente*.
- *trabaja_para*, un conjunto de relaciones entre entidades *empleado* con los indicadores de rol *jefe* y *trabajador*; la correspondencia de cardinalidades expresa que cada empleado trabaja para un único jefe, y que cada jefe supervisa a uno o más empleados. Obsérvese que este conjunto de relaciones ha sustituido al atributo *jefe* de *empleado*.

6.8.5 Diagrama E-R de la base de datos bancaria

Conforme a lo estudiado en el Apartado 6.8.4 se presenta ahora el diagrama E-R completo del ejemplo de la entidad bancaria. La Figura 6.25 muestra la representación completa de un modelo conceptual del banco, expresada en términos de los conceptos E-R. El diagrama incluye los conjuntos de entidades, los atributos, los conjuntos de relaciones y las correspondencias de cardinalidades a los que se ha llegado mediante el proceso de diseño de los Apartados 6.8.2 y 6.8.3 y que se han perfeccionado en el Apartado 6.8.4.

El diagrama E-R de esta visión simplificada de una entidad bancaria ya es bastante complejo. Los diagramas E-R de las empresas reales no se pueden dibujar en una sola página y hay que dividirlos en varias partes. Puede hacer falta que las entidades aparezcan varias veces en diferentes partes del diagrama. Los atributos de cada entidad se muestran en una sola aparición de la entidad (preferiblemente la primera) y todas las demás apariciones de la entidad se muestran sin atributos.

6.9 Reducción a esquemas relacionales

Las bases de datos que se ajustan a un esquema de bases de datos E-R se pueden representar mediante conjuntos de esquemas de relación. Para cada conjunto de entidades y para cada conjunto de relaciones de la base de datos hay un solo esquema de relación a la que se asigna el nombre del conjunto de entidades o del conjunto de relaciones correspondiente.

Tanto el modelo E-R de bases de datos como el relacional son representaciones abstractas y lógicas de empresas del mundo real. Como los dos modelos usan principios de diseño parecidos, los diseños E-R se pueden convertir en diseños relacionales.

En este apartado se describe la manera de representar los esquemas E-R mediante esquemas de relación y el modo de asignar las restricciones que surgen del modelo E-R a restricciones de los esquemas de relación.

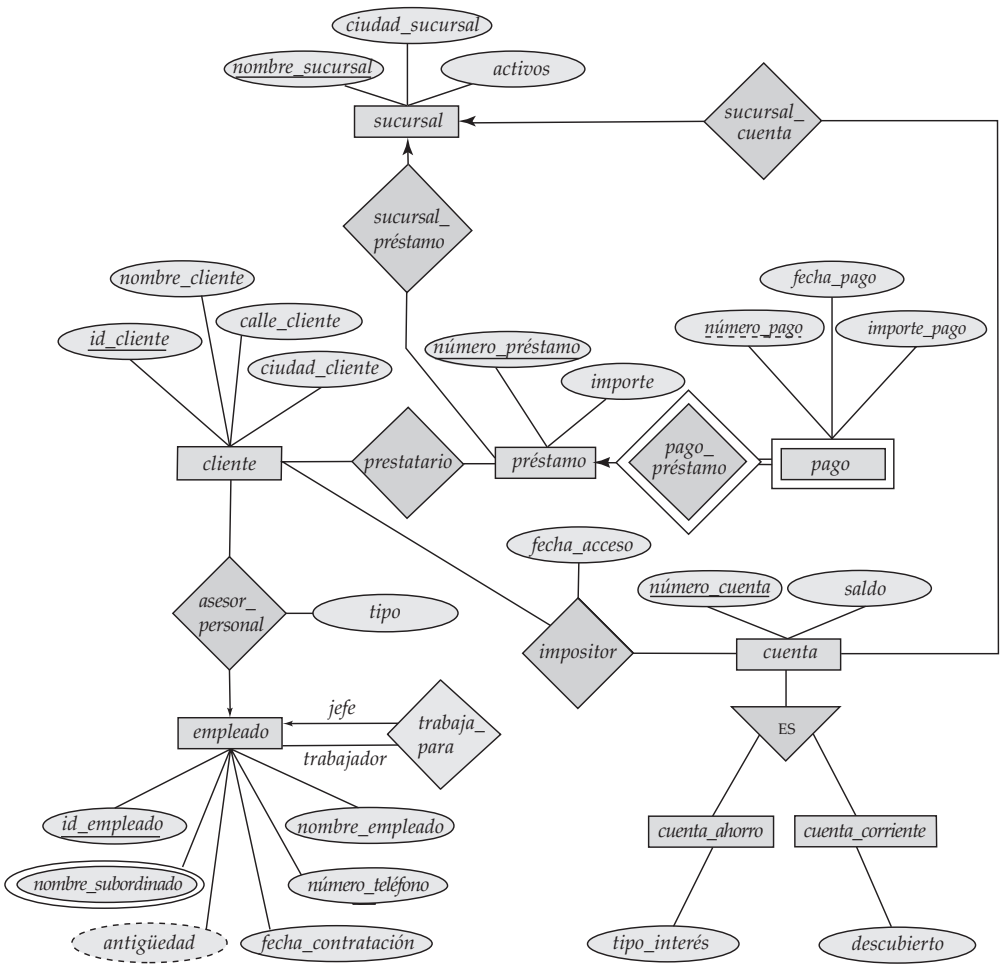


Figura 6.25 Diagrama E-R de una entidad bancaria.

6.9.1 Representación de los conjuntos de entidades fuertes

Sea E un conjunto de entidades fuertes con los atributos descriptivos $a_1, a_2, \ldots, a_n$. Esta entidad se representa mediante un esquema denominado E con n atributos distintos. Cada tupla de las relaciones de este esquema corresponde a una entidad del conjunto de entidades E (más adelante, en el Apartado 6.9.4, se describe la manera de tratar los atributos compuestos y los multivalorados).

Para los esquemas derivados de los conjuntos de entidades fuertes la clave primaria del conjunto de entidades sirve de clave primaria de los esquemas resultantes. Esto se deduce directamente del hecho de que cada tupla corresponde a una entidad concreta del conjunto de entidades.

Como ilustración, considérese el conjunto de entidades *préstamo* del diagrama E-R de la Figura 6.7. Este conjunto de entidades tiene dos atributos: *número_préstamo* e *importe*. Este conjunto de entidades se representa mediante un esquema denominado *préstamo*, con dos atributos:

$$préstamo = (\underline{número_préstamo}, \underline{importe})$$

Obsérvese que, como *número_préstamo* es la clave primaria del conjunto de entidades, también es la clave primaria del esquema de la relación.

En la Figura 6.26 se muestra una relación de este esquema. La tupla

(P-17, 1.000)

número_préstamo	*importe*
P-11	900
P-14	1.500
P-15	1.500
P-16	1.300
P-17	1.000
P-23	2.000
P-93	500

Figura 6.26 La tabla *préstamo*

significa que el número de préstamo P-17 tiene un importe de 1.000 €. Se pueden añadir entidades nuevas a la base de datos insertando tuplas en las relaciones correspondientes. También se pueden borrar o modificar entidades modificando las tuplas correspondientes.

6.9.2 Representación de los conjuntos de entidades débiles

Sea A un conjunto de entidades débiles con los atributos $a_1, a_2, \ldots, a_m$. Sea B el conjunto de entidades fuertes del que A depende. La clave primaria de B consiste en los atributos $b_1,\ b_2, \ldots, b_n$. El conjunto de entidades A se representa mediante el esquema de relación denominado A con una columna por cada miembro del conjunto:

$$\{a_1, a_2, \ldots, a_m\} \cup \{b_1, b_2, \ldots, b_n\}$$

Para los esquemas derivados de conjuntos de entidades débiles la combinación de la clave primaria del conjunto de entidades fuertes y del discriminador del conjunto de entidades débiles sirve de clave primaria del esquema. Además de crear una clave primaria, también se crea una restricción de clave externa para la relación A, que especifica que los atributos $b_1,\ b_2, \ldots, b_n$ hacen referencia a la clave primaria de la relación B. La restricción de clave externa garantiza que por cada tupla que represente a una entidad débil exista la tupla correspondiente que representa a la entidad fuerte correspondiente.

Como ilustración, considérese el conjunto de entidades *pago* del diagrama E-R de la Figura 6.19. Este conjunto de entidades tiene tres atributos: *número_pago*, *fecha_pago* e *importe_pago*. La clave primaria del conjunto de entidades *préstamo*, de la que *pago* depende, es *número_préstamo*. Por tanto, *pago* se representa mediante un esquema con cuatro atributos:

$$pago = (\underline{número_préstamo, número_pago}, fecha_pago, importe)$$

La clave primaria consiste en la clave primaria de *préstamo* y el discriminador de *pago* (*número_pago*). También se crea una restricción de clave externa para el esquema *pago*, con el atributo *número_pago* que hace referencia a la clave primaria del esquema *préstamo*.

6.9.3 Representación de los conjuntos de relaciones

Sea R un conjunto de relaciones, sea $a_1, a_2, \ldots, a_m$ el conjunto de atributos formado por la unión de las claves primarias de cada uno de los conjuntos de entidades que participan en R, y sean $b_1, b_2, \ldots, b_n$ los atributos descriptivos de R (si los hay). El conjunto de relaciones se representa mediante el esquema de relación R, con un atributo por cada uno de los miembros del conjunto:

$$\{a_1, a_2, \ldots, a_m\} \cup \{b_1, b_2, \ldots, b_n\}$$

Ya se ha descrito, en el Apartado 6.3.2.2, la manera de escoger la clave primaria de un conjunto de relaciones binarias. Como se vio en ese apartado, tomar todos los atributos de las claves primarias de todos los conjuntos de entidades primarias sirve para identificar una tupla concreta pero, para los conjuntos de relaciones uno a uno, varios a uno y uno a varios, esto resulta un conjunto de atributos mayor del que hace falta en la clave primaria. En vez de eso, la clave primaria se escoge de la manera siguiente:

- Para las relaciones binarias varios a varios la unión de los atributos de clave primaria de los conjuntos de entidades participantes pasa a ser la clave primaria.
- Para los conjuntos de relaciones binarias uno a uno la clave primaria de cualquiera de los conjuntos de entidades puede escogerse como clave primaria de la relación. La elección del conjunto de entidades de entre los relacionados por el conjunto de relaciones puede realizarse de manera arbitraria.
- Para los conjuntos de relaciones binarias varios a uno o uno a varios la clave primaria del conjunto de entidades de la parte "varios" de la relación sirve de clave primaria.
- Para los conjuntos de relaciones n-arias sin flechas en los segmentos la unión de los atributos de clave primaria de los conjuntos de entidades participantes pasa a ser la clave primaria.
- Para los conjuntos de relaciones n-arias con una flecha en uno de los segmentos las claves primarias de los conjuntos de entidades que no están en el lado "flecha" del conjunto de relaciones sirven de clave primaria del esquema. Recuérdese que sólo se permite una flecha saliente por conjunto de relaciones.

También se crean restricciones de clave externa para la relación R de la manera siguiente. Para cada conjunto de entidades E_i relacionado con el conjunto de relaciones R se crea una restricción de clave primaria de la relación R, con los atributos de R que eran atributos de clave primaria de E que hacen referencia a la clave primaria de la relación que representa E_i.

Como ejemplo, considérese el conjunto de relaciones *prestatario* del diagrama E-R de la Figura 6.7. Este conjunto de relaciones implica a los dos conjuntos de entidades siguientes:

- *cliente*, con la clave primaria *id_cliente*
- *préstamo*, con la clave primaria *número_préstamo*

Como el conjunto de relaciones no tiene ningún atributo, el esquema *prestatario* tiene dos atributos:

$$\textit{prestatario} = (\underline{\textit{id_cliente, número_préstamo}})$$

La clave primaria de la relación *prestatario* es la unión de los atributos de clave primaria de *cliente* y de *préstamo*. También se crean dos restricciones de clave externa para la relación *prestatario*, con el atributo *id_cliente* que hace referencia a la clave primaria de *cliente* y el atributo *número_préstamo* que hace referencia a la clave primaria de *préstamo*.

6.9.3.1 Redundancia de los esquemas

Los conjuntos de relaciones que enlazan los conjuntos de entidades débiles con el conjunto correspondiente de entidades fuertes se tratan de manera especial. Como se hizo notar en el Apartado 6.6, estas relaciones son varios a uno y no tienen atributos descriptivos. Además, la clave primaria de los conjuntos de entidades débiles incluye la clave primaria de los conjuntos de entidades fuertes. En el diagrama E-R de la Figura 6.19, el conjunto de entidades débiles *pago* depende del conjunto de entidades fuertes *préstamo* a través del conjunto de relaciones *pago_préstamo*. La clave primaria de *pago* es *número_préstamo*, *número_pago* y la clave primaria de préstamo es *número_préstamo*. Como *pago_préstamo* no tiene atributos descriptivos, el esquema *pago_préstamo* tiene dos atributos, *número_préstamo* y *número_pago*. El esquema del conjunto de entidades *pago* tiene cuatro atributos, *número_préstamo*, *número_pago*, *fecha_pago* e *importe_pago*. Cada combinación (*número_préstamo*, *número_pago*) de una relación de *pago_préstamo* también se halla presente en el esquema de relación *pago*, y viceversa. Por tanto, el esquema *pago_préstamo* es redundante. En general, el esquema de los conjuntos de relaciones que enlazan los conjuntos de entidades débiles con su conjunto correspondiente de entidades fuertes es redundante y no hace falta que esté presente en el diseño de la base de datos relacional basado en el diagrama E-R.

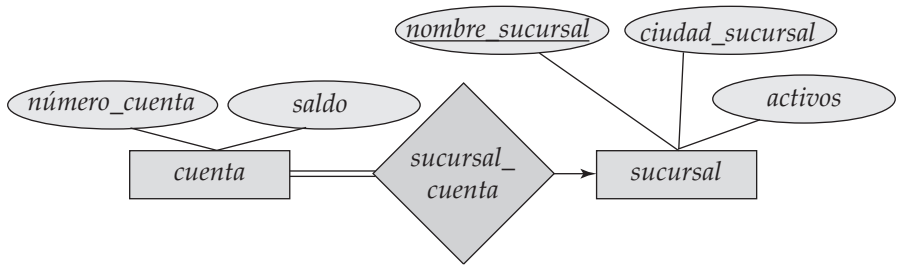


Figura 6.27 Diagrama E-R.

6.9.3.2 Combinación de esquemas

Considérese un conjunto AB de relaciones varios a uno del conjunto de entidades A al conjunto de entidades B. Usando el esquema de construcción de esquemas de relación descrito previamente se consiguen tres esquemas: A, B y AB. Supóngase, además, que la participación de A en la relación es total; es decir, todas las entidades a del conjunto de entidades A deben participar en la relación AB. Entonces se pueden combinar los esquemas A y AB para formar un solo esquema consistente en la unión de los atributos de los dos esquemas.

Como ilustración, considérese el diagrama E-R de la Figura 6.27. La línea doble del diagrama E-R indica que la participación de *cuenta* en *cuenta_sucursal* es total. Por tanto, no puede haber ninguna cuenta que no esté asociada a alguna sucursal. Además, el conjunto de relaciones *cuenta_sucursal* es varios a uno de *cuenta* a *sucursal*. Por lo tanto, se puede combinar el esquema de *cuenta_sucursal* con el esquema de *cuenta* y sólo se necesitan los dos esquemas siguientes:

- *cuenta* = (*número_cuenta*, *saldo*, *nombre_sucursal*)
- *sucursal* = (*nombre_sucursal, ciudad_sucursal, activos*)

En el caso de las relaciones uno a uno, el esquema de relación del conjunto de relaciones puede combinarse con el esquema de cualquiera de los conjuntos de entidades.

Se pueden combinar esquemas aunque la participación sea parcial, usando los valores nulos; en el ejemplo anterior se almacenarían valores nulos para el atributo *nombre_sucursal* de las cuentas que no tengan sucursal asociada.

La clave primaria del esquema combinado es la clave primaria del conjunto de entidades en cuyo esquema se ha fusionado el conjunto de relaciones. En el ejemplo anterior, la clave primaria es *número _cuenta*.

En circunstancias normales el esquema que representa al conjunto de relaciones habría tenido restricciones de clave externa que harían referencia a cada uno de los conjuntos de entidades que participan en el conjunto de relaciones. En este caso se ha descartado la restricción que hace referencia al conjunto de entidades en cuyo esquema se ha fusionado el conjunto de relaciones y se ha añadido las demás restricciones de clave externa al esquema combinado. En el ejemplo anterior, se descarta la restricción de clave externa que hace referencia a *cuenta*, pero se conserva la restricción de clave externa en la que *nombre_sucursal* hace referencia a *sucursal* como restricción del esquema combinado *cuenta*.

6.9.4 Atributos compuestos y multivalorados

Los atributos compuestos se tratan mediante la creación de un atributo diferente para cada uno de los atributos componentes; no se crea ningún atributo para el atributo compuesto propiamente dicho. Supóngase que *dirección* es un atributo compuesto del conjunto de entidades *cliente* y que los componentes de *dirección* son *calle* y *ciudad*. El esquema generado a partir de *cliente* contiene las columnas *calle_dirección* y *ciudad_dirección*; no hay ningún atributo ni esquema para *dirección*. Este asunto se vuelve a tratar en el Apartado 7.2.

Se ha visto que en los diagramas E-R los atributos se suelen asignar directamente a las columnas de los esquemas de relación correspondientes. Los atributos multivalorados, sin embargo, son una excepción; para estos atributos se crean esquemas de relación nuevos.

Para cada atributo multivalorado M se crea un esquema de relación E con un atributo A que corresponde a M y a los atributos correspondientes a la clave primaria del conjunto de entidades o de relaciones del que M es atributo.

Como ilustración, considérese el diagrama E-R de la Figura 6.25. El diagrama incluye el conjunto de entidades *empleado* con el atributo multivalorado *nombre_subordinado*. La clave primaria de *empleado* es *id_empleado*. Para este atributo multivalorado se crea el esquema de relación

$$nombre_subordinado\ (\underline{id_empleado, nombre_subordinado})$$

En este esquema cada subordinado de un empleado se representa mediante una sola tupla de la relación. Por tanto, si se tuviera un empleado con *id_empleado* 12-234 y los subordinados Martín y María, la relación *nombre_subordinado* tendría dos tuplas, (12-234, Martín) y (12-234, María).

Se crea una clave primaria del esquema de la relación consistente en todos los atributos del esquema. En el ejemplo anterior, la clave primaria consiste en todos los atributos de la relación *nombre_subordinado*.

Además, se crea una clave externa para el esquema de la relación, con el atributo generado a partir de la clave primaria del conjunto de entidades que hace referencia a la relación generada a partir del conjunto de entidades. En el ejemplo anterior, la restricción sería que el atributo *id_empleado* hace referencia a la relación *empleado*.

6.9.5 Representación de la generalización

Existen dos métodos diferentes para designar los esquemas de relación de los diagramas E-R que incluyen generalización. Aunque en esta discusión se hace referencia a la generalización de la Figura 6.20, se simplifica incluyendo sólo la primera capa de los conjuntos de entidades de nivel inferior—es decir, *empleado* y *cliente*. Se da por supuesto que *id_persona* es la clave primaria de *persona*.

1. Se crea un esquema para el conjunto de entidades de nivel superior. Para cada conjunto de entidades de nivel inferior se crea un esquema que incluye un atributo para cada uno de los atributos de ese conjunto de entidades más un atributo por cada atributo de la clave primaria del conjunto de entidades de nivel superior. Así, para el diagrama E-R de la Figura 6.20, se tienen tres esquemas:

   $$persona = (\underline{id_persona}, nombre, calle, ciudad)$$
   $$empleado = (\underline{id_persona}, sueldo)$$
   $$cliente = (\underline{id_persona}, calificación_crediticia)$$

   Los atributos de clave primaria del conjunto de entidades de nivel superior pasan a ser atributos de clave primaria del conjunto de entidades de nivel superior y de todos los conjuntos de entidades de nivel inferior. En el ejemplo anterior se pueden ver subrayados.

   Además, se crean restricciones de clave externa para los conjuntos de entidades de nivel inferior, con sus atributos de clave primaria que hacen referencia a la clave primaria de la relación creada a partir del conjunto de entidades de nivel superior. En el ejemplo anterior, el atributo *id_persona* de *empleado* haría referencia a la clave primaria de *persona*, y algo parecido puede decirse de *cliente*.

2. Es posible una representación alternativa si la generalización es disjunta y completa—es decir, si no hay ninguna entidad miembro de dos conjuntos de entidades de nivel inferior directamente por debajo de un conjunto de entidades de nivel superior, y si todas las entidades del conjunto de entidades de nivel superior también pertenece a uno de los conjuntos de entidades de nivel inferior. En este caso no se crea un esquema para el conjunto de entidades de nivel superior. En vez de eso, para cada conjunto de entidades de nivel inferior se crea un esquema que incluye un atributo por cada atributo de ese conjunto de entidades más un atributo por *cada* atributo del conjunto de entidades de nivel superior. Entonces, para el diagrama E-R de la Figura 6.20, se tienen dos esquemas:

empleado = (*id_persona*, nombre, calle, ciudad, sueldo)
cliente = (*id_persona*, nombre, calle, ciudad, calificación_crediticia)

Estos dos esquemas tiene *id_persona*, que es el atributo de clave primaria del conjunto de entidades de nivel superior *persona*, como clave primaria.

Un inconveniente del segundo método es la definición de las restricciones de clave externa. Para ilustrar el problema, supóngase que se tiene un conjunto de relaciones R que implica al conjunto de entidades *persona*. Con el primer método, al crear un esquema de relación R a partir del conjunto de relaciones, también se define una restricción de clave externa par R, que hace referencia al esquema *persona*. Desafortunadamente, con el segundo método no se tiene una única relación a la que pueda hacer referencia la restricción de clave externa de R. Para evitar este problema, hay que crear un esquema de relación *persona* que contenga, al menos, los atributos de clave primaria de la entidad *persona*.

Si se usara el segundo método para una generalización solapada, algunos valores se almacenarían varias veces, de manera innecesaria. Por ejemplo, si una persona es a la vez empleado y cliente, los valores de *calle* y de *ciudad* se almacenarían dos veces. Si la generalización no fuera completa—es decir, si alguna persona no fuera ni empleado ni cliente—entonces haría falta un esquema para representar a esas personas.

6.9.6 Representación de la agregación

El diseño de esquemas para los diagramas E-R que incluyen agregación es sencillo. Considérese el diagrama de la Figura 6.22. El esquema del conjunto de relaciones *dirige* entre la agregación de *trabaja_en* y el conjunto de entidades *director* incluye un atributo para cada atributo de las claves primarias del conjunto de entidades *director* y del conjunto de relaciones *trabaja_en*. También incluye un atributo para los atributos descriptivos, si los hay, del conjunto de relaciones *dirige*. Por tanto, se transforman los conjuntos de relaciones y de entidades de la entidad agregada siguiendo las reglas que se han definido anteriormente.

Las reglas que se han visto anteriormente para la creación de restricciones de clave primaria y de clave externa para los conjuntos de relaciones se pueden aplicar también a los conjuntos de relaciones que incluyen agregación, tratando la agregación como cualquier otra entidad. La clave primaria de la agregación es la clave primaria del conjunto de relaciones que la define. No hace falta ninguna relación más para que represente la agregación; en vez de eso, se usa la relación creada a partir de la relación definidora.

6.9.7 Esquemas relacionales para la entidad bancaria

En la Figura 6.25 se mostró el diagrama E-R de una entidad bancaria. El conjunto correspondiente de esquemas de relación, generado mediante las técnicas ya descritas en este apartado, se muestra a continuación. La clave primaria de cada esquema de relación se denota mediante el subrayado.

- Esquemas derivados de entidades fuertes:

  sucursal = (*nombre_sucursal*, *ciudad_sucursal*, *activos*)
  cliente = (*id_cliente*, *nombre_cliente*, *calle_cliente*, *ciudad_cliente*)
  préstamo = (*número_préstamo*, *importe*)
  cuenta = (*número_cuenta*, *saldo*)
  empleado = (*id_empleado*, *nombre_empleado*, *número_teléfono*, *fecha_contratación*)

- Esquemas derivados de atributos multivalorados (no se representan los atributos derivados). Se definen en una vista o en una función definida especialmente:

  nombre_subordinado = (*id_empleado*, *nombre_subordinado*)

- Esquemas derivados de conjuntos de relaciones que implican a conjuntos de entidades fuertes:

$$\begin{aligned}
&\textit{sucursal_cuenta} = (\underline{\textit{número_cuenta}}, \textit{nombre_sucursal})\\
&\textit{sucursal_préstamo} = (\underline{\textit{número_préstamo}}, \textit{nombre_sucursal})\\
&\textit{prestatario} = (\underline{\textit{id_cliente}, \textit{número_préstamo}})\\
&\textit{impositor} = (\underline{\textit{id_cliente}, \textit{número_cuenta}})\\
&\textit{asesor} = (\underline{\textit{id_cliente}, \textit{id_empleado}}, \textit{tipo})\\
&\textit{trabaja_para} = (\underline{\textit{id_empleado_trabajador}}, \textit{id_empleado_jefe})
\end{aligned}$$

- Esquemas derivados de conjuntos de entidades débiles (recuérdese que se probó en el Apartado 6.9.3.1 que el esquema *pago_préstamo* es redundante):

$$\textit{pago} = (\underline{\textit{número_préstamo}, \textit{número_pago}}, \textit{fecha_pago}, \textit{importe})$$

- Esquemas derivados de relaciones ES (se ha escogido la primera de las dos opciones presentadas en el Apartado 6.9.5 para permitir las cuentas que no son ni de ahorro ni corrientes):

$$\begin{aligned}
&\textit{cuenta_ahorro} = (\underline{\textit{número_cuenta}}, \textit{tasa_interés})\\
&\textit{cuenta_corriente} = (\underline{\textit{número_cuenta}}, \textit{importe_descubierto})
\end{aligned}$$

Se deja como ejercicio la creación de las restricciones de clave externa adecuadas para las relaciones anteriores.

6.10 Otros aspectos del diseño de bases de datos

La explicación sobre el diseño de esquemas dada en este capítulo puede crear la falsa impresión de que el diseño de esquemas es el único componente del diseño de bases de datos. En realidad, hay otras consideraciones que se tratarán con más profundidad en capítulos posteriores y que se describirán brevemente a continuación.

6.10.1 Restricciones de datos y diseño de bases de datos relacionales

Se ha visto gran variedad de restricciones de datos que pueden expresarse mediante SQL, como las restricciones de clave primaria, las de clave externa, las restricciones **check**, los asertos y los disparadores. Las restricciones tienen varios propósitos. El más evidente es la automatización de la conservación de la consistencia. Al expresar las restricciones en el lenguaje de definición de datos de SQL, el diseñador puede garantizar que el propio sistema de bases de datos haga que se cumplan las restricciones. Esto es más digno de confianza que dejar que cada programa haga cumplir las restricciones por su cuenta. También ofrece una ubicación central para la actualización de las restricciones y la adición de otras nuevas.

Otra ventaja de definir explícitamente las restricciones es que algunas restricciones resultan especialmente útiles en el diseño de esquemas de bases de datos relacionales. Si se sabe, por ejemplo, que el número de DNI identifica de manera unívoca a cada persona, se puede usar el número de DNI de una persona para vincular los datos relacionados con esa persona aunque aparezcan en varias relaciones. Compárese esta posibilidad, por ejemplo, con el color de ojos, que no es un identificador unívoco. El color de ojos no se puede usar para vincular los datos correspondientes a una persona concreta en varias relaciones, ya que los datos de esa persona no se podrían distinguir de los datos de otras personas con el mismo color de ojos.

En el Apartado 6.9 se generó un conjunto de esquemas de relación para un diseño E-R dado mediante las restricciones especificadas en el diseño. En el Capítulo 7 se formaliza esta idea y otras relacionadas y se muestra la manera en que puede ayudar al diseño de esquemas de bases de datos relacionales. El enfoque formal del diseño de bases de datos relacionales permite definir de manera precisa la bondad de cada diseño y mejorar los diseños malos. Se verá que el proceso de comenzar con un diseño entidad-relación y generar mediante algoritmos los esquemas de relación a partir de ese diseño es una buena manera de comenzar el proceso de diseño.

Las restricciones de datos también resultan útiles para determinar la estructura física de los datos. Puede resultar útil almacenar físicamente próximos en el disco los datos que están estrechamente relacionados entre sí, de modo que se mejore la eficiencia del acceso a disco. Algunas estructuras de índices funcionan mejor cuando el índice se crea sobre una clave primaria.

La aplicación de las restricciones se lleva a cabo a un precio potencialmente alto en rendimiento cada vez que se actualiza la base de datos. En cada actualización el sistema debe comprobar todas las restricciones y rechazar las actualizaciones que no las cumplen o ejecutar los disparadores correspondientes. La importancia de la penalización en rendimiento no sólo depende de la frecuencia de actualización, sino también del modo en que se haya diseñado la base de datos. En realidad, la eficiencia de la comprobación de determinados tipos de restricciones es un aspecto importante de la discusión del diseño de esquemas para bases de datos relacionales en el Capítulo 7.

6.10.2 Requisitos de uso: consultas y rendimiento

El rendimiento de los sistemas de bases de datos es un aspecto crítico de la mayor parte de los sistemas informáticos empresariales. El rendimiento no sólo tiene que ver con el uso eficiente del hardware de cálculo y de almacenamiento que se usa, sino también con la eficiencia de las personas que interactúan con el sistema y de los procesos que dependen de los datos de las bases de datos.

Existen dos métricas principales para el rendimiento.

- **Productividad**—el número de consultas o actualizaciones (a menudo denominadas *transacciones*) que pueden procesarse en promedio por unidad de tiempo.
- **Tiempo de respuesta**—el tiempo que tarda *una sola* transacción desde el comienzo hasta el final en promedio o en el peor de los casos.

Los sistemas que procesan gran número de transacciones agrupadas por lotes se centran en tener una productividad elevada. Los sistemas que interactúan con personas y los sistemas de tiempo crítico suelen centrarse en el tiempo de respuesta. Estas dos métricas no son equivalentes. La productividad elevada se consigue mediante un elevado uso de los componentes del sistema. Ello puede dar lugar a que algunas transacciones se pospongan hasta el momento en que puedan ejecutarse con mayor eficiencia. Las transacciones pospuestas sufren un bajo tiempo de respuesta.

Históricamente, la mayor parte de los sistemas de bases de datos comerciales se han centrado en la productividad; no obstante, gran variedad de aplicaciones, incluidas las aplicaciones basadas en la Web y los sistemas informáticos para telecomunicaciones necesitan un buen tiempo de respuesta promedio y una cota razonable para el peor tiempo de respuesta que pueden ofrecer.

La comprensión de los tipos de consultas que se espera que sean más frecuentes ayuda al proceso de diseño. Las consultas que implican reuniones necesitan evaluar más recursos que las que no las implican. A veces, cuando se necesita una reunión, puede que el administrador de la base de datos decida crear un índice que facilite la evaluación de la reunión. Para las consultas—tanto si está implicada una reunión como si no—se pueden crear índices para acelerar la evaluación de los predicados de selección (la cláusula **where** de SQL) que sea posible que aparezcan. Otro aspecto de las consultas que afecta a la elección de índices es la proporción relativa de operaciones de actualización y de lectura. Aunque los índices pueden acelerar las consultas, también ralentiza las actualizaciones, que se ven obligadas a realizar un trabajo adicional para mantener la exactitud de los índices.

6.10.3 Requisitos de autorización

Las restricciones de autorización también afectan al diseño de las bases de datos, ya que SQL permite que se autorice el acceso a los usuarios en función de los componentes del diseño lógico de la base de datos. Puede que haga falta descomponer un esquema de relación en dos o más esquemas para facilitar la concesión de derechos de acceso en SQL. Por ejemplo, un registro de empleados puede contener datos relativos a nóminas, funciones de los puestos y prestaciones sanitarias. Como diferentes unidades administrativas de la empresa pueden manejar cada uno de los diferentes tipos de datos, algunos usuarios necesitarán acceso a los datos de las nóminas, mientras se les deniega el acceso a los datos de las funciones de los puestos de trabajo, a los de las prestaciones sanitarias, etc. Si todos esos datos se hallan

en una tabla, la deseada división del acceso, aunque todavía posible mediante el uso de vistas, resulta más complicada. La división de los datos, de este modo, pasa a ser todavía más crítica cuando los datos se distribuyen en varios sistemas de una red informática, un aspecto que se considera en el Capítulo 22.

6.10.4 Flujos de datos y de trabajo

Las aplicaciones de bases de datos suelen formar parte de una aplicación empresarial de mayor tamaño que no sólo interactúa con el sistema de bases de datos, sino también con diferentes aplicaciones especializadas. Por ejemplo, en una compañía manufacturera, puede que un sistema de diseño asistido por computadora (computer-aided design, CAD) ayude al diseño de nuevos productos. Puede que el sistema CAD extraiga datos de la base de datos mediante instrucciones de SQL, procese internamente los datos, quizás interactuando con un diseñador de productos, y luego actualice la base de datos. Durante este proceso el control de los datos puede pasar a manos de varios diseñadores de productos y de otras personas. Como ejemplo adicional, considérese un informe de gastos de viaje. Lo crea un empleado que vuelve de un viaje de negocios (posiblemente mediante un paquete de software especial) y luego se envía al jefe de ese empleado, quizás a otros jefes de niveles superiores y, finalmente, al departamento de contabilidad para su pago (momento en el que interactúa con los sistemas informáticos de contabilidad de la empresa).

El término *flujo de trabajo* hace referencia a la combinación de datos y de tareas implicados en procesos como los de los ejemplos anteriores. Los flujos de trabajo interactúan con el sistema de bases de datos cuando se mueven entre los usuarios y los usuarios llevan a cabo sus tareas en el flujo de trabajo. Además de los datos sobre los que opera el flujo de trabajo, puede que la base de datos almacene datos sobre el propio flujo de trabajo, incluidas las tarea que lo conforman y la manera en que se han de hacer llegar a los usuarios. Por tanto, los flujos de trabajo especifican una serie de consultas y de actualizaciones de la base de datos que pueden tenerse en cuenta como parte del proceso de diseño de la base de datos. En otras palabras, el modelado de la empresa no sólo exige la comprensión de la semántica de los datos, sino también la de los procesos comerciales que los usan.

6.10.5 Otros problemas del diseño de bases de datos

El diseño de bases de datos no suele ser una actividad que se pueda dar por acabada. Las necesidades de las organizaciones evolucionan continuamente, y los datos que necesitan almacenar también evolucionan en consonancia. Durante las fases iniciales del diseño de la base de datos, o durante el desarrollo de las aplicaciones, puede que el diseñador de la base de datos se dé cuenta de que hacen falta cambios en el nivel del esquema conceptual, lógico o físico. Los cambios del esquema pueden afectar a todos los aspectos de la aplicación de bases de datos. Un buen diseño de bases de datos se anticipa a las necesidades futuras de la organización y el diseño se lleva a cabo de manera que se necesiten modificaciones mínimas a medida que evolucionen las necesidades de la organización.

Es importante distinguir entre las restricciones fundamentales y las que se anticipa que puedan cambiar. Por ejemplo, la restricción de que cada identificador de cliente identifique a un solo cliente es fundamental. Por otro lado, el banco puede tener la norma de que cada cliente sólo pueda tener una cuenta, lo que puede cambiar más adelante. Un diseño de la base de datos que sólo permita una cuenta por cliente necesitaría cambios importantes si el banco cambiase su normativa. Esos cambios no deben exigir modificaciones importantes en el diseño de la base de datos.

Además, es probable que la empresa a la que sirve la base de datos interactúe con otras empresas y, por tanto, puede que tengan que interactuar varias bases de datos. La conversión de los datos entre esquemas diferentes es un problema importante en las aplicaciones del mundo real. Se han propuesto diferentes soluciones para este problema. El modelo de datos XML, que se estudia en el Capítulo 10, se usa mucho para representar los datos cuando se intercambian entre diferentes aplicaciones.

Finalmente, merece la pena destacar que el diseño de bases de datos es una actividad orientada a los seres humanos en dos sentidos: los usuarios finales son personas (aunque se sitúe alguna aplicación entre la base de datos y los usuarios finales) y el diseñador de la base de datos debe interactuar intensamente con los expertos en el dominio de la aplicación para comprender los requisitos de datos de la aplicación. Todas las personas involucradas con los datos tiene necesidades y preferencias que se deben tener en cuenta para que el diseño y la implantación de la base de datos tengan éxito en la empresa.

6.11 El lenguaje de modelado unificado UML**

Los diagramas entidad-relación ayudan a modelar el componente de representación de datos de los sistemas de software. La representación de datos, sin embargo, sólo forma parte del diseño global del sistema. Otros componentes son los modelos de interacción del usuario con el sistema, la especificación de los módulos funcionales del sistema y su interacción, etc. El **lenguaje de modelado unificado** (Unified Modeling Language, UML) es una norma desarrollada bajo los auspicios del Grupo de Administración de Objetos (Object Management Group, OMG) para la creación de especificaciones de diferentes componentes de los sistemas de software. Algunas de las partes de UML son:

- **Diagramas de clase**. Los diagramas de clase son parecidos a los diagramas E-R. Más adelante en este apartado se mostrarán algunas características de los diagramas de clase y del modo en que se relacionan con los diagramas E-R.
- **Diagramas de caso de uso**. Los diagramas de caso de uso muestran la interacción entre los usuarios y el sistema, en especial los pasos de las tareas que llevan a cabo los usuarios (como retirar dinero o matricularse en una asignatura).
- **Diagramas de actividad**. Los diagramas de actividad describen el flujo de tareas entre los diferentes componentes del sistema.
- **Diagramas de implementación**. Los diagramas de implementación muestran los componentes del sistema y sus interconexiones, tanto en el nivel de los componentes de software como en el de hardware.

Aquí no se pretende ofrecer un tratamiento detallado de las diferentes partes del UML. Véanse las notas bibliográficas para encontrar referencias sobre UML. En vez de eso, se ilustrarán algunas características de la parte de UML que se relaciona con el modelado de datos mediante ejemplos.

La Figura 6.28 muestra varios constructores de diagramas E-R y sus constructores equivalentes de diagramas de clases de UML. Más adelante se describen estos constructores. UML muestra los conjuntos de entidades como cuadros y, a diferencia de E-R, muestra los atributos dentro de los cuadros en lugar de como elipses separadas. UML modela realmente objetos, mientras que E-R modela entidades. Los objetos son como entidades y tienen atributos, pero también proporcionan un conjunto de funciones (denominadas métodos) que se pueden invocar para calcular valores con base en los atributos de los objetos, o para actualizar el propio objeto. Los diagramas de clases pueden describir métodos además de atributos. Los objetos se tratan en el Capítulo 9.

Los conjuntos de relaciones binarias se representan en UML dibujando simplemente una línea que conecte los conjuntos de entidades. El nombre del conjunto de relaciones se escribe junto a la línea. También se puede especificar el rol que desempeña cada conjunto de entidades en un conjunto de relaciones escribiendo el nombre del rol sobre la línea, junto al conjunto de entidades. De manera alternativa, se puede escribir el nombre del conjunto de relaciones en un recuadro, junto con los atributos del conjunto de relaciones, y conectar el recuadro con una línea discontinua a la línea que describe el conjunto de relaciones. Este recuadro se puede tratar entonces como un conjunto de entidades, de la misma forma que la agregación en los diagramas E-R, y puede participar en relaciones con otros conjuntos de entidades.

Desde la versión 1.3 de UML, UML soporta las relaciones no binarias, usando la misma notación de rombos usada en los diagramas E-R. Las relaciones no binarias no se podían representar directamente en versiones anteriores de UML—había que convertirlas en relaciones binarias usando la técnica que se vio en el Apartado 6.5.3.

Las restricciones de cardinalidad se especifican en UML de la misma forma que en los diagramas E-R, de la forma $i..s$, donde i denota el número mínimo y s el máximo de relaciones en que puede participar cada entidad. Sin embargo, hay que ser consciente de que la ubicación de las restricciones es exactamente la contraria que en los diagramas E-R, como se muestra en la Figura 6.28. La restricción $0..*$ en el lado $E2$ y $0..1$ en el lado $E1$ significa que cada entidad $E2$ puede participar, a lo sumo, en una relación, mientras que cada entidad $E1$ puede participar en varias relaciones; en otras palabras, la relación es varios a uno de $E2$ a $E1$.

Los valores aislados como 1 o $*$ se pueden escribir en los arcos; el valor 1 sobre un arco se trata como equivalente de $1..1$, mientras que $*$ es equivalente a $0..*$.

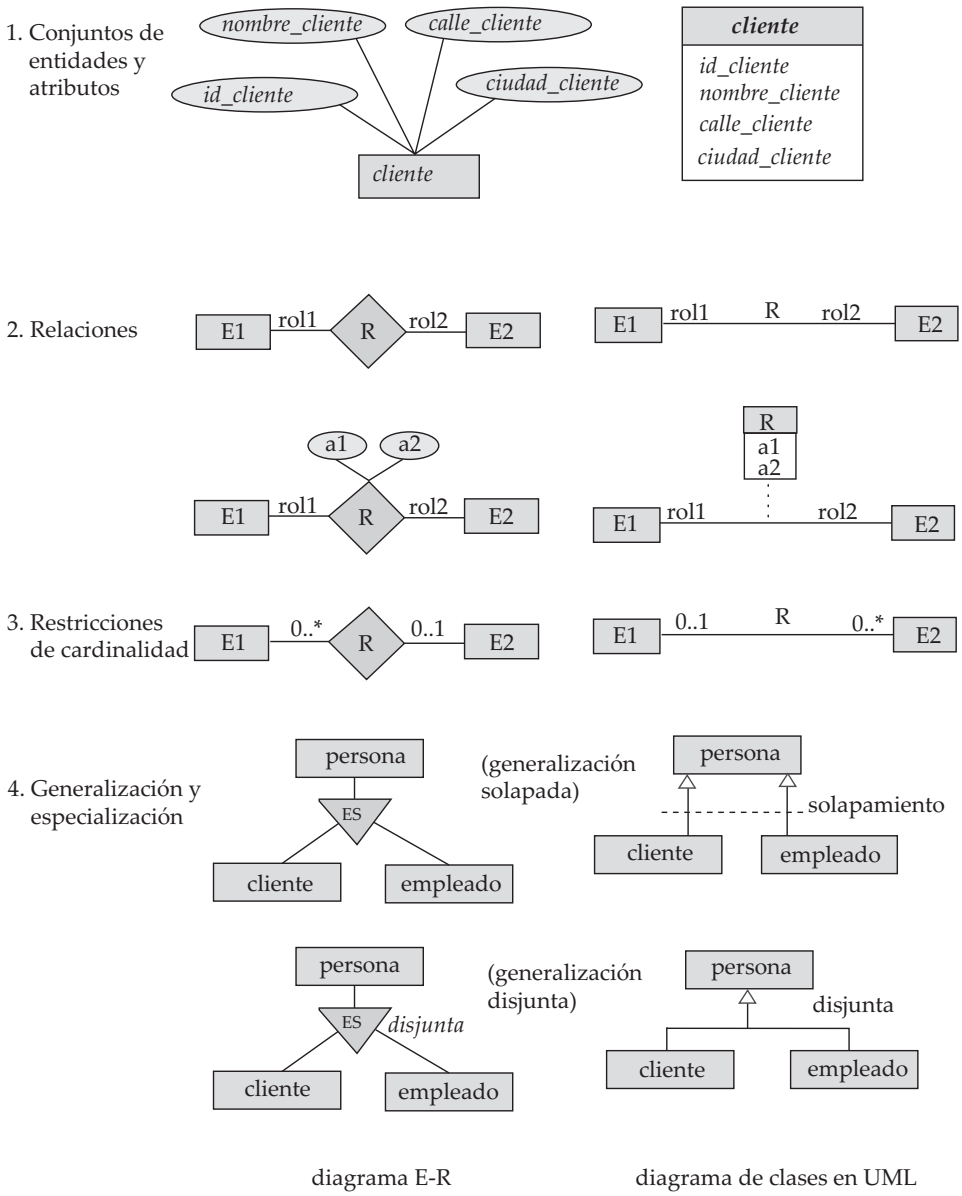


Figura 6.28 Símbolos usados en la notación de diagramas de clases de UML.

La generalización y la especialización se representan en UML conectando conjuntos de entidades mediante una línea con un triángulo al final correspondiente al conjunto de entidades más general. Por ejemplo, el conjunto de entidades *persona* es una generalización de *cliente* y de *empleado*. Los diagramas UML también pueden representar explícitamente las restricciones de la condición de disjunción y de solapamiento de las generalizaciones. La Figura 6.28 muestra generalizaciones disjuntas y solapadas de *cliente* y *empleado* a *persona*. Recuérdese que si la generalización de *cliente*/*empleado* a *persona* es disjunta, eso significa que nadie puede ser a la vez *cliente* y *empleado*. Una generalización solapada permite que una persona sea tanto *cliente* como *empleado*.

Los diagramas de clases de UML incluyen otras notaciones que no se corresponden con las notaciones E-R que se han visto. Por ejemplo, una línea entre dos conjuntos de entidades con un rombo en un extremo especifica que la entidad en el extremo del rombo contiene a la otra (la inclusión se denomina "agregación" en la terminología de UML). Por ejemplo, la entidad vehículo puede contener una entidad motor. Los diagramas de clases de UML también ofrecen notaciones para representar características del lenguaje orientadas a objetos, como las anotaciones públicas o privadas de los miembros de la clase e

interfaces (esto debe resultarle familiar a cualquiera que conozca los lenguajes Java o C#). Véanse las referencias en las notas bibliográficas para obtener más información sobre los diagramas de clases de UML.

6.12 Resumen

- El diseño de bases de datos supone principalmente el diseño del esquema de la base de datos. El modelo de datos **entidad-relación (E-R)** es un modelo de datos muy usado para el diseño de bases de datos. Ofrece una representación gráfica adecuada para ver los datos, las relaciones y las restricciones.
- El modelo está pensado principalmente para el proceso de diseño de la base de datos. Se desarrolló para facilitar el diseño de bases de datos al permitir la especificación de un **esquema de la empresa**. Este esquema representa la estructura lógica general de la base de datos. Esta estructura general se puede expresar gráficamente mediante un **diagrama E-R**.
- Una **entidad** es un objeto que existe en el mundo real y es distinguible de otros objetos. Esa distinción se expresa asociando a cada entidad un conjunto de atributos que describen el objeto.
- Una **relación** es una asociación entre diferentes entidades. Un **conjunto de relaciones** es una colección de entidades del mismo tipo, y un **conjunto de entidades** es una colección de entidades del mismo tipo.
- Una **superclave** de un conjunto de entidades es un conjunto de uno o más atributos que, tomados en conjunto, permiten identificar unívocamente una entidad del conjunto de entidades. Se elige una superclave mínima para cada conjunto de entidades de entre sus superclaves; la superclave mínima se denomina **clave primaria** del conjunto de entidades. Análogamente, un conjunto de relaciones es un conjunto de uno o más atributos que, tomados en conjunto, permiten identificar unívocamente una relación del conjunto de relaciones. De igual forma se elige una superclave mínima para cada conjunto de relaciones de entre todas sus superclaves; ésa es la clave primaria del conjunto de relaciones.
- La **correspondencia de cardinalidades** expresa el número de entidades con las que otra entidad se puede asociar mediante un conjunto de relaciones.
- Un conjunto de entidades que no tiene suficientes atributos para formar una clave primaria se denomina **conjunto de entidades débiles**. Un conjunto de entidades que tiene una clave primaria se denomina **conjunto de entidades fuertes**.
- La **especialización** y la **generalización** definen una relación de inclusión entre un conjunto de entidades de nivel superior y uno o más conjuntos de entidades de nivel inferior. La especialización es el resultado de tomar un subconjunto de un conjunto de entidades de nivel superior para formar un conjunto de entidades de inferior. La generalización es el resultado de tomar la unión de dos o más conjuntos disjuntos de entidades (de nivel inferior) para producir un conjunto de entidades de nivel superior. Los atributos de los conjuntos de entidades de nivel superior los heredan los conjuntos de entidades de nivel inferior.
- La **agregación** es una abstracción en la que los conjuntos de relaciones (junto con sus conjuntos de entidades asociados) se tratan como conjuntos de entidades de nivel superior y puede participar en las relaciones.
- Las diferentes características del modelo E-R ofrecen al diseñador de bases de datos numerosas opciones a la hora de representar lo mejor posible la empresa que se modela. Los conceptos y los objetos pueden, en ciertos casos, representarse mediante entidades, relaciones o atributos. Ciertos aspectos de la estructura global de la empresa se pueden describir mejor usando los conjuntos de entidades débiles, la generalización, la especialización o la agregación. A menudo, el diseñador debe sopesar las ventajas de un modelo simple y compacto frente a las de otro más preciso pero más complejo.

- El diseño de una base de datos especificado en un diagrama E-R se puede representar mediante un conjunto de esquemas de relación. Para cada conjunto de entidades y para cada conjunto de relaciones de la base de datos hay un solo esquema de relación al que se le asigna el nombre del conjunto de entidades o de relaciones correspondiente. Esto forma la base para la obtención del diseño de la base de datos relacional a partir del E-R.
- El **lenguaje de modelado unificado (UML)** ofrece un medio gráfico de modelar los diferentes componentes de los sistemas de software. El componente diagrama de clases de UML se basa en los diagramas E-R. Sin embargo, hay algunas diferencias entre los dos que se deben tener presentes.

Términos de repaso

- Modelo de datos entidad-relación.
- Entidad.
- Conjunto de entidades.
- Relación y conjunto de relaciones.
- Rol.
- Conjunto de relaciones recursivo.
- Atributos descriptivos.
- Conjunto de relaciones binarias.
- Grado de un conjunto de relaciones.
- Atributos.
- Dominio.
- Atributos simples y compuestos.
- Atributos monovalorados y multivalorados.
- Valor nulo.
- Atributo derivado.
- Superclave, clave candidata y clave primaria.
- Correspondencia de cardinalidad:
  - Relación uno a uno.
  - Relación uno a varios.
  - Relación varios a uno.
  - Relación varios a varios.
- Participación:
  - Total.
  - Parcial.
- Conjuntos de entidades débiles y fuertes.
  - Atributos discriminantes.
  - Relaciones identificadoras.
- Especialización y generalización.
  - Superclase y subclase.
  - Herencia de atributos.
  - Herencia simple y múltiple.
  - Pertenencia definida por condición y definida por el usuario.
  - Generalización disjunta y solapada.
- Restricción de completitud.
  - Generalización total y parcial.
- Agregación.
- Diagrama E-R.
- Lenguaje de modelado unificado (UML).

Ejercicios prácticos

6.1 Constrúyase un diagrama E-R para una compañía de seguros de coches cuyos clientes poseen uno o más coches cada uno. Cada coche tiene asociado un valor que va de cero al número de accidentes registrados.

6.2 La secretaría de una universidad conserva datos acerca de las siguientes entidades: (a) asignaturas, incluyendo el número, título, créditos, programa, y requisitos; (b) oferta de asignaturas, incluyendo el número de asignatura, año, semestre, número de sección, profesor(es), horario y aulas; (c) estudiantes, incluyendo identificador de estudiante, nombre y programa; y (d) profesores, incluyendo número de identificación, nombre, departamento y título. Además, la matrícula de los estudiantes en asignaturas y las notas concedidas a los estudiantes en cada asignatura en la que están matriculados se deben modelar adecuadamente.

Constrúyase un diagrama E-R para la secretaría. Documéntense todas las suposiciones que se hagan acerca de las restricciones de asignaciones.

6.3 Considérese una base de datos usada para registrar las notas que obtienen los estudiantes en diferentes exámenes de distintas asignaturas.

a. Constrúyase un diagrama E-R que modele los exámenes como entidades y utilice una relación ternaria para la base de datos.

b. Constrúyase un diagrama E-R alternativo que sólo utilice una relación binaria entre *estudiantes* y *asignaturas*. Hay que asegurarse de que sólo existe una relación entre cada pareja formada por un estudiante y una asignatura, pero se pueden representar las notas que obtiene cada estudiante en diferentes exámenes de una asignatura.

6.4 Diséñese un diagrama E-R para almacenar los logros de su equipo deportivo favorito. Se deben almacenar los partidos jugados, el resultado de cada partido, los jugadores de cada partido y las estadísticas de cada jugador en cada partido. Las estadísticas resumidas se deben representar como atributos derivados.

6.5 Considérese un diagrama E-R en el que el mismo conjunto de entidades aparezca varias veces. ¿Por qué permitir esa redundancia es una mala práctica que se debe evitar siempre que sea posible?

6.6 Considérese la base de datos de una universidad para la planificación de las aulas para los exámenes finales. Esta base de datos se puede modelar como el conjunto de entidades único *examen*, con los atributos *nombre_asignatura*, *número_sección*, *número_aula* y *hora*. Alternativamente se pueden definir uno o más conjuntos de entidades adicionales, junto con conjuntos de relaciones para sustituir algunos de los atributos del conjunto de entidades *examen*, como

- *asignatura* con atributos *nombre, departamento* y *número_a*
- *sección* con atributos *número_s* y *matriculados*, que es un conjunto de entidades débiles dependiente de *curso*.
- *aula* con atributos *número_a, capacidad* y *edificio*.

a. Muéstrese un diagrama E-R que ilustre el uso de los tres conjuntos de entidades adicionales citados.

b. Explíquense las características de la aplicación que influyen en la decisión de incluir o no incluir cada uno de los conjuntos de entidades adicionales.

6.7 Cuando se diseña el diagrama E-R para una empresa concreta se tienen varias alternativas para elegir.

a. ¿Qué criterio se debe considerar para tomar la decisión adecuada?

b. Diséñense tres diagramas E-R alternativos para representar la secretaría de la universidad del Ejercicio práctico 6.2. Menciónense las ventajas de cada uno. Arguméntese a favor de una de las alternativas.

6.8 Los diagramas E-R se pueden ver como grafos. ¿Qué significa lo siguientes en términos de la estructura del esquema de una empresa?

a. El grafo es inconexo.

b. El grafo es acíclico.

6.9 Considérese la representación de una relación ternaria mediante relaciones binarias como se describió en el Apartado 6.5.3 e ilustrado en la Figura 6.29 (no se muestran los atributos).

a. Muéstrese un ejemplar simple de E, A, B, C, R_A, R_B y R_C que no pueda corresponder a ningún ejemplar de A, B, C y R.

b. Modifíquese el diagrama E-R de la Figura 6.29b para introducir restricciones que garanticen que cualquier ejemplar E, A, B, C, R_A, R_B y R_C que satisfaga las restricciones corresponda a algún ejemplar de A, B, C y R.

c. Modifíquese la traducción anterior para manejar las restricciones de participación total sobre las relaciones ternarias.

d. La representación anterior exige que se cree un atributo de clave primaria para E. Muéstrese la manera de tratar E como un conjunto de entidades débiles de forma que no haga falta ningún atributo de clave primaria.

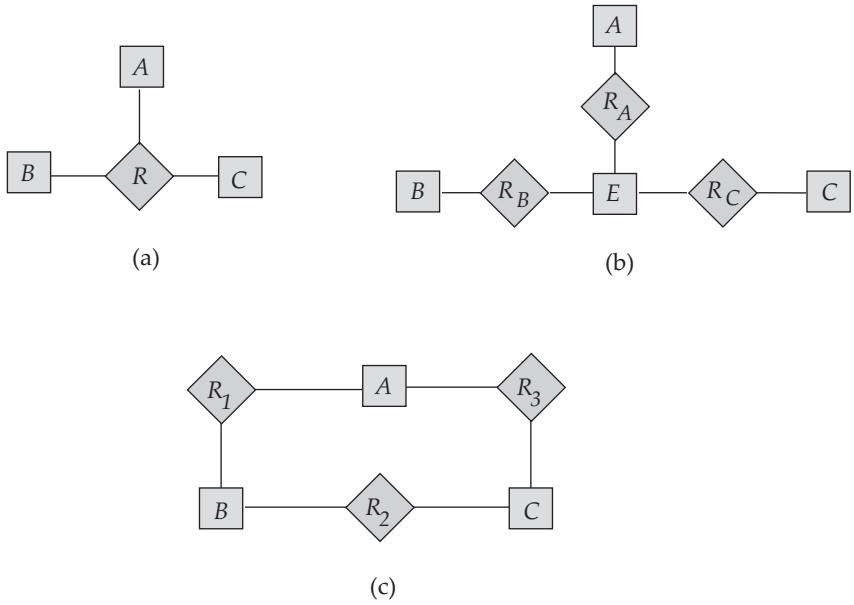


Figura 6.29 Diagrama E-R para el Ejercicio práctico 6.9 y el Ejercicio 6.22.

6.10 Los conjuntos de entidades débiles se pueden convertir en conjuntos de entidades fuertes simplemente añadiendo a sus atributos los atributos de clave primaria del conjunto de entidades identificadoras. Descríbase el tipo de redundancia que se produce al hacerlo.

6.11 La Figura 6.30 muestra una estructura reticular de generalización y especialización (no se muestran los atributos). Para los conjuntos de entidades A, B y C explíquese cómo se heredan los atributos desde los conjuntos de entidades de nivel superior X e Y. Explíquese la manera de manejar el caso de que un atributo de X tenga el mismo nombre que algún atributo de Y.

6.12 Considérense dos bancos diferentes que deciden fusionarse. Supóngase que ambos bancos usan exactamente el mismo esquema de bases de datos E-R—el de la Figura 6.25. (Evidentemente, esta suposición es muy poco realista; se considera un caso más realista en el Apartado 22.8). Si el banco fusionado va a tener una sola base de datos, hay varios problemas posibles:

- La posibilidad de que los dos bancos originales tengan sucursales con el mismo nombre.
- La posibilidad de que algunos clientes lo sean de los dos bancos originales.
- La posibilidad de que algún número de préstamo o de cuenta se usara en los dos bancos originales (para diferentes préstamos o cuentas, por supuesto).

Para cada uno de estos posibles problemas descríbase el motivo de que existan de hecho la posibilidad de dificultades. Propóngase una solución al problema. Para la solución propuesta, explíquese cualquier cambio que haya que hacer y descríbase el efecto que tendrá en el esquema y en los datos.

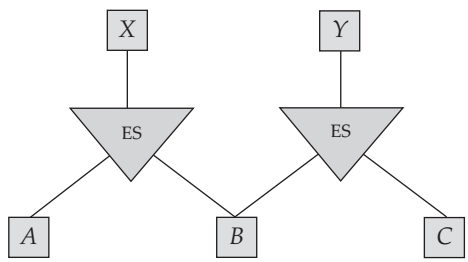


Figura 6.30 Diagrama E-R para el Ejercicio práctico 6.11.

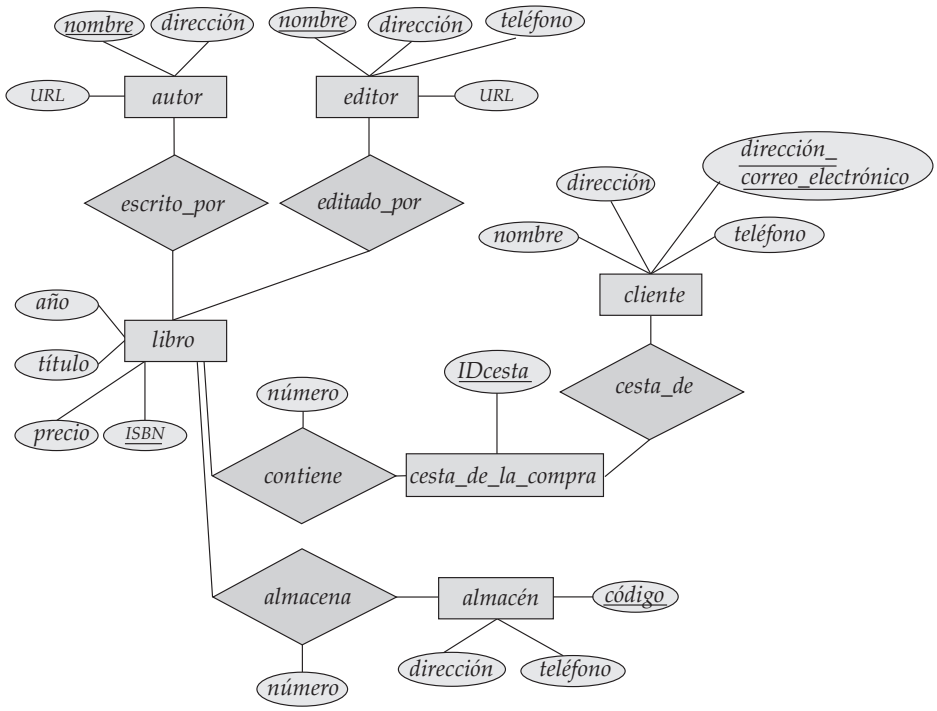


Figura 6.31 Diagrama E-R para el Ejercicio 6.21.

6.13 Reconsidérese la situación descrita en el Ejercicio práctico 6.12 bajo la suposición de que un banco está en España y el otro en Portugal. Como antes, los bancos usan el esquema de la Figura 6.25, excepto que el banco portugués usa el número de DNI portugués, mientras que el banco español usa el DNI español para la identificación de los clientes. ¿Qué problemas (además de los identificados en el Ejercicio práctico 6.12) pueden darse en este caso multinacional? ¿Cómo se pueden resolver? Asegúrese de tomar en consideración tanto el esquema como los datos reales al elaborar la respuesta.

Ejercicios

6.14 Explíquense las diferencias entre los términos clave primaria, clave candidata y superclave.

6.15 Constrúyase un diagrama E-R para un hospital con un conjunto de pacientes y un conjunto de médicos. Asóciese con cada paciente un registro de las diferentes pruebas y exámenes realizados.

6.16 Constrúyanse tablas adecuadas para cada uno de los diagramas E-R de los Ejercicios prácticos 6.1 y 6.2.

6.17 Extiéndase el diagrama E-R del Ejercicio práctico 6.4 para que almacene la misma información para todos los equipos de una liga.

6.18 Explíquense las diferencias entre los conjuntos de entidades débiles y los conjuntos de entidades fuertes.

6.19 Se puede convertir cualquier conjunto de entidades débiles en un conjunto de entidades fuertes simplemente añadiendo los atributos apropiados. ¿Por qué, entonces, se tienen conjuntos de entidades débiles?

6.20 Defínase el concepto de agregación. Propónganse dos ejemplos en los que este concepto sea útil.

6.21 Considérese el diagrama E-R de la Figura 6.31, que modela una librería en línea.

a. Cítense los conjuntos de entidades y sus claves primarias.

b. Supóngase que la librería añade casetes de música y discos compactos a su colección. El mismo elemento musical puede estar presente en formato de casete o de disco compacto, con diferentes precios. Extiéndase el diagrama E-R para modelar esta adición, ignorando el efecto sobre las cestas de la compra.

c. Extiéndase ahora el diagrama E-R mediante la generalización para modelar el caso en que una cesta de la compra pueda contener cualquier combinación de libros, casetes de música o discos compactos.

6.22 En el Apartado 6.5.3 se representó una relación ternaria (que se repite en la Figura 6.29a) mediante relaciones binarias, como se muestra en la Figura 6.29b. Considérese la alternativa mostrada en la Figura 6.29c. Explíquense las ventajas relativas de estas dos representaciones alternativas de una relación ternaria mediante relaciones binarias.

6.23 Considérense los esquemas de relación mostrados en el Apartado 6.9.7, que se generaron a partir del diagrama E-R de la Figura 6.25. Para cada esquema, especifíquense las restricciones de clave externa que hay que crear, si es que hay que crear alguna.

6.24 Diséñese una jerarquía de especialización–generalización para una compañía de venta de vehículos de motor. La compañía vende motocicletas, coches, furgonetas y autobuses. Justifíquese la ubicación de los atributos en cada nivel de la jerarquía. Explíquese el motivo de que no se deban ubicar en un nivel superior o inferior.

6.25 Explíquese la diferencia entre las restricciones definidas por condición y las definidas por el usuario. ¿Cuáles de estas restricciones pueden comprobar el sistema automáticamente? Explíquese la respuesta.

6.26 Explíquese la diferencia entre las restricciones disjuntas y las solapadas.

6.27 Explíquese la diferencia entre las restricciones totales y las parciales.

6.28 Dibújense los equivalentes de UML de los diagramas E-R de las Figuras 6.8c, 6.9, 6.11, 6.12 y 6.20.

Notas bibliográficas

El modelo de datos E-R fue introducido por Chen [1976]. Teorey et al. [1986] presentó una metodología de diseño lógico para las bases de datos relacionales que usa el modelo E-R extendido. La norma de Definición de la integración para el modelado de información (IDEF1X, Integration Definition for Information Modeling) IDEF1X [1993] publicado por el Instituto Nacional de Estados Unidos para Normas y Tecnología (United States National Institute of Standards and Technology, NIST) definió las normas para los diagramas E-R. No obstante, hoy en día se usa gran variedad de notaciones E-R.

Thalheim [2000] proporciona un tratamiento detallado pero propio de libro de texto de la investigación en el modelado E-R. En los libros de texto Batini et al. [1992] y Elmasri y Navathe [2003] se ofrecen explicaciones básicas. Davis et al. [1983] proporciona una colección de artículos sobre el modelo E-R.

En 2004 la versión más reciente de UML era la 1.5, con la versión 2.0 de UML a punto de adoptarse. Véase www.uml.org para obtener más información sobre las normas y las herramientas de UML.

Herramientas

Muchos sistemas de bases de datos ofrecen herramientas para el diseño de bases de datos que soportan los diagramas E-R. Estas herramientas ayudan al diseñador a crear diagramas E-R y pueden crear automáticamente las tablas correspondientes de la base de datos. Véanse las notas bibliográficas del Capítulo 1 para consultar referencias de sitios Web de fabricantes de bases de datos. También hay algunas herramientas de modelado de datos independientes de las bases de datos que soportan los diagramas E-R y los diagramas de clases de UML. Entre ellas están Rational Rose (www.rational.com/products/rose), Microsoft Visio (véase www.microsoft.com/office/visio), ERwin (búsquese ERwin en el sitio www.cai.com/products), Poseidon para UML (www.gentleware.com) y SmartDraw (www.smartdraw.com).

CAPÍTULO 7

Diseño de bases de datos relacionales

En este capítulo se considera el problema de diseñar el esquema de una base de datos relacional. Muchos de los problemas que conlleva son parecidos a los de diseño que se han considerado en el Capítulo 6 en relación con el modelo E-R.

En general, el objetivo del diseño de una base de datos relacional es la generación de un conjunto de esquemas de relación que permita almacenar la información sin redundancias innecesarias, pero que también permita recuperarla fácilmente. Esto se consigue mediante el diseño de esquemas que se hallen en la *forma normal* adecuada. Para determinar si el esquema de una relación se halla en una de las formas normales deseables es necesario obtener información sobre la empresa real que se está modelando con la base de datos. Parte de esa información se halla en un diagrama E-R bien diseñado, pero puede ser necesaria información adicional sobre la empresa.

En este capítulo se introduce un enfoque formal al diseño de bases de datos relacionales basado en el concepto de dependencia funcional. Posteriormente se definen las formas normales en términos de las dependencias funcionales y de otros tipos de dependencias de datos. En primer lugar, sin embargo, se examina el problema del diseño relacional desde el punto de vista de los esquemas derivados de un diseño entidad-relación dado.

7.1 Características de los buenos diseños relacionales

El estudio del diseño entidad-relación llevado a cabo en el Capítulo 6 ofrece un excelente punto de partida para el diseño de bases de datos relacionales. Ya se vio en el Apartado 6.9 que es posible generar directamente un conjunto de esquemas de relación a partir del diseño E-R. Evidentemente, la adecuación (o no) del conjunto de esquemas resultante depende, en primer lugar, de la calidad del diseño E-R. Más adelante en este capítulo se estudiarán maneras precisas de evaluar la adecuación de los conjuntos de esquemas de relación. No obstante, es posible llegar a un buen diseño empleando conceptos que ya se han estudiado.

Para facilitar las referencias, en la Figura 7.1 se repiten los esquemas del Apartado 6.9.7.

7.1.1 Alternativa de diseño: esquemas grandes

A continuación se examinan las características de este diseño de base de datos relacional y algunas alternativas. Supóngase que, en lugar de los esquemas *prestatario* y *préstamo*, se considerase el esquema:

$$prestatario_préstamo = (id_cliente, número_préstamo, importe)$$

Esto representa el resultado de la reunión natural de las relaciones correspondientes a *prestatario* y a *préstamo*. Parece una buena idea, ya que es posible expresar algunas consultas empleando menos reuniones, hasta que se considera detenidamente la entidad bancaria que condujo al diseño E-R. Obsérvese

sucursal = (*nombre_sucursal*, *ciudad_sucursal*, *activos*)
cliente = (*id_cliente*, *nombre_cliente*, *calle_cliente*, *ciudad_cliente*)
préstamo = (*número_préstamo*, *importe*)
cuenta = (*número_cuenta*, *saldo*)
empleado = (*id_empleado*, *nombre_empleado*, *número_teléfono*, *fecha_contratación*)
nombre_subordinado = (*id_empleado*, *nombre_subordinado*)
sucursal_cuenta = (*número_cuenta*, *nombre_sucursal*)
sucursal_préstamo = (*número_préstamo*, *nombre_sucursal*)
prestatario = (*id_cliente*, *número_préstamo*)
impositor = (*id_cliente*, *número_cuenta*)
asesor = (*id_cliente*, *id_empleado*, *tipo*)
trabaja_para = (*id_empleado_trabajador*, *id_empleado_jefe*)
pago = (*número_préstamo*, *número_pago*, *fecha_pago*, *importe_pago*)
cuenta_ahorro = (*número_cuenta*, *tasa_interés*)
cuenta_corriente = (*número_cuenta*, *importe_descubierto*)

Figura 7.1 Los esquemas bancarios del Apartado 6.9.7.

que el conjunto de relaciones *prestatario* es varios a varios. Esto permite que cada cliente tenga varios préstamos y, también, que se puedan conceder préstamos conjuntamente a varios clientes. Se tomó esta decisión para que fuese posible representar los préstamos realizados conjuntamente a matrimonios o a consorcios de personas (que pueden participar en una aventura empresarial conjunta). Ése es el motivo de que la clave primaria del esquema *prestatario* consista en *id_cliente* y *número_préstamo* en lugar de estar formada sólo por *número_préstamo*.

Considérese un préstamo que se concede a uno de esos consorcios y considérense las tuplas que deben hallarse en la relación del esquema *prestatario_préstamo*. Supóngase que el préstamo P-100 se concede al consorcio formado por los clientes Santiago (con identificador de cliente 23652), Antonio (con identificador de cliente 15202) y Jorge (con identificador de cliente 23521) por un importe de 10.000 €.

La Figura 7.2 muestra la forma en que es posible representar esta situación empleando *préstamo* y *prestatario* y con el diseño alternativo usando *prestatario_préstamo*. La tupla (P-100, 10.000) de la relación del esquema *préstamo* se reúne con tres tuplas de la relación del esquema *prestatario*, lo que genera tres tuplas de la relación del esquema *prestatario_préstamo*. Obsérvese que en *prestatario_préstamo* es necesario repetir una vez el importe del préstamo por cada cliente integrante del consorcio que solicita el préstamo. Es importante que todas estas tuplas concuerden en lo relativo al importe del préstamo P-100, ya que, en caso contrario, la base de datos quedaría en un estado inconsistente. En el diseño original que utiliza *préstamo* y *prestatario* se almacenaba el importe de cada préstamo exactamente una vez. Esto sugiere que el empleo de *prestatario_préstamo* no es buena idea, ya que se almacena el importe de cada préstamo de manera redundante y se corre el riesgo de que algún usuario actualice el importe del préstamo en una de las tuplas pero no en todas, y, por tanto, genere inconsistencia.

Considérese ahora otra alternativa. Sea *préstamo_importe_sucursal* = (*número_préstamo*, *importe*, *nombre_sucursal*), creada a partir de *préstamo* y de *sucursal_préstamo* (mediante una reunión de las relaciones correspondientes). Esto se parece al ejemplo que se acaba de considerar, pero con una importante diferencia. En este caso, *número_préstamo* es la clave primaria de los dos esquemas, *sucursal_préstamo* y *préstamo* y, por tanto, también lo es de *préstamo_importe_sucursal*. Esto se debe a que el conjunto de relaciones *sucursal_préstamo* está definido como varios a uno, a diferencia del conjunto de relaciones *prestatario* del ejemplo anterior. Para cada préstamo sólo hay una sucursal asociada. Por tanto, cada número de préstamo concreto sólo aparece una vez en *sucursal_préstamo*. Supóngase que el préstamo P-100 está asociado con la sucursal de Soria. La Figura 7.3 muestra la manera en que esto se representa empleando *préstamo_importe_sucursal*. La tupla (P-100, 10.000) de la relación del esquema *préstamo* se reúne con una sola tupla de la relación del esquema *sucursal_préstamo*, lo que sólo genera una tupla de la relación del esquema *préstamo_importe_sucursal*. No hay repetición de información en *préstamo_importe_sucursal* y, por tanto, se evitan los problemas que se encontraron en el ejemplo anterior.

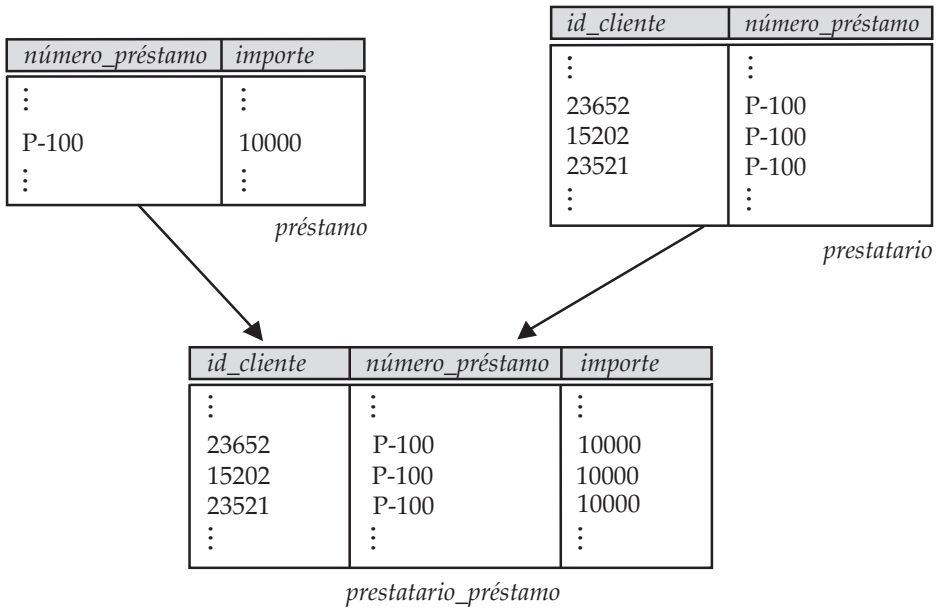


Figura 7.2 Lista parcial de las tuplas de las relaciones *préstamo*, *prestatario* y *prestatario_préstamo*.

Antes de acceder finalmente al empleo de *préstamo_importe_sucursal* en lugar de *sucursal_préstamo* y de *préstamo*, hay otro aspecto que se debe considerar. Puede que se desee registrar en la base de datos un préstamo y su sucursal asociada antes de que se haya determinado su importe. En el diseño antiguo, el esquema *sucursal_préstamo* puede hacerlo pero, con el diseño revisado *préstamo_importe_sucursal*, habría que crear una tupla con un valor nulo para *importe*. En algunos casos los valores nulos pueden crear problemas, como se vio en el estudio de SQL. No obstante, si se decide que en este caso no suponen ningún problema, se puede aceptar el empleo del diseño revisado.

Los dos ejemplos que se acaban de examinar muestran la importancia de la naturaleza de las claves primarias al decidir si las combinaciones de esquemas tienen sentido. Se han presentado problemas —concretamente, la repetición de información—cuando el atributo de reunión (*número_préstamo*) no era la clave primaria de *los dos* esquemas que se combinaban.

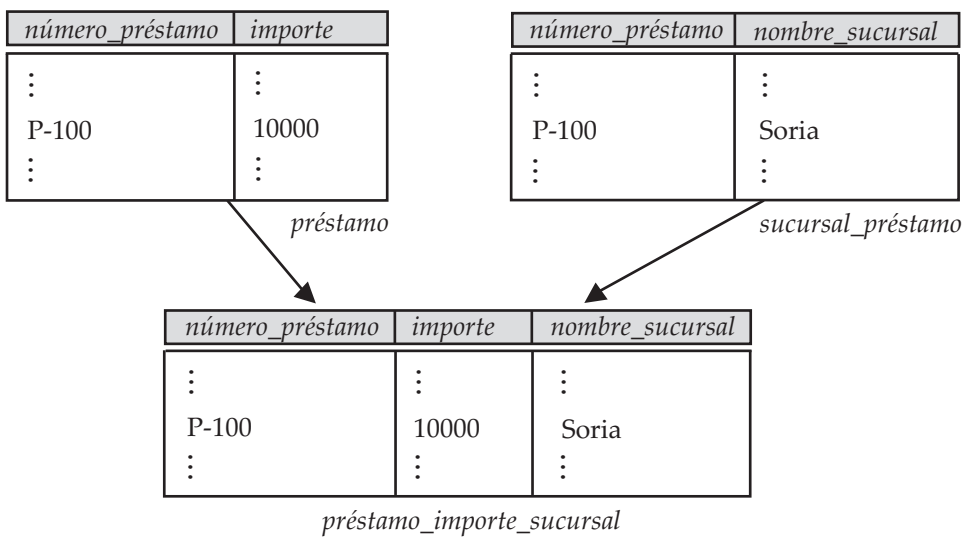


Figura 7.3 Lista parcial de las tuplas de las relaciones *préstamo*, *sucursal_préstamo* y *préstamo_importe_sucursal*.

7.1.2 Alternativa de diseño: esquemas pequeños

Supóngase nuevamente que, de algún modo, se ha comenzado a trabajar con el esquema *prestatario_préstamo*. ¿Cómo se reconoce que necesita que haya repetición de la información y que hay que dividirlo en dos esquemas, *prestatario* y *préstamo*? Como no se dispone de los esquemas *prestatario* y *préstamo*, se carece de la información sobre la clave primaria que se empleó para describir el problema de *prestatario_préstamo*.

Mediante la observación del contenido de las relaciones reales del esquema *prestatario_préstamo* se puede percibir la repetición de información resultante de tener que relacionar el importe del préstamo una vez por cada prestatario asociado con cada uno de ellos. Sin embargo, se trata de un proceso en el que no se puede confiar. Las bases de datos reales tienen gran número de esquemas y un número todavía mayor de atributos. El número de tuplas puede ser del orden de millones. Descubrir la repetición puede resultar costoso. Hay un problema más fundamental todavía en este enfoque. No permite determinar si la carencia de repeticiones es meramente un caso especial "afortunado" o la manifestación de una regla general. En el ejemplo anterior, ¿cómo se *sabe* que en la entidad bancaria cada préstamo (identificado por su número correspondiente) sólo *debe* tener un importe? ¿El hecho de que el préstamo P-100 aparezca tres veces con el mismo importe es sólo una coincidencia? Estas preguntas no se pueden responder sin volver a la propia empresa y comprender sus reglas de funcionamiento. En concreto, hay que averiguar si el banco exige que cada préstamo (identificado por su número correspondiente) *sólo* tenga un importe.

En el caso de *prestatario_préstamo*, el proceso de creación del diseño E-R logró evitar la creación de ese esquema. Sin embargo, esta situación fortuita no se produce siempre. Por tanto, hay que permitir que el diseñador de la base de datos especifique normas como "cada valor de *número_préstamo* corresponde, como máximo, a un *importe*", incluso en los casos en los que *número_préstamo* no sea la clave primaria del esquema en cuestión. En otras palabras, hay que escribir una norma que diga "si hay un esquema (*número_préstamo, importe*), entonces *número_préstamo* puede hacer de clave primaria". Esta regla se especifica como la **dependencia funcional** *número_préstamo* → *importe*. Dada esa regla, ya se tiene suficiente información para reconocer el problema del esquema *prestatario_préstamo*. Como *número_préstamo* no puede ser la clave primaria de *prestatario_préstamo* (porque puede que cada préstamo necesite varias tuplas de la relación del esquema *prestatario_préstamo*), tal vez haya que repetir el importe del préstamo.

Observaciones como éstas y las reglas (especialmente las dependencias funcionales) que generan permiten al diseñador de la base de datos reconocer las situaciones en que hay que dividir, o descomponer, un esquema en dos o más. No es difícil comprender que la manera correcta de descomponer *prestatario_préstamo* es dividirlo en los esquemas *prestatario* y *préstamo*, como en el diseño original. Hallar la descomposición correcta es mucho más difícil para esquemas con gran número de atributos y varias dependencias funcionales. Para trabajar con ellos hay que confiar en una metodología formal que se desarrolla más avanzado este capítulo.

No todas las descomposiciones de los esquemas resultan útiles. Considérese el caso extremo en que todo lo que tengamos sean esquemas con un solo atributo. No se puede expresar ninguna relación interesante de ningún tipo. Considérese ahora un caso menos extremo en el que se decida descomponer el esquema *empleado* en

$$empleado1 = (\underline{id_empleado}, nombre_empleado)$$
$$empleado2 = (nombre_empleado, número_teléfono, fecha_contratación)$$

El problema con esta descomposición surge de la posibilidad de que la empresa tenga dos empleados que se llamen igual. Esto no es improbable en la práctica, ya que muchas culturas tienen algunos nombres muy populares y, además, puede que los niños se llamen como sus padres. Por supuesto, cada persona tiene un identificador de empleado único, que es el motivo de que se pueda utilizar *id_empleado* como clave primaria. A modo de ejemplo, supóngase que dos empleados llamados Koldo trabajan para el mismo banco y tienen las tuplas siguientes de la relación del esquema *empleado* del diseño original:

(1234567890, Koldo, 918820000, 29/03/1984)
(9876543210, Koldo, 918699999, 16/01/1981)

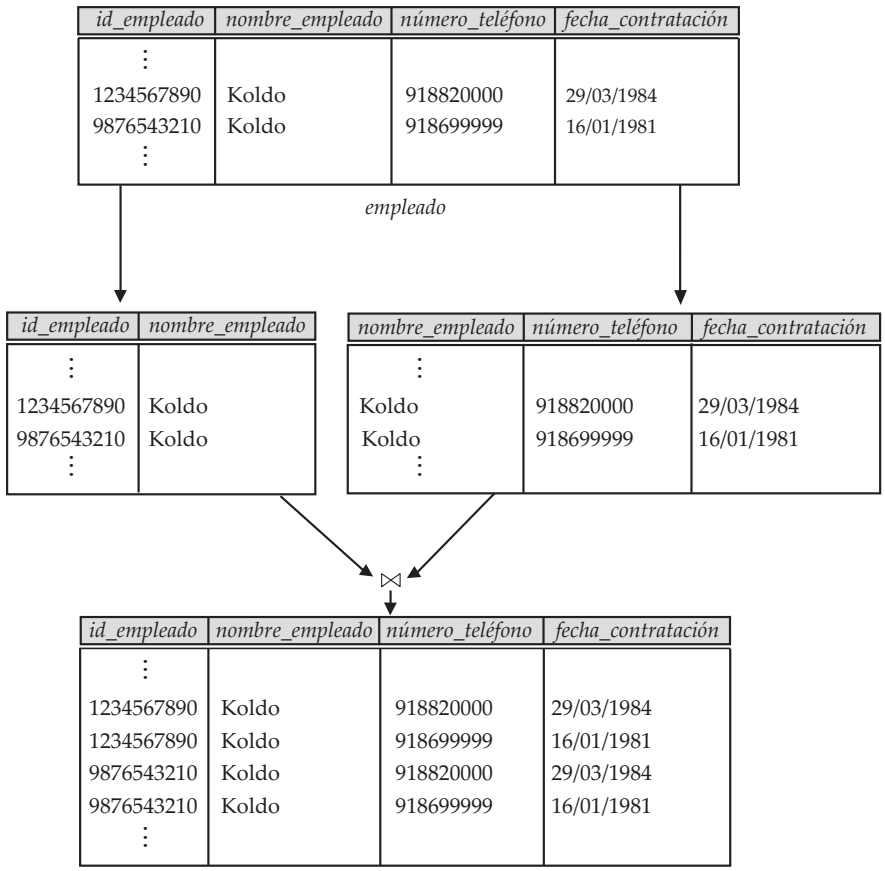


Figura 7.4 Pérdida de información debida a una mala descomposición.

La Figura 7.4 muestra estas tuplas, las tuplas resultantes al utilizar los esquemas procedentes de la descomposición y el resultado que se obtendría si se intentara volver a generar las tuplas originales mediante una reunión natural. Como se ve en la figura, las dos tuplas originales aparecen en el resultado junto con dos tuplas nuevas que mezclan de manera incorrecta valores de fechas correspondientes a los dos empleados llamados Koldo. Aunque se dispone de más tuplas, se tiene en realidad menos información en el sentido siguiente: se puede indicar que un determinado número de teléfono y una fecha de contratación dada corresponden a alguien llamado Koldo, pero no se puede distinguir a cuál de ellos. Por tanto, la descomposición propuesta es incapaz de representar algunos datos importantes de la entidad bancaria. Evidentemente, es preferible evitar este tipo de descomposiciones. Estas descomposiciones se denominan **descomposiciones con pérdidas**, y, a la inversa, aquéllas que no tienen pérdidas se denominan **descomposiciones sin pérdidas**.

7.2 Dominios atómicos y la primera forma normal

El modelo E-R permite que los conjuntos de entidades y los de relaciones tengan atributos con algún tipo de subestructura. Concretamente, permite atributos multivalorados, como *nombre_subordinado* de la Figura 6.25, y atributos compuestos (como el atributo *dirección* con los atributos componentes *calle* y *ciudad*). Cuando se crean tablas a partir de los diseños E-R que contienen ese tipo de atributos, se elimina esa subestructura. Para los atributos compuestos, se deja que cada atributo componente sea un atributo de pleno derecho. Para los atributos multivalorados, se crea una tupla por cada elemento del conjunto multivalorado.

En el modelo relacional se formaliza esta idea de que los atributos no tienen subestructuras. Un dominio es **atómico** si se considera que los elementos de ese dominio son unidades indivisibles. Se dice que

el esquema de la relación *R* está en la **primera forma normal** (1FN) si los dominios de todos los atributos de *R* son atómicos.

Los conjuntos de nombres son ejemplos de valores no atómicos. Por ejemplo, si el esquema de la relación *empleado* incluyera el atributo *hijos*, los elementos de cuyo dominio son conjuntos de nombres, el esquema no estaría en la primera forma normal.

Los atributos compuestos, como el atributo *dirección* con los atributos componentes *calle* y *ciudad*, tienen también dominios no atómicos.

Se da por supuesto que los números enteros son atómicos, por lo que el conjunto de los números enteros es un dominio atómico; el conjunto de todos los conjuntos de números enteros es un dominio no atómico. La diferencia estriba en que normalmente no se considera que los enteros tengan subpartes, pero sí se considera que las tienen los conjuntos de enteros—es decir, los enteros que componen el conjunto. Pero lo importante no es lo que sea el propio dominio, sino el modo en que se utilizan los elementos de ese dominio en la base de datos. El dominio de todos los enteros sería no atómico si se considerara que cada entero es una lista ordenada de cifras.

Como ilustración práctica del punto anterior, considérese una organización que asigna a los empleados números de identificación de la manera siguiente: las dos primeras letras especifican el departamento y las cuatro cifras restantes son un número único para cada empleado dentro de ese departamento. Ejemplos de estos números pueden ser $CS0012$ y $EE1127$. Estos números de identificación pueden dividirse en unidades menores y, por tanto, no son atómicos. Si el esquema de la relación tuviera un atributo cuyo dominio consistiera en números de identificación así codificados, el esquema no se hallaría en la primera forma normal.

Cuando se utilizan números de identificación de este tipo, se puede averiguar el departamento de cada empleado escribiendo código que analice la estructura de los números de identificación. Ello exige programación adicional, y la información queda codificada en el programa de aplicación en vez de en la base de datos. Surgen nuevos problemas si se utilizan esos números de identificación como claves primarias: cada vez que un empleado cambie de departamento, habrá que modificar su número de identificación en todos los lugares en que aparezca, lo que puede constituir una tarea difícil; en caso contrario, el código que interpreta ese número dará un resultado erróneo.

El empleo de atributos con el valor definido por un conjunto puede llevar a diseños con almacenamiento de datos redundante; lo que, a su vez, puede dar lugar a inconsistencias. Por ejemplo, en lugar de representar la relación entre las cuentas y los clientes como la relación independiente *impositor*, puede que el diseñador de bases de datos esté tentado a almacenar un conjunto de *titulares* con cada cuenta, y un conjunto de *cuentas* con cada cliente. Siempre que se cree una cuenta, o se actualice el conjunto de titulares de una cuenta, habrá que llevar a cabo la actualización en dos lugares; no llevar a cabo esas dos actualizaciones puede dejar la base de datos en un estado inconsistente. Guardar sólo uno de esos conjuntos evitaría la información repetida, pero complicaría algunas consultas.

Algunos tipos de valores no atómicos pueden resultar útiles, aunque deben utilizarse con cuidado. Por ejemplo, los atributos con valores compuestos suelen resultar útiles, y los atributos con el valor determinado por un conjunto también resultan útiles en muchos casos, que es el motivo por el que el modelo E-R los soporta. En muchos dominios en los que las entidades tienen una estructura compleja, la imposición de la representación en la primera forma normal supone una carga innecesaria para el programador de las aplicaciones, que tiene que escribir código para convertir los datos a su forma atómica. También hay sobrecarga en tiempo de ejecución por la conversión de los datos entre su forma habitual y su forma atómica. Por tanto, el soporte de los valores no atómicos puede resultar muy útil en ese tipo de dominios. De hecho, los sistemas modernos de bases de datos soportan muchos tipos de valores no atómicos, como se verá en el Capítulo 9. Sin embargo, en este capítulo nos limitamos a las relaciones en la primera forma normal y, por tanto, todos los dominios son atómicos.

7.3 Descomposición mediante dependencias funcionales

En el Apartado 7.1 se indicó que hay una metodología formal para evaluar si un esquema relacional debe descomponerse. Esta metodología se basa en los conceptos de clave y de dependencia funcional.

7.3.1 Claves y dependencias funcionales

Las claves y, más en general, las dependencias funcionales son restricciones de la base de datos que exigen que las relaciones cumplan determinadas propiedades. Las relaciones que cumplen todas esas restricciones son **legales**.

En el Capítulo 6 se definió el concepto de *superclave* de la manera siguiente. Sea R el esquema de una relación. El subconjunto C de R es una **superclave** de R si, en cualquier relación legal $r(R)$, para todos los pares t_1 y t_2 de tuplas de r tales que $t_1 \neq t_2$, $t_1[C] \neq t_2[C]$. Es decir, ningún par de tuplas de una relación legal $r(R)$ puede tener el mismo valor del conjunto de atributos C.

Mientras que una clave es un conjunto de atributos que identifica de manera unívoca toda una tupla, una dependencia funcional permite expresar restricciones que identifican de manera unívoca el valor de determinados atributos. Considérese el esquema de una relación R y sean $\alpha \subseteq R$ y $\beta \subseteq R$. La **dependencia funcional** $\alpha \rightarrow \beta$ se cumple para el esquema R si, en cualquier relación legal $r(R)$, para todos los pares de tuplas t_1 y t_2 de r tales que $t_1[\alpha] = t_2[\alpha]$, también se cumple que $t_1[\beta] = t_2[\beta]$.

Empleando la notación para la dependencia funcional, se dice que C es superclave de R si $C \rightarrow R$. Es decir, C es superclave si, siempre que $t_1[C] = t_2[C]$, también se cumple que $t_1[R] = t_2[R]$ (es decir, $t_1 = t_2$).

Las dependencias funcionales permiten expresar las restricciones que no se pueden expresar con superclaves. En el Apartado 7.1.2 se consideró el esquema

$$\textit{prestatario_préstamo} = (\underline{\textit{id_cliente, número_préstamo}}, \textit{importe})$$

en el que se cumple la dependencia funcional *número_préstamo* → *importe*, ya que por cada préstamo (identificado por *número_préstamo*) hay un solo importe. El hecho de que el par de atributos (*id_cliente, número_préstamo*) forma una superclave para *prestatario_préstamo* se denota escribiendo

$$\textit{id_cliente, número_préstamo} \rightarrow \textit{id_cliente, número_préstamo, importe}$$

o, de manera equivalente,

$$\textit{id_cliente, número_préstamo} \rightarrow \textit{prestatario_préstamo}$$

Las dependencias funcionales se emplean de dos maneras:

1. Para probar las relaciones y ver si son legales de acuerdo con un conjunto dado de dependencias funcionales. Si una relación r es legal según el conjunto F de dependencias funcionales, se dice que r **satisface** F.
2. Para especificar las restricciones del conjunto de relaciones legales. Así, *sólo* habrá que preocuparse de las relaciones que satisfagan un conjunto dado de dependencias funcionales. Si uno desea restringirse a las relaciones del esquema R que satisfacen el conjunto F de dependencias funcionales, se dice que F se **cumple** en R.

Considérese la relación r de la Figura 7.5, para ver las dependencias funcionales que se satisfacen. Obsérvese que se satisface $A \rightarrow C$. Hay dos tuplas que tienen un valor para A de a_1. Estas tuplas tienen el mismo valor de C—por ejemplo, c_1. De manera parecida, las dos tuplas con un valor para A de a_2

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4

Figura 7.5 Relación de ejemplo r.

tienen el mismo valor de *C*, c_2. No hay más pares de tuplas diferentes que tengan el mismo valor de *A*. Sin embargo, la dependencia funcional $C \rightarrow A$ no se satisface. Para verlo, considérense las tuplas $t_1 = (a_2, b_3, c_2, d_3)$ y $t_2 = (a_3, b_3, c_2, d_4)$. Estas dos tuplas tienen el mismo valor de *C*, c_2, pero tienen valores diferentes para *A*, a_2 y a_3, respectivamente. Por tanto, se ha hallado un par de tuplas t_1 y t_2 tales que $t_1[C] = t_2[C]$, but $t_1[A] \neq t_2[A]$.

Se dice que algunas dependencias funcionales son **triviales** porque las satisfacen todas las relaciones. Por ejemplo, $A \rightarrow A$ la satisfacen todas las relaciones que implican al atributo *A*. La lectura literal de la definición de dependencia funcional deja ver que, para todas las tuplas t_1 y t_2 tales que $t_1[A] = t_2[A]$, se cumple que $t_1[A] = t_2[A]$. De manera análoga, $AB \rightarrow A$ la satisfacen todas las relaciones que implican al atributo *A*. En general, las dependencias funcionales de la forma $\alpha \rightarrow \beta$ son **triviales** si se cumple la condición $\beta \subseteq \alpha$.

Es importante darse cuenta de que una relación dada puede, en cualquier momento, satisfacer algunas dependencias funcionales cuyo cumplimiento no sea necesario en el esquema de la relación. En la relación *cliente* de la Figura 2.4 puede verse que se satisface *calle_cliente* → *ciudad_cliente*. Sin embargo, se sabe que en el mundo real dos ciudades diferentes pueden tener calles que se llamen igual. Por tanto, es posible, en un momento dado, tener un ejemplar de la relación *cliente* en el que no se satisfaga *calle_cliente* → *ciudad_cliente*. Por consiguiente, no se incluirá *calle_cliente* → *ciudad_cliente* en el conjunto de dependencias funcionales que se cumplen en la relación *cliente*.

Dado un conjunto de dependencias funcionales *F* que se cumple en una relación *r*, es posible que se pueda inferir que también se deban cumplir en esa misma relación otras dependencias funcionales. Por ejemplo, dado el esquema $r = (A, B, C)$, si se cumplen en *r* las dependencias funcionales $A \rightarrow B$ y $B \rightarrow C$, se puede inferir que también $A \rightarrow C$ se debe cumplir en *r*. Ya que, dado cualquier valor de *A*, sólo puede haber un valor correspondiente de *B* y, para ese valor de *B*, sólo puede haber un valor correspondiente de *C*. Más adelante, en el Apartado 7.4.1, se estudia la manera de realizar esas inferencias.

Se utiliza la notación F^+ para denotar el **cierre** del conjunto *F*, es decir, el conjunto de todas las dependencias funcionales que pueden inferirse dado el conjunto *F*. Evidentemente, F^+ es un superconjunto de *F*.

7.3.2 Forma normal de Boyce–Codd

Una de las formas normales más deseables que se pueden obtener es la **forma normal de Boyce-Codd** (**FNBC**). Elimina todas las redundancias que se pueden descubrir a partir de las dependencias funcionales aunque, como se verá en el Apartado 7.6, puede que queden otros tipos de redundancia. El esquema de relación *R* está en la FNBC respecto al conjunto *F* de dependencias funcionales si, para todas las dependencias funcionales de F^+ de la forma $\alpha \rightarrow \beta$, donde $\alpha \subseteq R$ y $\beta \subseteq R$, se cumple, al menos, una de las siguientes condiciones:

- $\alpha \rightarrow \beta$ es una dependencia funcional trivial (es decir, $\beta \subseteq \alpha$).
- α es superclave del esquema *R*.

Los diseños de bases de datos están en la FNBC si cada miembro del conjunto de esquemas de relación que constituye el diseño se halla en la FNBC.

Ya se ha visto que el esquema *prestatario_préstamo* = (*id_cliente, número_préstamo, importe*) no se halla en FNBC. La dependencia funcional *número_préstamo* → *importe* se cumple en *prestatario_préstamo*, pero *número_préstamo* no es superclave (ya que, como se recordará, los préstamos se pueden conceder a consorcios formados por varios clientes). En el Apartado 7.1.2 se vio que la descomposición de *prestatario_préstamo* en *prestatario* y *préstamo* es mejor diseño. El esquema *prestatario* se halla en la FNBC, ya que no se cumple en él ninguna dependencia funcional que no sea trivial. El esquema *préstamo* tiene una dependencia funcional no trivial que se cumple, *número_préstamo* → *importe*, pero *número_préstamo* es superclave (realmente, en este caso, la clave primaria) de *préstamo*. Por tanto, *préstamo* se halla en la FNBC.

Ahora se va a definir una regla general para la descomposición de esquemas que no se hallen en la FNBC. Sea R un esquema que no se halla en la FNBC. En ese caso, queda, al menos, una dependencia funcional no trivial $\alpha \rightarrow \beta$ tal que α no es superclave de R. En el diseño se sustituye R por dos esquemas:

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

En el caso anterior de *prestatario_préstamo*, $\alpha =$ *número_préstamo*, $\beta =$ *importe* y *prestatario_préstamo* se sustituye por

- $(\alpha \cup \beta)$ = (*número_préstamo*, *importe*)
- $(R - (\beta - \alpha))$ = (*id_cliente*, *número_préstamo*)

En este caso resulta que $\beta - \alpha = \beta$. Hay que definir la regla tal y como se ha hecho para que trate correctamente las dependencias funcionales que tienen atributos que aparecen a los dos lados de la flecha. Las razones técnicas para esto se tratarán más adelante, en el Apartado 7.5.1.

Cuando se descomponen esquemas que no se hallan en la FNBC, puede que uno o varios de los esquemas resultantes no se hallen en la FNBC. En ese caso, hacen falta más descomposiciones, cuyo resultado final es un conjunto de esquemas FNBC.

7.3.3 FNBC y la conservación de las dependencias

Se han visto varias maneras de expresar las restricciones de consistencia de las bases de datos: restricciones de clave primaria, dependencias funcionales, restricciones **check**, asertos y disparadores. La comprobación de estas restricciones cada vez que se actualiza la base de datos puede ser costosa y, por tanto, resulta útil diseñar la base de datos de manera que las restricciones se puedan comprobar de manera eficiente. En concreto, si la comprobación de las dependencias funcionales puede realizarse considerando sólo una relación, el coste de comprobación de esa restricción será bajo. Se verá que la descomposición en la FNBC puede impedir la comprobación eficiente de determinadas dependencias funcionales.

Para ilustrar esto, supóngase que se lleva a cabo una modificación aparentemente pequeña en el modo en que opera la entidad bancaria. En el diseño de la Figura 6.25 cada cliente sólo puede tener un empleado como "asesor personal". Esto se deduce de que el conjunto de relaciones *asesor* es varios a uno de *cliente* a *empleado*. La "pequeña" modificación que se va a llevar a cabo es que cada cliente pueda tener más de un asesor personal pero, como máximo, uno por sucursal.

Esto se puede implementar en el diagrama E-R haciendo que el conjunto de relaciones *asesor* sea varios a varios (ya que ahora cada cliente puede tener más de un asesor personal) y añadiendo un nuevo conjunto de relaciones, *trabaja_en*, entre *empleado* y *sucursal*, que indique los pares empleado-sucursal, en los que el empleado trabaja en la sucursal. Hay que hacer *trabaja_en* varios a uno de *empleado* a *sucursal*, ya que cada sucursal puede tener varios empleados, pero cada empleado sólo puede trabajar en una sucursal. La Figura 7.6 muestra un subconjunto de la Figura 6.25 con estas adiciones.

No obstante, hay un fallo en este diseño. Permite que cada cliente tenga dos (o más) asesores personales que trabajen en la misma sucursal, algo que el banco no permite. Lo ideal sería que hubiera un solo conjunto de relaciones al que se pudiera hacer referencia para hacer que se cumpla esta restricción. Esto exige que se considere una manera diferente de modificar el diseño E-R. En lugar de añadir el conjunto de relaciones *trabaja_en*, se sustituye el conjunto de relaciones *asesor* por la relación ternaria *sucursal_asesor*, que implica a los conjuntos de entidades *cliente*, *empleado* y *sucursal* y es varios a uno del par *cliente*, *empleado* a *sucursal*, como puede verse en la Figura 7.7. Como este diseño permite que un solo conjunto de relaciones represente la restricción, supone una ventaja significativa respecto del primer enfoque considerado.

Sin embargo, la diferencia entre estos dos enfoques no es tan clara. El esquema derivado de *sucursal_asesor* es

sucursal_asesor = (*id_cliente*, *id_empleado*, *nombre_sucursal*, *tipo*)

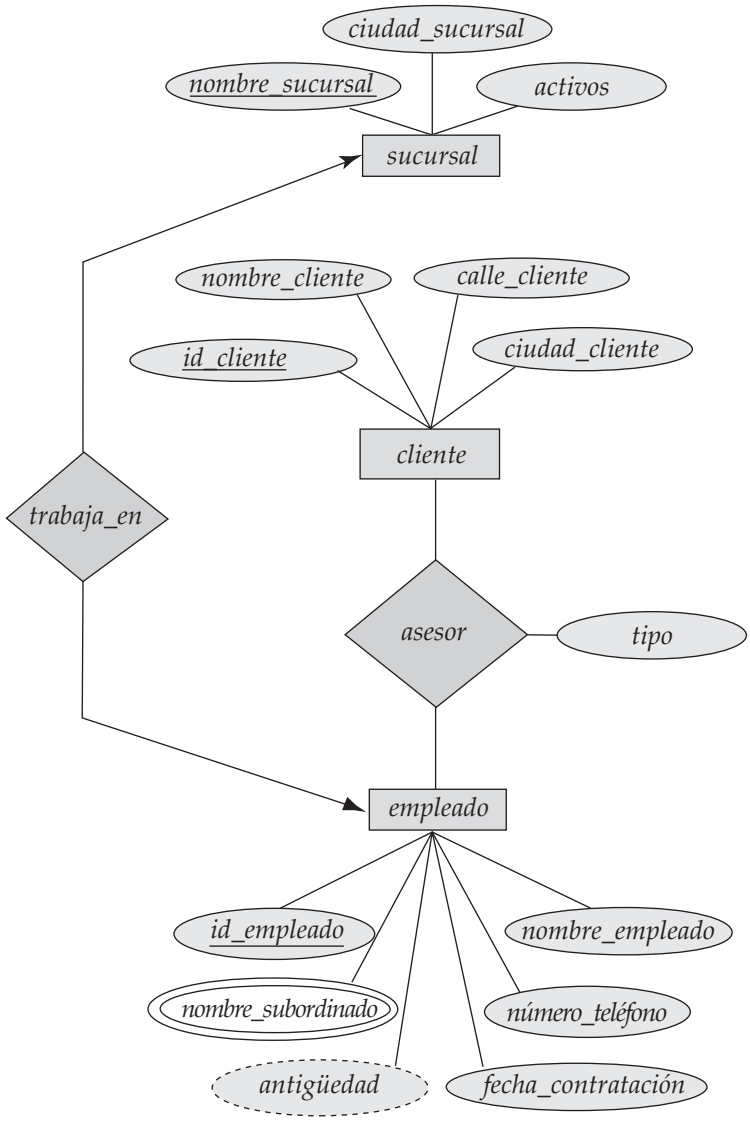


Figura 7.6 Los conjuntos de relaciones *asesor* y *trabaja_en*.

Como cada empleado sólo puede trabajar en una sucursal, en la relación del esquema *sucursal_asesor* sólo puede haber un valor de *nombre_sucursal* asociado con cada valor de *id_empleado*; es decir:

$$id_empleado \rightarrow nombre_sucursal$$

No obstante, no queda más remedio que repetir el nombre de la sucursal cada vez que un empleado participa en la relación *sucursal_asesor*. Es evidente que *sucursal_asesor* no se halla en la FNBC, ya que *id_empleado* no es superclave. De acuerdo con la regla para la descomposición en la FNBC, se obtiene:

(*id_cliente*, *id_empleado*, *tipo*)
(*id_empleado*, *nombre_sucursal*)

Este diseño, que es exactamente igual que el primer enfoque que utilizaba el conjunto de relaciones *trabaja_en*, dificulta que se haga cumplir la restricción de que cada cliente pueda tener, como máximo, un asesor en cada sucursal. Esta restricción se puede expresar mediante la dependencia funcional

$$id_cliente, nombre_sucursal \rightarrow id_empleado$$

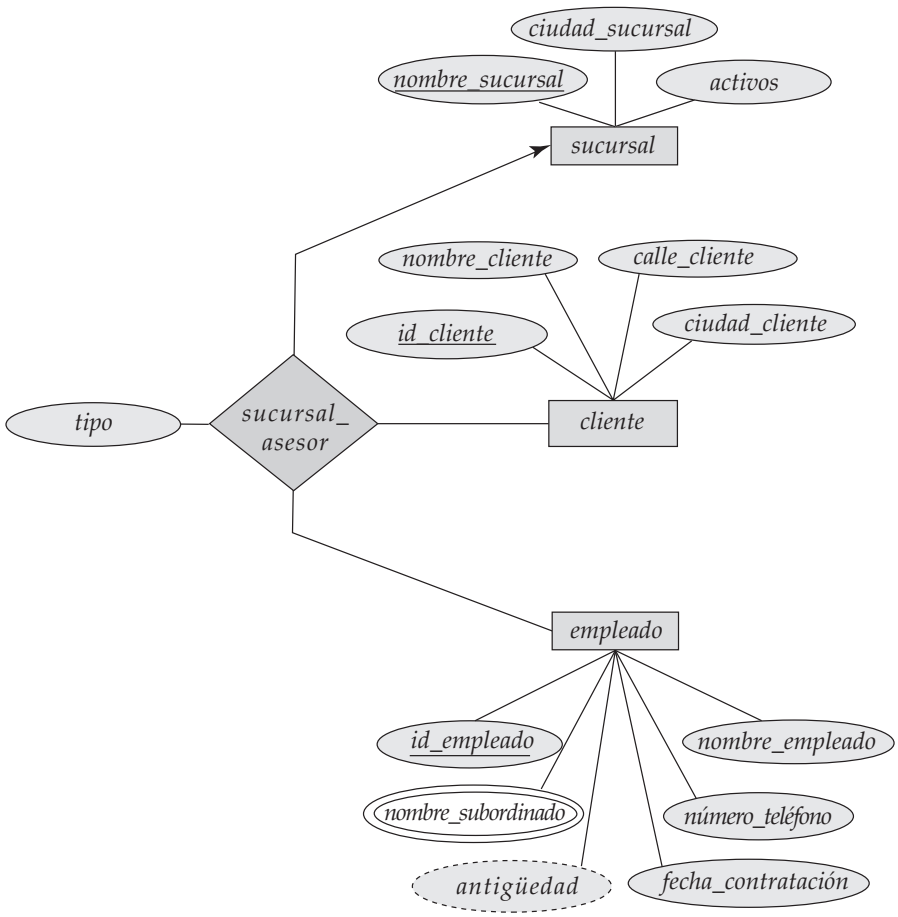


Figura 7.7 El conjunto de relaciones *sucursal_asesor*.

y hacer notar que en el diseño FNBC escogido no hay ningún esquema que incluya todos los atributos que aparecen en esta dependencia funcional. Como el diseño hace computacionalmente difícil que se haga cumplir esta dependencia funcional, se dice que el diseño no **conserva las dependencias**[1]. Como la conservación de las dependencias suele considerarse deseable, se considera otra forma normal, más débil que la FNBC, que permita que se conserven las dependencias. Esa forma normal se denomina tercera forma normal[2].

7.3.4 Tercera forma normal

La FNBC exige que todas las dependencias no triviales sean de la forma $\alpha \rightarrow \beta$, donde α es una superclave. La tercera forma normal (3NF) relaja ligeramente esta restricción al permitir dependencias funcionales no triviales cuya parte izquierda no sea una superclave. Antes de definir la 3NF hay que recordar que las claves candidatas son superclaves mínimas—es decir, superclaves de las que ningún subconjunto propio sea también superclave.

El esquema de relación R está en **tercera forma normal** respecto a un conjunto F de dependencias funcionales si, para todas las dependencias funcionales de F^+ de la forma $\alpha \rightarrow \beta$, donde $\alpha \subseteq R$ y $\beta \subseteq R$, se cumple, al menos, una de las siguientes condiciones:

1. Técnicamente es posible que una dependencia cuyos atributos no aparezcan en su totalidad en ningún esquema se siga haciendo cumplir de manera implícita, debido a la presencia de otras dependencias que la impliquen lógicamente. Este caso se abordará más adelante, en el Apartado 7.4.5.
2. Puede que se haya observado que se ha omitido la segunda forma normal. Sólo tiene significación histórica y, en la práctica, no se utiliza.

- $\alpha \rightarrow \beta$ es una dependencia funcional trivial.
- α es superclave de R.
- Cada atributo A de $\beta - \alpha$ está contenido en alguna clave candidata de R.

Obsérvese que la tercera condición no dice que una sola clave candidata deba contener todos los atributos de $\beta - \alpha$; cada atributo A de $\beta - \alpha$ puede estar contenido en una clave candidata *diferente*.

Las dos primeras alternativas son iguales que las dos alternativas de la definición de la FNBC. La tercera alternativa de la definición de la 3FN parece bastante poco intuitiva, y no resulta evidente el motivo de su utilidad. Representa, en cierto sentido, una relajación mínima de las condiciones de la FNBC que ayuda a garantizar que cada esquema tenga una descomposición que conserve las dependencias en la 3FN. Su finalidad se aclarará más adelante, cuando se estudie la descomposición en la 3FN.

Obsérvese que cualquier esquema que satisfaga la FNBC satisface también la 3FN, ya que cada una de sus dependencias funcionales satisfará una de las dos primeras alternativas. Por tanto, la FNBC es una forma normal más restrictiva que la 3FN.

La definición de la 3FN permite ciertas dependencias funcionales que no se permiten en la FNBC. Las dependencias $\alpha \rightarrow \beta$ que sólo satisfacen la tercera alternativa de la definición de la 3FN no se permiten en la FNBC, pero sí en la 3FN[3].

Considérese nuevamente el ejemplo *sucursal_asesor* y la dependencia funcional

$$\textit{id_empleado} \rightarrow \textit{nombre_sucursal}$$

que hacía que el esquema no se hallara en FNBC. Obsérvese que, en este caso, $\alpha = \textit{id_empleado}$, $\beta = \textit{nombre_sucursal}$ y $\beta - \alpha = \textit{nombre_sucursal}$. Resulta que *nombre_sucursal* está contenido en una clave candidata y que, por tanto, *sucursal_asesor* se halla en la 3FN. Probar esto, no obstante, exige un pequeño esfuerzo.

Se sabe que, además de las dependencias funcionales

$$\textit{id_empleado} \rightarrow \textit{nombre_sucursal}$$
$$\textit{id_cliente, nombre_sucursal} \rightarrow \textit{id_empleado}$$

que se cumplen, la dependencia funcional

$$\textit{id_cliente, id_empleado} \rightarrow \textit{sucursal_asesor}$$

se cumple como consecuencia de que (*id_cliente, id_empleado*) es la clave primaria. Esto hace de (*id_cliente, id_empleado*) clave candidata. Por supuesto, no contiene *nombre_sucursal*, por lo que hay que ver si hay otras claves candidatas. Se concluye que el conjunto de atributos (*id_cliente, nombre_sucursal*) es clave candidata. Veamos el motivo.

Dados unos valores concretos de *id_cliente* y de *nombre_sucursal*, se sabe que sólo hay un valor asociado de *id_empleado*, ya que

$$\textit{id_cliente, nombre_sucursal} \rightarrow \textit{id_empleado}$$

Pero entonces, para esos valores concretos de *id_cliente* y de *id_empleado*, sólo puede haber una tupla asociada de *sucursal_asesor*, ya que

$$\textit{id_cliente, id_empleado} \rightarrow \textit{sucursal_asesor}$$

Por tanto, se ha argumentado que (*id_cliente, nombre_sucursal*) es superclave. Como resulta que ni *id_cliente* ni *nombre_sucursal* por separado son superclaves, (*id_cliente, nombre_sucursal*) es clave candida-

3. Estas dependencias son ejemplos de **dependencias transitivas** (véase el Ejercicio práctico 7.14). La definición original de la 3NF se hizo en términos de las dependencias transitivas. La definición que aquí se utiliza es equivalente, pero más fácil de comprender.

ta. Dado que esta clave candidata contiene *nombre_sucursal*, la dependencia funcional

$$id_empleado \rightarrow nombre_sucursal$$

no viola las reglas de la 3FN.

La demostración de que *sucursal_asesor* se halla en la 3NF ha supuesto cierto esfuerzo. Por esta razón (y por otras), resulta útil adoptar un enfoque estructurado y formal del razonamiento sobre las dependencias funcionales, las formas normales y la descomposición de los esquemas, lo cual se hará en el Apartado 7.4.

Se ha visto el equilibrio que hay que guardar entre la FNBC y la 3NF cuando no hay ningún diseño FNBC que conserve las dependencias. Estos equilibrios se describen con más detalle en el Apartado 7.5.3; en ese apartado también se describe un enfoque de la comprobación de dependencias mediante vistas materializadas que permite obtener las ventajas de la FNBC y de la 3NF.

7.3.5 Formas normales superiores

Puede que en algunos casos el empleo de las dependencias funcionales para la descomposición de los esquemas no sea suficiente para evitar la repetición innecesaria de información. Considérese una ligera variación de la definición del conjunto de entidades *empleado* en la que se permita que los empleados tengan varios números de teléfono, alguno de los cuales puede ser compartido entre varios empleados. Entonces, *número_teléfono* será un atributo multivalorado y, de acuerdo con las reglas para la generación de esquemas a partir de los diseños E-R, habrá dos esquemas, uno por cada uno de los atributos multivalorados *número_teléfono* y *nombre_subordinado*:

(*id_empleado, nombre_subordinado*) (*id_empleado, número_teléfono*)

Si se combinan estos esquemas para obtener

(*id_empleado, nombre_subordinado, número_teléfono*)

se descubre que el resultado se halla en la FNBC, ya que sólo se cumplen dependencias funcionales no triviales. En consecuencia, se puede pensar que ese tipo de combinación es una buena idea. Sin embargo, se trata de una mala idea, como puede verse si se considera el ejemplo de un empleado con dos subordinados y dos números de teléfono. Por ejemplo, sea el empleado con *id_empleado* 999999999 que tiene dos subordinados llamados "David" y "Guillermo" y dos números de teléfono, 512555123 y 512555432. En el esquema combinado hay que repetir los números de teléfono una vez por cada subordinado:

(999999999, David, 512555123)
(999999999, David, 512555432)
(999999999, Guillermo, 512555123)
(999999999, Guillermo, 512555432)

Si no se repitieran los números de teléfono y sólo se almacenaran la primera y la última tupla, se habrían guardado el nombre de los subordinados y los números de teléfono, pero las tuplas resultantes implicarían que David correspondería al 512555123 y que Guillermo correspondería al 512555432. Como sabemos, esto es incorrecto.

Debido a que las formas normales basadas en las dependencias funcionales no son suficientes para tratar con situaciones como ésta, se han definido otras dependencias y formas normales. Se estudian en los Apartados 7.6 y 7.7.

7.4 Teoría de las dependencias funcionales

Se ha visto en los ejemplos que resulta útil poder razonar de manera sistemática sobre las dependencias funcionales como parte del proceso de comprobación de los esquemas para la FNBC o la 3NF.

7.4.1 Cierre de los conjuntos de dependencias funcionales

No es suficiente considerar el conjunto dado de dependencias funcionales. También hay que considerar *todas* las dependencias funcionales que se cumplen. Se verá que, dado un conjunto F de dependencias funcionales, se puede probar que también se cumple alguna otra dependencia funcional. Se dice que F "implica lógicamente" esas dependencias funcionales.

De manera más formal, dado un esquema relacional R, una dependencia funcional f de R está **implicada lógicamente** por un conjunto de dependencias funcionales F de R si cada ejemplar de la relación $r(R)$) que satisface F satisface también f.

Supóngase que se tiene el esquema de relación $R = (A, B, C, G, H, I)$ y el conjunto de dependencias funcionales

$$A \rightarrow B$$
$$A \rightarrow C$$
$$CG \rightarrow H$$
$$CG \rightarrow I$$
$$B \rightarrow H$$

La dependencia funcional

$$A \rightarrow H$$

está implicada lógicamente. Es decir, se puede demostrar que, siempre que el conjunto dado de dependencias funcionales se cumple en una relación, también se debe cumplir $A \rightarrow H$. Supóngase que t_1 y t_2 son tuplas tales que

$$t_1[A] = t_2[A]$$

Como se tiene que $A \rightarrow B$, se deduce de la definición de dependencia funcional que

$$t_1[B] = t_2[B]$$

Entonces, como se tiene que $B \rightarrow H$, se deduce de la definición de dependencia funcional que

$$t_1[H] = t_2[H]$$

Por tanto, se ha demostrado que, siempre que t_1 y t_2 sean tuplas tales que $t_1[A] = t_2[A]$, debe ocurrir que $t_1[H] = t_2[H]$. Pero ésa es exactamente la definición de $A \rightarrow H$.

Sea F un conjunto de dependencias funcionales. El **cierre** de F, denotado por F^+, es el conjunto de todas las dependencias funcionales implicadas lógicamente por F. Dado F, se puede calcular F^+ directamente a partir de la definición formal de dependencia funcional. Si F fuera de gran tamaño, ese proceso sería largo y difícil. Este tipo de cálculo de F^+ requiere argumentos del tipo que se acaba de utilizar para demostrar que $A \rightarrow H$ está en el cierre del conjunto de dependencias de ejemplo.

Los **axiomas**, o reglas de inferencia, proporcionan una técnica más sencilla para el razonamiento sobre las dependencias funcionales. En las reglas que se ofrecen a continuación se utilizan las letras griegas (α, β, γ, ...) para los conjuntos de atributos y las letras latinas mayúsculas desde el comienzo del alfabeto para los atributos. Se emplea $\alpha\beta$ para denotar $\alpha \cup \beta$.

Se pueden utilizar las tres reglas siguientes para hallar las dependencias funcionales implicadas lógicamente. Aplicando estas reglas *repetidamente* se puede hallar todo F^+, dado F. Este conjunto de reglas se denomina **axiomas de Armstrong** en honor de la persona que las propuso por primera vez.

- **Regla de la reflexividad**. Si α es un conjunto de atributos y $\beta \subseteq \alpha$, entonces se cumple que $\alpha \rightarrow \beta$.
- **Regla de la aumentatividad**. Si se cumple que $\alpha \rightarrow \beta$ y γ es un conjunto de atributos, entonces se cumple que $\gamma\alpha \rightarrow \gamma\beta$.
- **Regla de la transitividad**. Si se cumple que $\alpha \rightarrow \beta$ y que $\beta \rightarrow \gamma$, entonces se cumple que $\alpha \rightarrow \gamma$.

Los axiomas de Armstrong son **correctos**, ya que no generan dependencias funcionales incorrectas. Son **completos**, ya que, para un conjunto de dependencias funcionales F dado, permiten generar todo

F^+. Las notas bibliográficas proporcionan referencias de las pruebas de su corrección y de su completitud.

Aunque los axiomas de Armstrong son completos, resulta pesado utilizarlos directamente para el cálculo de F^+. Para simplificar más las cosas se dan unas reglas adicionales. Se pueden utilizar los axiomas de Armstrong para probar que son correctas (véanse los Ejercicios prácticos 7.4 y 7.5 y el Ejercicio 7.21).

- **Regla de la unión**. Si se cumple que $\alpha \rightarrow \beta$ y que $\alpha \rightarrow \gamma$, entonces se cumple que $\alpha \rightarrow \beta\gamma$.
- **Regla de la descomposición**. Si se cumple que $\alpha \rightarrow \beta\gamma$, entonces se cumple que $\alpha \rightarrow \beta$ y que $\alpha \rightarrow \gamma$.
- **Regla de la pseudotransitividad**. Si se cumple que $\alpha \rightarrow \beta$ y que $\gamma\beta \rightarrow \delta$, entonces se cumple que $\alpha\gamma \rightarrow \delta$.

Se aplicarán ahora las reglas al ejemplo del esquema $R = (A, B, C, G, H, I)$ y del conjunto F de dependencias funcionales $\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$. A continuación se relacionan varios miembros de F^+:

- $A \rightarrow H$. Dado que se cumplen $A \rightarrow B$ y $B \rightarrow H$, se aplica la regla de transitividad. Obsérvese que resultaba mucho más sencillo emplear los axiomas de Armstrong para demostrar que se cumple que $A \rightarrow H$, que deducirlo directamente a partir de las definiciones, como se ha hecho anteriormente en este apartado.
- $CG \rightarrow HI$. Dado que $CG \rightarrow H$ y $CG \rightarrow I$, la regla de la unión implica que $CG \rightarrow HI$.
- $AG \rightarrow I$. Dado que $A \rightarrow C$ y $CG \rightarrow I$, la regla de pseudotransitividad implica que se cumple que $AG \rightarrow I$.

  Otra manera de averiguar que se cumple que $AG \rightarrow I$ es la siguiente. Se utiliza la regla de aumentatividad en $A \rightarrow C$ para inferir que $AG \rightarrow CG$. Aplicando la regla de transitividad a esta dependencia y a $CG \rightarrow I$, se infiere que $AG \rightarrow I$.

La Figura 7.8 muestra un procedimiento que demuestra formalmente el modo de utilizar los axiomas de Armstrong para calcular F^+. En este procedimiento puede que la dependencia funcional ya esté presente en F^+ cuando se le añade y, en ese caso, no hay ninguna modificación en F^+. En el Apartado 7.4.2 se verá una manera alternativa de calcular F^+.

Los términos a la derecha y a la izquierda de las dependencias funcionales son subconjuntos de R. Dado que un conjunto de tamaño n tiene 2^n subconjuntos, hay un total de $2 \times 2^n = 2^{n+1}$ dependencias funcionales posibles, donde n es el número de atributos de R. Cada iteración del bucle repetir del procedimiento, salvo la última, añade como mínimo una dependencia funcional a F^+. Por tanto, está garantizado que el procedimiento termine.

```
F+ = F
repeat
    for each dependencia funcional f de F+
        aplicar las reglas de reflexividad y de aumentatividad a f
        añadir las dependencias funcionales resultantes a F+
    for each par de dependencias funcionales f1 y f2 de F+
        if f1 y f2 se pueden combinar mediante la transitividad
            añadir la dependencia funcional resultante a F+
until F+ deje de cambiar
```

Figura 7.8 Procedimiento para calcular F^+.

7.4.2 Cierre de los conjuntos de atributos

Se dice que un atributo B está **determinado funcionalmente** por α si $\alpha \rightarrow B$. Para comprobar si un conjunto α es superclave hay que diseñar un algoritmo para el cálculo del conjunto de atributos determinados funcionalmente por α. Una manera de hacerlo es calcular F^+, tomar todas las dependencias funcionales con α como término de la izquierda y tomar la unión de los términos de la derecha de todas esas dependencias. Sin embargo, hacer esto puede resultar costoso, ya que F^+ puede ser de gran tamaño.

Un algoritmo eficiente para el cálculo del conjunto de atributos determinados funcionalmente por α no sólo resulta útil para comprobar si α es superclave, sino también para otras tareas, como se verá más adelante en este apartado.

Sea α un conjunto de atributos. Al conjunto de todos los atributos determinados funcionalmente por α bajo un conjunto F de dependencias funcionales se le denomina **cierre** de α bajo F; se denota mediante α^+. La Figura 7.9 muestra un algoritmo, escrito en pseudocódigo, para calcular α^+. La entrada es un conjunto F de dependencias funcionales y el conjunto α de atributos. La salida se almacena en la variable *resultado*.

Para ilustrar el modo en que trabaja el algoritmo se utilizará para calcular $(AG)^+$ con las dependencias funcionales definidas en el Apartado 7.4.1. Se comienza con $result = AG$. La primera vez que se ejecuta el bucle **while** para comprobar cada dependencia funcional se halla que

- $A \rightarrow B$ hace que se incluya B en *resultado*. Para comprobarlo, obsérvese que $A \rightarrow B$ se halla en F y $A \subseteq$ *resultado* (que es AG), por lo que *resultado* := *resultado* $\cup B$.
- $A \rightarrow C$ hace que *resultado* se transforme en $ABCG$.
- $CG \rightarrow H$ hace que *resultado* se transforme en $ABCGH$.
- $CG \rightarrow I$ hace que *resultado* se transforme en $ABCGHI$.

La segunda vez que se ejecuta el bucle **while** ya no se añade ningún atributo nuevo a *resultado*, y se termina el algoritmo.

Veamos ahora el motivo de que el algoritmo de la Figura 7.9 sea correcto. El primer paso es correcto, ya que $\alpha \rightarrow \alpha$ se cumple siempre (por la regla de reflexividad). Se afirma que, para cualquier subconjunto β de *resultado*, $\alpha \rightarrow \beta$. Dado que el bucle **while** se inicia con $\alpha \rightarrow$ *resultado* como cierto, sólo se puede añadir γ a *resultado* si $\beta \subseteq$ *resultado* y $\beta \rightarrow \gamma$. Pero, entonces, *resultado* $\rightarrow \beta$ por la regla de reflexividad, por lo que $\alpha \rightarrow \beta$ por transitividad. Otra aplicación de la transitividad demuestra que $\alpha \rightarrow \gamma$ (empleando $\alpha \rightarrow \beta$ y $\beta \rightarrow \gamma$). La regla de la unión implica que $\alpha \rightarrow$ *resultado* $\cup \gamma$, por lo que α determina funcionalmente cualquier resultado nuevo generado en el bucle **while**. Por tanto, cualquier atributo devuelto por el algoritmo se halla en α^+.

Resulta sencillo ver que el algoritmo halla todo α^+. Si hay un atributo de α^+ que no se halle todavía en *resultado*, debe haber una dependencia funcional $\beta \rightarrow \gamma$ para la que $\beta \subseteq$ *resultado* y, como mínimo, un atributo de γ no se halla en *resultado*.

En el peor de los casos es posible que este algoritmo tarde un tiempo proporcional al cuadrado del tamaño de F. Hay un algoritmo más rápido (aunque ligeramente más complejo) que se ejecuta en un tiempo proporcional al tamaño de F; ese algoritmo se presenta como parte del Ejercicio práctico 7.8.

Existen varias aplicaciones del algoritmo de cierre de atributos:

```
resultado := α;
while (cambios en resultado) do
        for each dependencia funcional β → γ in F do
            begin
                if β ⊆ resultado then resultado := resultado ∪ γ;
            fin
```

Figura 7.9 Algoritmo para el cálculo de α^+, el cierre de α bajo F.

- Para comprobar si α es superclave, se calcula α^+ y se comprueba si contiene todos los atributos de *R*.
- Se puede comprobar si se cumple la dependencia funcional $\alpha \rightarrow \beta$ (o, en otras palabras, si se halla en F^+), comprobando si $\beta \subseteq \alpha^+$. Es decir, se calcula α^+ empleando el cierre de los atributos y luego se comprueba si contiene a β. Esta prueba resulta especialmente útil, como se verá más adelante en este mismo capítulo.
- Ofrece una manera alternativa de calcular F^+: para cada $\gamma \subseteq R$ se halla el cierre γ^+ y, para cada $S \subseteq \gamma^+$, se genera la dependencia funcional $\gamma \rightarrow S$.

7.4.3 Recubrimiento canónico

Supóngase que se tiene un conjunto *F* de dependencias funcionales de un esquema de relación. Siempre que un usuario lleve a cabo una actualización de la relación, el sistema de bases de datos debe asegurarse de que la actualización no viole ninguna dependencia funcional, es decir, que se satisfagan todas las dependencias funcionales de *F* en el nuevo estado de la base de datos.

El sistema debe retroceder la actualización si viola alguna dependencia funcional del conjunto *F*.

Se puede reducir el esfuerzo dedicado a la comprobación de las violaciones comprobando un conjunto simplificado de dependencias funcionales que tenga el mismo cierre que el conjunto dado. Cualquier base de datos que satisfaga el conjunto simplificado de dependencias funcionales satisfará también el conjunto original y viceversa, ya que los dos conjuntos tienen el mismo cierre. Sin embargo, el conjunto simplificado resulta más sencillo de comprobar. En breve se verá el modo en que se puede crear ese conjunto simplificado. Antes, hacen falta algunas definiciones.

Se dice que un atributo de una dependencia funcional es **raro** si se puede eliminar sin modificar el cierre del conjunto de dependencias funcionales. La definición formal de los **atributos raros** es la siguiente. Considérese un conjunto *F* de dependencias funcionales y la dependencia funcional $\alpha \rightarrow \beta$ de *F*.

- El atributo *A* es raro en $alpha$ si $A \in \alpha$ y *F* implica lógicamente a $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
- El atributo *A* es raro en β si $A \in \beta$ y el conjunto de dependencias funcionales $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ implica lógicamente a *F*.

Por ejemplo, supóngase que se tienen las dependencias funcionales $AB \rightarrow C$ y $A \rightarrow C$ de F. Entonces, B es raro en $AB \rightarrow C$. Como ejemplo adicional, supóngase que se tienen las dependencias funcionales $AB \rightarrow CD$ y $A \rightarrow C$ de F. Entonces, C será raro en el lado derecho de $AB \rightarrow CD$.

Hay que tener cuidado con la dirección de las implicaciones al utilizar la definición de los atributos raros: si se intercambian el lado derecho y el izquierdo, la implicación se cumplirá *siempre*. Es decir, $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$ siempre implica lógicamente a F, y F también implica lógicamente siempre a $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$.

A continuación se muestra el modo de comprobar de manera eficiente si un atributo es raro. Sea R el esquema de la relación y F el conjunto dado de dependencias funcionales que se cumplen en R. Considérese el atributo A de la dependencia $\alpha \rightarrow \beta$.

- Si $A \in \beta$, para comprobar si A es raro hay que considerar el conjunto
  $$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$$
  y comprobar si $\alpha \rightarrow A$ puede inferirse a partir de F'. Para ello hay que calcular α^+ (el cierre de α) bajo F'; si α^+ incluye a A, entonces A es raro en β.
- Si $A \in \alpha$, para comprobar si A es raro, sea $\gamma = \alpha - \{A\}$, hay que comprobar si se puede inferir que $\gamma \rightarrow \beta$ a partir de F. Para ello hay que calcular γ^+ (el cierre de γ) bajo F; si γ^+ incluye todos los atributos de β, entonces A es raro en α.

Por ejemplo, supóngase que F contiene $AB \rightarrow CD$, $A \rightarrow E$ y $E \rightarrow C$. Para comprobar si C es raro en $AB \rightarrow CD$, hay que calcular el cierre de los atributos de AB bajo $F' = \{AB \rightarrow D, A \rightarrow E$ y $E \rightarrow C\}$. El cierre es $ABCDE$, que incluye a CD, por lo que se infiere que C es raro.

$F_c = F$
repeat
  Utilizar la regla de unión para sustituir las dependencias de F_c de la forma
    $\alpha_1 \rightarrow \beta_1$ y $\alpha_1 \rightarrow \beta_2$ con $\alpha_1 \rightarrow \beta_1 \beta_2$.
  Hallar una dependencia funcional $\alpha \rightarrow \beta$ de F_c con un atributo
    raro en α o en β.
    /* Nota: la comprobación de los atributos raros se lleva a cabo empleando F_c, no F */
  Si se halla algún atributo raro, hay que eliminarlo de $\alpha \rightarrow \beta$.
until F_c ya no cambie.

Figura 7.10 Cálculo del recubrimiento canónico.

El **recubrimiento canónico** F_c de F es un conjunto de dependencias tal que F implica lógicamente todas las dependencias de F_c y F_c implica lógicamente todas las dependencias de F. Además, F_c debe tener las propiedades siguientes:

- Ninguna dependencia funcional de F_c contiene atributos raros.
- El lado izquierdo de cada dependencia funcional de F_c es único. Es decir, no hay dos dependencias $\alpha_1 \rightarrow \beta_1$ y $\alpha_2 \rightarrow \beta_2$ de F_c tales que $\alpha_1 = \alpha_2$.

El recubrimiento canónico del conjunto de dependencias funcionales F puede calcularse como se muestra en la Figura 7.10. Es importante destacar que, cuando se comprueba si un atributo es raro, la comprobación utiliza las dependencias del valor actual de F_c, y **no** las dependencias de F. Si una dependencia funcional sólo contiene un atributo en su lado derecho, por ejemplo, $A \rightarrow C$, y se descubre que ese atributo es raro, se obtiene una dependencia funcional con el lado derecho vacío. Hay que eliminar esas dependencias funcionales.

Se puede demostrar que el recubrimiento canónico de F, F_c, tiene el mismo cierre que F; por tanto, comprobar si se satisface F_c es equivalente a comprobar si se satisface F. Sin embargo, F_c es mínimo en un cierto sentido—no contiene atributos raros, y combina las dependencias funcionales con el mismo lado izquierdo. Resulta más económico comprobar F_c que comprobar el propio F.

Considérese el siguiente conjunto F de dependencias funcionales para el esquema (A, B, C):

$$A \rightarrow BC$$
$$B \rightarrow C$$
$$A \rightarrow B$$
$$AB \rightarrow C$$

Calcúlese el recubrimiento canónico de F.

- Hay dos dependencias funcionales con el mismo conjunto de atributos a la izquierda de la flecha:

  $$A \rightarrow BC$$
  $$A \rightarrow B$$

  Estas dependencias funcionales se combinan en $A \rightarrow BC$.

- A es raro en $AB \rightarrow C$, ya que F implica lógicamente a $(F - \{AB \rightarrow C\}) \cup \{B \rightarrow C\}$. Esta aseveración es cierta porque $B \rightarrow C$ ya se halla en el conjunto de dependencias funcionales.
- C es raro en $A \rightarrow BC$, ya que $A \rightarrow BC$ está implicada lógicamente por $A \rightarrow B$ y $B \rightarrow C$.

Por tanto, el recubrimiento canónico es

$$A \rightarrow B$$
$$B \rightarrow C$$

Dado un conjunto F de dependencias funcionales, puede suceder que toda una dependencia funcional del conjunto sea rara, en el sentido de que eliminarla no modifique el cierre de F. Se puede demostrar que el recubrimiento canónico F_c de F no contiene esa dependencia funcional rara. Supóngase que, por el contrario, esa dependencia rara estuviera en F_c. Los atributos del lado derecho de la dependencia serían raros, lo que no es posible por la definición de recubrimiento canónico.

Puede que el recubrimiento canónico no sea único. Por ejemplo, considérese el conjunto de dependencias funcionales $F = \{A \rightarrow BC, B \rightarrow AC \text{ y } C \rightarrow AB\}$. Si se aplica la prueba de rareza a $A \rightarrow BC$ se descubre que tanto B como C son raros bajo F. Sin embargo, sería incorrecto eliminar los dos. El algoritmo para hallar el recubrimiento canónico selecciona uno de los dos y lo elimina. Entonces:

1. Si se elimina C, se obtiene el conjunto $F' = \{A \rightarrow B, B \rightarrow AC \text{ y } C \rightarrow AB\}$. Ahora B ya no es raro en el lado derecho de $A \rightarrow B$ bajo F'. Siguiendo con el algoritmo se descubre que A y B son raros en el lado derecho de $C \rightarrow AB$, lo que genera dos recubrimientos canónicos:

$$F_c = \{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$$
$$F_c = \{A \rightarrow B, B \rightarrow AC, C \rightarrow B\}$$

2. Si se elimina B, se obtiene el conjunto $\{A \rightarrow C, B \rightarrow AC \text{ y } C \rightarrow AB\}$. Este caso es simétrico del anterior y genera dos recubrimientos canónicos:

$$F_c = \{A \rightarrow C, C \rightarrow B, B \rightarrow A\}$$
$$F_c = \{A \rightarrow C, B \rightarrow C, C \rightarrow AB\}$$

Como ejercicio, el lector debe intentar hallar otro recubrimiento canónico de F.

7.4.4 Descomposición sin pérdida

Sean R un esquema de relación y F un conjunto de dependencias funcionales de R. Supóngase que R_1 y R_2 forman una descomposición de R. Sea $r(R)$ una relación con el esquema R. Se dice que la descomposición es una **descomposición sin pérdidas** si, para todos los ejemplares legales de la base de datos (es decir, los ejemplares de la base de datos que satisfacen las dependencias funcionales especificadas y otras restricciones),

$$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

En otros términos, si se proyecta r sobre R_1 y R_2 y se calcula la reunión natural del resultado de la proyección, se vuelve a obtener exactamente r. Las descomposiciones que no son sin pérdidas se denominan **descomposiciones con pérdidas**. Los términos **descomposición de reunión sin pérdidas** y **descomposición de reunión con pérdidas** se utilizan a veces en lugar de descomposición sin pérdidas y descomposición con pérdidas.

Se pueden utilizar las dependencias funcionales para probar si ciertas descomposiciones son sin pérdidas. Sean R, R_1, R_2 y F como se acaban de definir. R_1 y R_2 forman una descomposición sin pérdidas de R si, como mínimo, una de las dependencias funcionales siguientes se halla en F^+:

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

En otros términos, si $R_1 \cap R_2$ forma una superclave de R_1 o de R_2, la descomposición de R es una descomposición sin pérdidas. Se puede utilizar el cierre de los atributos para buscar superclaves de manera eficiente, como ya se ha visto antes.

Para ilustrar esto, considérese el esquema

$$\textit{prestatario_préstamo} = (\underline{\textit{id_cliente, número_préstamo}}, \textit{importe})$$

que se descompuso en el Apartado 7.1.2 en

$$\textit{prestatario} = (\textit{id_cliente}, \underline{\textit{número_préstamo}})$$
$$\textit{préstamo} = (\underline{\textit{número_préstamo}}, \textit{importe})$$

```
calcular F+;
for each esquema Ri in D do
    begin
        Fi : = la restricción de F+ a Ri;
    end
F' := ∅
for each restricción Fi do
    begin
        F' = F' ∪ Fi
    end
calcular F'+;
if (F'+ = F+) then return (cierto)
              else return (falso);
```

Figura 7.11 Comprobación de la conservación de las dependencias.

En este caso *prestatario* ∩ *préstamo* = *número_préstamo* y *número_préstamo* → *importe*, lo que satisface la regla de la descomposición sin pérdidas.

Para el caso general de la descomposición simultánea de un esquema en varios, la comprobación de la descomposición sin pérdidas es más complicada. Véanse las notas bibliográficas para tener referencias sobre este tema.

Mientras que la comprobación de la descomposición binaria es, claramente, una condición suficiente para la descomposición sin pérdidas, sólo se trata de una condición necesaria si todas las las restricciones son dependencias funcionales. Más adelante se verán otros tipos de restricciones (especialmente, un tipo de restricción denominada dependencia multivalorada, que se estudia en el Apartado 7.6.1), que pueden garantizar que una descomposición sea sin pérdidas aunque no esté presente ninguna dependencia funcional.

7.4.5 Conservación de las dependencias

Resulta más sencillo caracterizar la conservación de las dependencias empleando la teoría de las dependencias funcionales que mediante el enfoque ad hoc que se utilizó en el Apartado 7.3.3.

Sea F un conjunto de dependencias funcionales del esquema R y $R_1, R_2, \ldots, R_n$ una descomposición de R. La **restricción** de F a R_i es el conjunto F_i de todas las dependencias funcionales de F^+ que *sólo* incluyen atributos de R_i. Dado que todas las dependencias funcionales de cada restricción implican a los atributos de un único esquema de relación, es posible comprobar el cumplimiento de esas dependencias verificando sólo una relación.

Obsérvese que la definición de restricción utiliza todas las dependencias de F^+, no sólo las de F. Por ejemplo, supóngase que se tiene $F = \{A \rightarrow B, B \rightarrow C\}$ y que se tiene una descomposición en AC y AB. La restricción de F a AC es, por tanto, $A \rightarrow C$, ya que $A \rightarrow C$ se halla en F^+, aunque no se halle en F.

El conjunto de restricciones $F_1, F_2, \ldots, F_n$ es el conjunto de dependencias que pueden comprobarse de manera eficiente. Ahora cabe preguntarse si es suficiente comprobar sólo las restricciones. Sea $F' = F_1 \cup F_2 \cup \cdots \cup F_n$. F' es un conjunto de dependencias funcionales del esquema R pero, en general $F' \neq F$. Sin embargo, aunque $F' \neq F$, puede ocurrir que $F'^+ = F^+$. Si esto último es cierto, entonces, todas las dependencias de F están implicadas lógicamente por F' y, si se comprueba que se satisface F', se habrá comprobado que se satisface F. Se dice que las descomposiciones que tienen la propiedad $F'^+ = F^+$ son **descomposiciones que conservan las dependencias**.

La Figura 7.11 muestra un algoritmo para la comprobación de la conservación de las dependencias. La entrada es el conjunto $D = \{R_1, R_2, \ldots, R_n\}$ de esquemas de relaciones descompuestas y el conjunto F de dependencias funcionales. Este algoritmo resulta costoso, ya que exige el cálculo de F^+. En lugar de aplicar el algoritmo de la Figura 7.11, se consideran dos alternativas.

En primer lugar, hay que tener en cuenta que si se puede comprobar cada miembro de F en una de las relaciones de la descomposición, la descomposición conserva las dependencias. Se trata de una manera sencilla de demostrar la conservación de las dependencias; no obstante, no funciona siempre. Hay casos en los que, aunque la descomposición conserve las dependencias, hay alguna dependencia de F que no

se puede comprobar para ninguna relación de la descomposición. Por tanto, esta prueba alternativa sólo se puede utilizar como condición suficiente que es fácil de comprobar; si falla, no se puede concluir que la descomposición no conserve las dependencias; en vez de eso, hay que aplicar la prueba general.

A continuación se da una comprobación alternativa de la conservación de las dependencias, que evita tener que calcular F^+. La idea intuitiva subyacente a esta comprobación se explicará tras presentarla. La comprobación aplica el procedimiento siguiente a cada $\alpha \rightarrow \beta$ de F.

resultado = α
while (cambios en *resultado*) **do**
    for each R_i de la descomposición
        t = (*resultado* $\cap$ R_i)$^+$ $\cap$ R_i
        resultado = *resultado* $\cup$ t

En este caso, el cierre de los atributos se halla bajo el conjunto de dependencias funcionales F. Si *resultado* contiene todos los atributos de β, se conserva la dependencia funcional $\alpha \rightarrow \beta$. La descomposición conserva las dependencias si, y sólo si, el procedimiento prueba que se conservan todas las dependencias de F.

Las dos ideas claves subyacentes a la comprobación anterior son las siguientes.

- La primera idea es comprobar cada dependencia funcional $\alpha \rightarrow \beta$ de F para ver si se conserva en F' (donde F' es tal y como se define en la Figura 7.11). Para ello, se calcula el cierre de α bajo F'; la dependencia se conserva exactamente cuando el cierre incluye a β. La descomposición conserva la dependencia si (y sólo si) se comprueba que se conservan todas las dependencias de F.
- La segunda idea es emplear una forma modificada del algoritmo de cierre de atributos para calcular el cierre bajo F', sin llegar a calcular antes F'. Se desea evitar el cálculo de F', ya que resulta bastante costoso. Téngase en cuenta que F' es la unión de las F_i, donde F_i es la restricción de F sobre R_i. El algoritmo calcula el cierre de los atributos de (*resultado* $\cap$ R_i) con respecto a F, intersecta el cierre con R_i y añade el conjunto de atributos resultante a *resultado*; esta secuencia de pasos es equivalente al cálculo del cierre de *resultado* bajo F_i. La repetición de este paso para cada *i* del interior del bucle mientras genera el cierre de *resultado* bajo F'.

  Para comprender el motivo de que este enfoque modificado al cierre de atributos funcione correctamente, hay que tener en cuenta que para cualquier $\gamma \subseteq R_i$, $\gamma \rightarrow \gamma^+$ es una dependencia funcional de F^+, y que $\gamma \rightarrow \gamma^+ \cap R_i$ es una dependencia funcional que se halla en F_i, la restricción de F^+ a R_i. A la inversa, si $\gamma \rightarrow \delta$ estuvieran en F_i, δ sería un subconjunto de $\gamma^+ \cap R_i$.

Esta comprobación tarda un tiempo que es polinómico, en lugar del exponencial necesario para calcular F^+.

7.5 Algoritmos de descomposición

Los esquemas de bases de datos del mundo real son mucho mayores que los ejemplos que caben en las páginas de un libro. Por este motivo, hacen falta algoritmos para la generación de diseños que se hallen en la forma normal adecuada. En este apartado se presentan algoritmos para la FNBC y para la 3NF.

7.5.1 Descomposición en la FNBC

Se puede emplear la definición de la FNBC para comprobar directamente si una relación se halla en esa forma normal. Sin embargo, el cálculo de F^+ puede resultar una tarea tediosa. En primer lugar se van a describir pruebas simplificadas para verificar si una relación dada se halla en la FNBC. En caso de que no lo esté, se puede descomponer para crear relaciones que sí estén en la FNBC. Más avanzado este apartado se describirá un algoritmo para crear descomposiciones sin pérdidas de las relaciones, de modo que esas descomposiciones se hallen en la FNBC.

7.5.1.1 Comprobación de la FNBC

La comprobación de relaciones para ver si satisfacen la FNBC se puede simplificar en algunos casos:

- Para comprobar si la dependencia no trivial $\alpha \rightarrow \beta$ provoca alguna violación de la FNBC hay que calcular α^+ (el cierre de los atributos de α) y comprobar si incluye todos los atributos de R; es decir, si es superclave de R.
- Para comprobar si el esquema de relación R se halla en la FNBC basta con comprobar sólo si las dependencias del conjunto F dado violan la FNBC, en vez de comprobar todas las dependencias de F^+.

  Se puede probar que, si ninguna de las dependencias de F provoca violaciones de la FNBC, ninguna de las dependencias de F^+ lo hace tampoco.

Por desgracia, el último procedimiento no funciona cuando se descomponen relaciones. Es decir, en las descomposiciones *no* basta con emplear F cuando se comprueba si la relación R_i de R viola la FNBC. Por ejemplo, considérese el esquema de relación $R\ (A, B, C, D, E)$, con las dependencias funcionales F que contienen $A \rightarrow B$ and $BC \rightarrow D$. Supóngase que se descompusiera en $R_1(A, B)$ y en $R_2(A, C, D, E)$. Ahora ninguna de las dependencias de F contiene únicamente atributos de (A, C, D, E), por lo que se podría creer erróneamente que R_2 satisface la FNBC. De hecho, hay una dependencia $AC \rightarrow D$ de F^+ (que se puede inferir empleando la regla de la pseudotransitividad con las dos dependencias de F), que prueba que R_2 no se halla en la FNBC. Por tanto, puede que haga falta una dependencia que esté en F^+, pero no en F, para probar que una relación descompuesta no se halla en la FNBC.

Una comprobación alternativa de la FNBC resulta a veces más sencilla que calcular todas las dependencias de F^+. Para comprobar si una relación R_i de una descomposición de R se halla en la FNBC, se aplica esta comprobación:

- Para cada subconjunto α de atributos de R_i se comprueba que α^+ (el cierre de los atributos de α bajo F) no incluye ningún atributo de $R_i - \alpha$ o bien incluye todos los atributos de R_i.

Si algún conjunto de atributos α de R_i viola esta condición, considérese la siguiente dependencia funcional, que se puede probar que se halla en F^+:

$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$.

La dependencia anterior muestra que R_i viola la FNBC.

7.5.1.2 Algoritmo de descomposición de la FNBC

Ahora se puede exponer un método general para descomponer los esquemas de relación de manera que satisfagan la FNBC. La Figura 7.12 muestra un algoritmo para esta tarea. Si R no está en la FNBC, se puede descomponer en un conjunto de esquemas en la FNBC, $R_1, R_2, \ldots, R_n$ utilizando este algoritmo. El algoritmo utiliza las dependencias que demuestran la violación de la FNBC para llevar a cabo la descomposición.

```
resultado := {R};
hecho := falso;
calcular F+;
while (not hecho) do
    if (hay algún esquema Ri de resultado que no se halle en la FNBC)
      then begin
              sea α → β una dependencia funcional no trivial que se cumple
              en Ri tal que α → Ri no se halla en F+, y α ∩ β = ∅ ;
              resultado := (result − Ri) ∪ (Ri − β) ∪ ( α, β);
           end
      else hecho := cierto;
```

Figura 7.12 Algoritmo de descomposición en la FNBC.

La descomposición que genera este algoritmo no sólo está en la FNBC, sino que también es una descomposición sin pérdidas. Para ver el motivo de que el algoritmo sólo genere descomposiciones sin pérdidas hay que darse cuenta de que, cuando se sustituye el esquema R_i por $((R_i - \beta)$ y (α, β), se cumple que $\alpha \rightarrow \beta$ y que $(R_i - \beta) \cap (\alpha, \beta) = \alpha$.

Si no se exigiera que $\alpha \cap \beta = \emptyset$, los atributos de $\alpha \cap \beta$ no aparecerían en el esquema $((R_i - \beta)$, y ya no se cumpliria la dependencia $\alpha \rightarrow \beta$.

Es fácil ver que la descomposición de *prestatario_préstamo* del Apartado 7.3.2 se obtiene de la aplicación del algoritmo. La dependencia funcional *número_préstamo* $\rightarrow$ *importe* satisface la condición $\alpha \cap \beta = \emptyset$ y, por tanto, se elige para descomponer el esquema.

El algoritmo de descomposición en la FNBC tarda un tiempo exponencial en relación con el tamaño del esquema inicial, ya que el algoritmo para comprobar si las relaciones de la descomposición satisfacen la FNBC puede tardar un tiempo exponencial. Las notas bibliográficas ofrecen referencias de un algoritmo que puede calcular la descomposición en la FNBC en un tiempo polinómico. Sin embargo, puede que ese algoritmo "sobrenormalice", es decir, descomponga relaciones sin que sea necesario.

A modo de ejemplo, más prolijo, del empleo del algoritmo de descomposición en la FNBC, supóngase que se tiene un diseño de base de datos que emplea el siguiente esquema *empréstito*:

empréstito = (*nombre_sucursal*, *ciudad_sucursal*, *activos*, *nombre_cliente*, *número_préstamo*, *importe*)

El conjunto de dependencias funcionales que se exige que se cumplan en *empréstito* es

nombre_sucursal $\rightarrow$ *activos ciudad_sucursal*
número_préstamo $\rightarrow$ *importe nombre_sucursal*

Una clave candidata para este esquema es {*número_préstamo*, *nombre_cliente*}.

Se puede aplicar el algoritmo de la Figura 7.12 al ejemplo *empréstito* de la manera siguiente:

- La dependencia funcional

  nombre_sucursal $\rightarrow$ *activos ciudad_sucursal*

  se cumple, pero *nombre_sucursal* no es superclave. Por tanto, *empréstito* no se halla en la FNBC. Se sustituye *empréstito* por

  sucursal = (*nombre_sucursal*, *ciudad_sucursal*, *activos*)
  info_préstamo = (*nombre_sucursal*, *nombre_cliente*, *número_préstamo*, *importe*)

- Entre las pocas dependencias funcionales no triviales que se cumplen en *sucursal* está *nombre_sucursal*, al lado izquierdo de la flecha. Dado que *nombre_sucursal* es clave de *sucursal*, la relación *sucursal* se halla en la FNBC.

- La dependencia funcional

  número_préstamo $\rightarrow$ *importe nombre_sucursal*

  se cumple en *info_préstamo*, pero *número_préstamo* no es clave de *info_préstamo*. Se sustituye *info_préstamo* por

  sucursal_préstamo = (*número_préstamo*, *nombre_sucursal*, *importe*)
  prestatario = (*nombre_cliente*, *número_préstamo*)

- *sucursal_préstamo* y *prestatario* están en la FNBC.

Por tanto, la descomposición de *empréstito* da lugar a los tres esquemas de relación *sucursal*, *sucursal_préstamo* y *prestatario*, cada uno de los cuales se halla en la FNBC. Se puede comprobar que la descomposición es sin pérdidas y que conserva las dependencias.

```
sea F_c un recubrimiento canónico de F;
i := 0;
for each dependencia funcional α → β de F_c do
   if ninguno de los esquemas R_j, j = 1, 2, ..., i contiene α β
      then begin
              i := i + 1;
              R_i := α β;
           end
if ninguno de los esquemas R_j, j = 1, 2, ..., i contiene una clave candidata de R
   then begin
           i := i + 1;
           R_i := cualquier clave candidata de R;
      end
return (R_1, R_2, ..., R_i)
```

Figura 7.13 Descomposición en la 3NF sin pérdidas que conserva las dependencias.

Téngase en cuenta que, aunque el esquema *sucursal_préstamo* anterior se halle en la FNBC, se puede decidir descomponerlo aún más empleando la dependencia funcional *número_préstamo* → *importe* para obtener los esquemas

préstamo = (*número_préstamo*, *importe*) *sucursal_préstamo* = (*número_préstamo*, *nombre_sucursal*)

Estos esquemas corresponden a los que se han utilizado en este capítulo.

7.5.2 Descomposición en la 3FN

La Figura 7.13 muestra un algoritmo para la búsqueda de descomposiciones en la 3FN sin pérdidas y que conserven las dependencias. El conjunto de dependencias F_c utilizado en el algoritmo es un recubrimiento canónico de F. Obsérvese que el algoritmo considera el conjunto de esquemas $R_j, j = 1, 2, \ldots, i$; inicialmente $i = 0$ y, en ese caso, el conjunto está vacío.

Se aplicará este algoritmo al ejemplo del Apartado 7.3.3 en el que se demostró que

sucursal_asesor = (*id_cliente*, *id_empleado*, *nombre_sucursal*, *tipo*)

se halla en la 3NF, aunque no se halle en la FNBC. El algoritmo utiliza las dependencias funcionales de F, que, en este caso, también son F_c:

(*id_cliente*, *id_empleado* → *nombre_sucursal*, *tipo*)
id_empleado → *nombre_sucursal*

y considera dos esquemas en el bucle **for**. A partir de la primera dependencia funcional el algoritmo genera como R_1 el esquema (*id_cliente*, *id_empleado*, *nombre_sucursal*, *tipo*). A partir de la segunda, el algoritmo genera el esquema (*id_empleado*, *nombre_sucursal*), pero no lo crea como R_2, ya que se halla incluido en R_1 y, por tanto, falla la condición **if**.

El algoritmo garantiza la conservación de las dependencias mediante la creación explícita de un esquema para cada dependencia del recubrimiento canónico. Asegura que la descomposición sea sin pérdidas al garantizar que, como mínimo, un esquema contenga una clave candidata del esquema que se está descomponiendo. El Ejercicio práctico 7.12 proporciona algunos indicios de la prueba de que esto basta para garantizar una descomposición sin pérdidas.

Este algoritmo también se denomina **algoritmo de síntesis de la 3NF**, ya que toma un conjunto de dependencias y añade los esquemas de uno en uno, en lugar de descomponer el esquema inicial de manera repetida. El resultado no queda definido de manera única, ya que cada conjunto de dependencias funcionales puede tener más de un recubrimiento canónico y, además, en algunos casos, el resultado del algoritmo depende del orden en que considere las dependencias de F_c.

Si una relación R_i está en la descomposición generada por el algoritmo de síntesis, entonces R_i está en la 3FN. Recuérdese que, cuando se busca la 3FN, basta con considerar las dependencias funcionales cuyo lado derecho sea un solo atributo. Por tanto, para ver si R_i está en la 3FN, hay que convencerse de que cualquier dependencia funcional $\gamma \rightarrow B$ que se cumpla en R_i satisface la definición de la 3FN.

Supóngase que la dependencia que generó R_i en el algoritmo de síntesis es $\alpha \rightarrow \beta$. Ahora bien, B debe estar en α o en β, ya que B está en R_i y $\alpha \rightarrow \beta$ ha generado R_i. Considérense los tres casos posibles:

- B está tanto en α como en β. En ese caso, la dependencia $\alpha \rightarrow \beta$ no habría estado en F_c, ya que B sería rara en β. Por tanto, este caso no puede darse.
- B está en β pero no en α. Considérense dos casos:
  - γ es superclave. Se satisface la segunda condición de la 3FN.
  - γ no es superclave. Entonces, α debe contener algún atributo que no se halle en γ. Ahora bien, como $\gamma \rightarrow B$ se halla en F^+, debe poder obtenerse a partir de F_c mediante el algoritmo del cierre de atributos de γ. La obtención no puede haber empleado $\alpha \rightarrow \beta$—si lo hubiera hecho, α debería estar contenida en el cierre de los atributos de γ, lo que no es posible, ya que se ha dado por supuesto que γ no es superclave. Ahora bien, empleando $\alpha \rightarrow (\beta - \{B\})$ y $\gamma \rightarrow B$, se puede obtener que $\alpha \rightarrow B$ (debido a que $\gamma \subseteq \alpha\beta$ y a que γ no puede contener a B porque $\gamma \rightarrow B$ no es trivial). Esto implicaría que B es raro en el lado derecho de $\alpha \rightarrow \beta$, lo que no es posible, ya que $\alpha \rightarrow \beta$ está en el recubrimiento canónico F_c. Por tanto, si B está en β, γ debe ser superclave, y se debe satisfacer la segunda condición de la 3FN.
- B está en α pero no en β.
  Como α es clave candidata, se satisface la tercera alternativa de la definición de la 3FN.

Es interesante que el algoritmo que se ha descrito para la descomposición en la 3FN pueda implementarse en tiempo polinómico, aunque la comprobación de una relación dada para ver si satisface la 3FN sea NP-dura (lo que significa que es muy improbable que se invente nunca un algoritmo de tiempo polinómico para esta tarea).

7.5.3 Comparación de la FNBC y la 3FN

De las dos formas normales para los esquemas de las bases de datos relacionales, la 3FN y la FNBC, la 3FN es más conveniente, ya que se sabe que siempre es posible obtener un diseño en la 3FN sin sacrificar la ausencia de pérdidas ni la conservación de las dependencias. Sin embargo, la 3FN presenta inconvenientes: puede que haya que emplear valores nulos para representar algunas de las relaciones significativas posibles entre los datos y existe el problema de la repetición de la información.

Los objetivos del diseño de bases de datos con dependencias funcionales son:

1. FNBC
2. Ausencia de pérdidas
3. Conservación de las dependencias

Como no siempre resulta posible satisfacer las tres, puede que nos veamos obligados a escoger entre la FNBC y la conservación de las dependencias con la 3FN.

Merece la pena destacar que el SQL no ofrece una manera de especificar las dependencias funcionales, salvo para el caso especial de la declaración de las superclaves mediante las restricciones **primary key** o **unique**. Es posible, aunque un poco complicado, escribir asertos que hagan que se cumpla una dependencia funcional determinada (véase el Ejercicio práctico 7.9); por desgracia, la comprobación de los asertos resultaría muy costosa en la mayor parte de los sistemas de bases de datos. Por tanto, aunque se tenga una descomposición que conserve las dependencias, si se utiliza el SQL estándar, sólo se podrán comprobar de manera eficiente las dependencias funcionales cuyo lado izquierdo sea una clave.

Aunque puede que la comprobación de las dependencias funcionales implique una reunión si la descomposición no conserva las dependencias, se puede reducir su coste empleando vistas materializadas, que se pueden utilizar en la mayor parte de los sistemas de bases de datos. Dada una descomposición en la FNBC que no conserve las dependencias, se considera cada dependencia de un recubrimiento mínimo F_c que no se conserva en la descomposición. Para cada una de esas dependencias $\alpha \rightarrow \beta$, se define una vista materializada que calcula una reunión de todas las relaciones de la descomposición y proyecta el resultado sobre $\alpha\beta$. La dependencia funcional puede comprobarse fácilmente en la vista materializada

mediante una restricción **unique**(α). La parte negativa es que hay una sobrecarga espacial y temporal debida a la vista materializada, pero la positiva es que el programador de la aplicación no tiene que preocuparse de escribir código para hacer que los datos redundantes se conserven consistentes en las actualizaciones; es labor del sistema de bases de datos conservar la vista materializada, es decir, mantenerla actualizada cuando se actualice la base de datos (más adelante, en el Apartado 14.5, se describe el modo en que el sistema de bases de datos puede llevar a cabo de manera eficiente el mantenimiento de las vistas materializadas).

Por tanto, en caso de que no se pueda obtener una descomposición en la FNBC que conserve las dependencias, suele resultar preferible optar por la 3NF y emplear técnicas como las vistas materializadas para reducir el coste de la comprobación de las dependencias funcionales.

7.6 Descomposición mediante dependencias multivaloradas

No parece que algunos esquemas de relación, aunque se hallen en la FNBC, estén suficientemente normalizados, en el sentido de que siguen sufriendo el problema de la repetición de información. Considérese nuevamente el ejemplo bancario. Supóngase que, en un diseño alternativo del esquema de la base de datos bancaria, se tiene el esquema

$$\textit{préstamo_cliente} = (\underline{\textit{número_préstamo, id_cliente}}, \textit{nombre_cliente, calle_cliente, ciudad_cliente})$$

El lector avispado reconocerá este esquema como no correspondiente a la FNBC, debido a la dependencia funcional

$$\textit{id_cliente} \rightarrow \textit{nombre_cliente, calle_cliente ciudad_cliente}$$

y a que *id_cliente* no es clave de *préstamo_cliente*. No obstante, supóngase que el banco está atrayendo clientes ricos que tienen varios domicilios (por ejemplo, una residencia de invierno y otra de verano). Entonces, ya no se desea que se cumpla la dependencia funcional *id_cliente* →*calle_cliente ciudad_cliente*, aunque, por supuesto, se sigue deseando que se cumpla *id_cliente* → *nombre_cliente* (es decir, el banco no trata con clientes que operan con varios alias). A partir del algoritmo de descomposición en la FNBC se obtienen dos esquemas:

$$R_1 = (\underline{\textit{id_cliente}}, \textit{nombre_cliente})$$
$$R_2 = (\underline{\textit{número_préstamo, id_cliente}}, \textit{calle_cliente, ciudad_cliente})$$

Los dos se encuentran en la FNBC (recuérdese que no sólo puede cada cliente tener concedido más de un préstamo, sino que también se pueden conceder préstamos a grupos de personas y, por tanto, no se cumplen ni *id_cliente* → *número_préstamo* ni *número_préstamo* → *id_cliente*).

Pese a que R_2 se halla en la FNBC, hay redundancia. Se repite la dirección de cada residencia de cada cliente una vez por cada préstamo que tenga concedido ese cliente. Este problema se puede resolver descomponiendo más aún R_2 en:

id_cliente_préstamo = (*número_préstamo, id_cliente*)
residencia_cliente = (*id_cliente, calle_cliente, ciudad_cliente*)

pero no hay ninguna restricción que nos lleve a hacerlo.

Para tratar este problema hay que definir una nueva modalidad de restricción, denominada *dependencia multivalorada*. Al igual que se hizo con las dependencias funcionales, se utilizarán las dependencias multivaloradas para definir una forma normal para los esquemas de relación. Esta forma normal, denominada **cuarta forma normal** (4FN), es más restrictiva que la FNBC. Se verá que cada esquema en la 4FN también se halla en la FNBC, pero que hay esquemas en la FNBC que no se hallan en la 4FN.

7.6.1 Dependencias multivaloradas

Las dependencias funcionales impiden que ciertas tuplas estén en una relación dada. Si $A \rightarrow B$, entonces no puede haber dos tuplas con el mismo valor de A y diferentes valores de B. Las dependencias

	α	β	$R - \alpha - \beta$
t_1	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
t_2	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
t_3	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
t_4	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$

Figura 7.14 Representación tabular de $\alpha \twoheadrightarrow \beta$.

multivaloradas, por otro lado, no impiden la existencia de esas tuplas. Por el contrario, *exigen* que estén presentes en la relación otras tuplas de una forma determinada. Por este motivo, las dependencias funcionales se denominan a veces **dependencias que generan igualdades**; y las dependencias multivaloradas, **dependencias!que generan tuplas**.

Sea R un esquema de relación y sean $\alpha \subseteq R$ y $\beta \subseteq R$. La **dependencia multivalorada**

$$\alpha \twoheadrightarrow \beta$$

se cumple en R si, en cualquier relación legal $r(R)$ para todo par de tuplas t_1 y t_2 de r tales que $t_1[\alpha] = t_2[\alpha]$ existen unas tuplas t_3 y t_4 de r tales que

$$\begin{aligned} t_1[\alpha] &= t_2[\alpha] = t_3[\alpha] = t_4[\alpha] \\ t_3[\beta] &= t_1[\beta] \\ t_3[R - \beta] &= t_2[R - \beta] \\ t_4[\beta] &= t_2[\beta] \\ t_4[R - \beta] &= t_1[R - \beta] \end{aligned}$$

Esta definición es menos complicada de lo que parece. La Figura 7.14 muestra una representación tabular de t_1, t_2, t_3 y t_4. De manera intuitiva, la dependencia multivalorada $\alpha \twoheadrightarrow \beta$ indica que la relación entre α y β es independiente de la relación entre α y $R - \beta$. Si todas las relaciones del esquema R satisfacen la dependencia multivalorada $\alpha \twoheadrightarrow \beta$, entonces $\alpha \twoheadrightarrow \beta$ es una dependencia multivalorada *trivial* del esquema R. Por tanto, $\alpha \twoheadrightarrow \beta$ es trivial si $\beta \subseteq \alpha$ o $\beta \cup \alpha = R$.

Para ilustrar la diferencia entre las dependencias funcionales y las multivaloradas, considérense de nuevo el esquema R_2 y la relación de ejemplo de ese esquema mostrada en la Figura 7.15. Hay que repetir el número de préstamo una vez por cada dirección que tenga el cliente, y la dirección en cada préstamo que tenga concedido ese cliente. Esta repetición es innecesaria, ya que la relación entre cada cliente y su dirección es independiente de la relación entre ese cliente y el préstamo. Si un cliente con identificador de cliente 99123 tiene concedido un préstamo (por ejemplo, el préstamo número P-23), se desea que ese préstamo esté asociado con todas las direcciones de ese cliente. Por tanto, la relación de la Figura 7.16 es ilegal. Para hacer que esa relación sea legal hay que añadir a la relación de la Figura 7.16 las tuplas (P-23, 99123, Mayor, Chinchón) y (P-27, 99123, Carretas, Cerceda).

Si se compara el ejemplo anterior con la definición de dependencia multivalorada, se ve que se desea que se cumpla la dependencia multivalorada

id_cliente $\twoheadrightarrow$ *calle_cliente ciudad_cliente*

También se cumplirá la dependencia multivalorada *id_cliente* $\twoheadrightarrow$ *número_préstamo*. Pronto se verá que son equivalentes.

Al igual que las dependencias funcionales, las dependencias multivaloradas se utilizan de dos maneras:

número_préstamo	*id_cliente*	*calle_cliente*	*ciudad_cliente*
P-23	99123	Carretas	Cerceda
P-23	99123	Mayor	Chinchón
P-93	15106	Leganitos	Aluche

Figura 7.15 Ejemplo de redundancia en un esquema en la FNBC.

número_préstamo	*id_cliente*	*calle_cliente*	*ciudad_cliente*
P-23	99123	Carretas	Cerceda
P-27	99123	Mayor	Chinchón

Figura 7.16 La relación ilegal R_2.

1. Para verificar las relaciones y determinar si son legales bajo un conjunto dado de dependencias funcionales y multivaloradas.
2. Para especificar restricciones del conjunto de relaciones legales; de este modo, *sólo* habrá que preocuparse de las relaciones que satisfagan un conjunto dado de dependencias funcionales y multivaloradas.

Téngase en cuenta que, si una relación *r* no satisface una dependencia multivalorada dada, se puede crear una relación *r'* que *sí* la satisfaga añadiendo tuplas a *r*.

Supóngase que *D* denota un conjunto de dependencias funcionales y multivaloradas. El **cierre** D^+ de *D* es el conjunto de todas las dependencias funcionales y multivaloradas implicadas lógicamente por *D*. Al igual que se hizo con las dependencias funcionales, se puede calcular D^+ a partir de *D*, empleando las definiciones formales de dependencia funcional y multivalorada. Con este razonamiento se puede trabajar con dependencias multivaloradas muy sencillas. Afortunadamente, parece que las dependencias multivaloradas que se dan en la práctica son bastante sencillas. Para dependencias complejas es mejor razonar con conjuntos de dependencias mediante un sistema de reglas de inferencia (el Apartado C.1.1 del apéndice describe un sistema de reglas de inferencia para las dependencias multivaloradas).

A partir de la definición de dependencia multivalorada se puede obtener la regla siguiente:

- Si $\alpha \rightarrow \beta$, entonces $\alpha \twoheadrightarrow \beta$.

En otras palabras, todas las dependencias funcionales son también dependencias multivaloradas.

7.6.2 Cuarta forma normal

Considérese nuevamente el ejemplo del esquema en la FNBC

$$R_2 = (\underline{\textit{número_préstamo, id_cliente}}, \textit{calle_cliente, ciudad_cliente})$$

en el que se cumple la dependencia multivalorada *id_cliente* $\twoheadrightarrow$ *calle_cliente ciudad_cliente*. Se vio en los primeros párrafos del Apartado 7.6 que, aunque este esquema se halla en la FNBC, el diseño no es el ideal, ya que hay que repetir la información sobre la dirección del cliente para cada préstamo. Se verá que se puede utilizar esta dependencia multivalorada para mejorar el diseño de la base de datos, realizando la descomposición del esquema en la **cuarta forma normal**.

Un esquema de relación R está en la **cuarta forma normal** (4FN) con respecto a un conjunto D de dependencias funcionales y multivaloradas si, para todas las dependencias multivaloradas de D^+ de la forma $\alpha \twoheadrightarrow \beta$, donde $\alpha \subseteq R$ y $\beta \subseteq R$, se cumple, como mínimo, una de las condiciones siguientes

- $\alpha \twoheadrightarrow \beta$ es una dependencia multivalorada trivial.
- α es superclave del esquema R.

El diseño de una base de datos está en la 4FN si cada componente del conjunto de esquemas de relación que constituye el diseño se halla en la 4FN.

Téngase en cuenta que la definición de la 4FN sólo se diferencia de la definición de la FNBC en el empleo de las dependencias multivaloradas en lugar de las dependencias funcionales. Todos los esquemas en la 4FN están en la FNBC. Para verlo hay que darse cuenta de que, si un esquema R no se halla en la FNBC, hay una dependencia funcional no trivial $\alpha \rightarrow \beta$ que se cumple en R en la que α no es superclave. Como $\alpha \rightarrow \beta$ implica $\alpha \twoheadrightarrow \beta$, R no puede estar en la 4FN.

```
resultado := {R};
hecho := falso;
calcular D+; dado el esquema Ri, Di denotará la restricción de D+ a Ri
while (not hecho) do
    if (hay un esquema Ri en resultado que no se halla en la 4FN con respecto a Di)
      then begin
              sea α ↠ β una dependencia multivalorada no trivial que se cumple
              en Ri tal que α → Ri no se halla en Di, y α ∩ β = ∅;
              resultado := (resultado − Ri) ∪ (Ri − β) ∪ (α, β);
           end
      else hecho := cierto;
```

Figura 7.17 Algoritmo de descomposición en la 4FN.

Sea R un esquema de relación y sean $R_1, R_2, \ldots, R_n$ una descomposición de R. Para comprobar si cada esquema de relación R_i de la descomposición se halla en la 4FN hay que averiguar las dependencias multivaloradas que se cumplen en cada R_i. Recuérdese que, para un conjunto F de dependencias funcionales, la restricción F_i de F a R_i son todas las dependencias funcionales de F^+ que *sólo* incluyen los atributos de R_i. Considérese ahora un conjunto D de dependencias funcionales y multivaloradas. La **restricción** de D a R_i es el conjunto D_i consistente en

1. Todas las dependencias funcionales de D^+ que sólo incluyen atributos de R_i
2. Todas las dependencias multivaloradas de la forma

$$\alpha \twoheadrightarrow \beta \cap R_i$$

donde $\alpha \subseteq R_i y \alpha \twoheadrightarrow \beta$ está en D^+.

7.6.3 Descomposición en la 4FN

La analogía entre la 4FN y la FNBC es aplicable al algoritmo para descomponer esquemas en la 4FN. La Figura 7.17 muestra el algoritmo de descomposición en la 4FN. Es idéntico al algoritmo de descomposición en la FNBC de la Figura 7.12, salvo en que emplea dependencias multivaloradas en lugar de funcionales y en que utiliza la restricción de D^+ a R_i.

Si se aplica el algoritmo de la Figura 7.17 a (*número_préstamo, id_cliente, calle_cliente, ciudad_cliente*), se descubre que *id_cliente* ↠ *número_préstamo* es una dependencia multivalorada no trivial, y que *id_cliente* no es superclave del esquema. De acuerdo con el algoritmo, se sustituye el esquema original por estos dos esquemas:

id_cliente_préstamo = (*número_préstamo*, *id_cliente*)
residencia_cliente = (*id_cliente*, *calle_cliente*, *ciudad_cliente*)

Este par de esquemas, que se hallan en la 4FN, elimina la redundancia que se encontró anteriormente.

Como ocurría cuando se trataba solamente con las dependencias funcionales, resultan interesantes las descomposiciones que carecen de pérdidas y que conservan las dependencias. La siguiente circunstancia relativa a las dependencias multivaloradas y a la ausencia de pérdidas muestra que el algoritmo de la Figura 7.17 sólo genera descomposiciones sin pérdidas:

- Sean R un esquema de relación y D un conjunto de dependencias funcionales y multivaloradas de R. Supóngase que R_1 y R_2 forman una descomposición de R. Esa descomposición de R carecerá de pérdidas si, y sólo si, como mínimo, una de las siguientes dependencias multivaloradas se halla en D^+:

$$R_1 \cap R_2 \twoheadrightarrow R_1$$
$$R_1 \cap R_2 \twoheadrightarrow R_2$$

Recuérdese que se afirmó en el Apartado 7.4.4 que, si $R_1 \cap R_2 \rightarrow R_1$ o $R_1 \cap R_2 \rightarrow R_2$, entonces R_1 y R_2 son descomposiciones de R sin pérdidas. Esta circunstancia relativa a las dependencias multivaloradas es una declaración más general sobre la carencia de pérdidas. Afirma que, para *toda* descomposición de R sin pérdidas en dos esquemas R_1 y R_2, debe cumplirse una de las dos dependencias $R_1 \cap R_2 \twoheadrightarrow R_1$ o $R_1 \cap R_2 \twoheadrightarrow R_2$.

El problema de la conservación de las dependencias al descomponer relaciones se vuelve más complejo en presencia de las dependencias multivaloradas. El Apartado C.1.2 del apéndice aborda este asunto.

7.7 Más formas normales

La cuarta forma normal no es, de ningún modo, la forma normal "definitiva". Como ya se ha visto, las dependencias multivaloradas ayudan a comprender y a abordar algunas formas de repetición de la información que no pueden comprenderse en términos de las dependencias funcionales. Hay restricciones denominadas **dependencias de reunión** que generalizan las dependencias multivaloradas y llevan a otra forma normal denominada **forma normal de reunión por proyección (FNRP)** (la FNRP se denomina en algunos libros **quinta forma normal**). Hay una clase de restricciones todavía más generales, que lleva a una forma normal denominada **forma normal de dominios y claves** (FNDC).

Un problema práctico del empleo de estas restricciones generalizadas es que no sólo es difícil razonar con ellas, sino que tampoco hay un conjunto de reglas de inferencia seguras y completas para razonar sobre las restricciones. Por tanto, FNRP y FNDC se utilizan muy rara vez. El Apéndice C ofrece más detalles sobre estas formas normales.

Destaca por su ausencia en este estudio de las formas normales la **segunda forma normal** (2FN). No se ha estudiado porque sólo es de interés histórico. Simplemente se definirá, y se permitirá al lector experimentar con ella, en el Ejercicio práctico 7.15.

7.8 Proceso de diseño de las bases de datos

Hasta ahora se han examinado aspectos detallados de las formas normales y de la normalización. En este apartado se estudiará el modo de encajar la normalización en el proceso global de diseño de las bases de datos.

Anteriormente, en este capítulo, a partir del Apartado 7.3, se ha dado por supuesto que se tenía un esquema de relación R y que se procedía a normalizarlo. Hay varios modos de obtener ese esquema R:

1. R puede haberse generado al convertir un diagrama E-R en un conjunto de esquemas de relación.
2. R puede haber sido una sola relación que contenía *todos* los atributos que resultaban de interés. El proceso de normalización divide a R en relaciones más pequeñas.
3. R puede haber sido el resultado de algún diseño ad hoc de las relaciones, que hay que comprobar para asegurarse de que satisface la forma normal deseada.

En el resto de este apartado se examinarán las implicaciones de estos enfoques. También se examinarán algunos aspectos prácticos del diseño de las bases de datos, incluida la desnormalización para el rendimiento y ejemplos de mal diseño que no son detectados por la normalización.

7.8.1 El modelo ER y la normalización

Cuando se definen con cuidado los diagramas E-R, identificando correctamente todas las entidades, los esquemas de relación generados a partir de ellos no deben necesitar mucha más normalización. No obstante, puede haber dependencias funcionales entre los atributos de alguna entidad. Por ejemplo, supóngase que la entidad *empleado* tiene los atributos *número_departamento* y *dirección_departamento*, y que hay una dependencia funcional *número_departamento* $\rightarrow$ *dirección_departamento*. Habrá que normalizar la relación generada a partir de *empleado*.

La mayor parte de los ejemplos de este tipo de dependencias surge de un mal diseño del diagrama E-R. En el ejemplo anterior, si se hubiera diseñado correctamente el diagrama E-R, se habría creado una entidad *departamento* con el atributo *dirección_departamento* y una relación entre *empleado* y *departamento*.

De manera parecida, puede que una relación que implique a más de dos entidades no se halle en la forma normal deseable. Como la mayor parte de las relaciones son binarias, estos casos resultan relativamente raros (de hecho, algunas variantes de los diagramas E-R hacen realmente difícil o imposible especificar relaciones no binarias).

Las dependencias funcionales pueden ayudar a detectar un mal diseño E-R. Si las relaciones generadas no se hallan en la forma normal deseada, el problema puede solucionarse en el diagrama E-R. Es decir, la normalización puede llevarse a cabo formalmente como parte del modelado de los datos. De manera alternativa, la normalización puede dejarse a la intuición del diseñador durante el modelado E-R, y puede hacerse formalmente sobre las relaciones generadas a partir del modelo E-R.

El lector atento se habrá dado cuenta de que para que se pudiera ilustrar la necesidad de las dependencias multivaloradas y de la cuarta forma normal hubo que comenzar con esquemas que no se obtuvieron a partir del diseño E-R. En realidad, el proceso de creación de diseños E-R tiende a generar diseños 4FN. Si se cumple alguna dependencia multivalorada y no la implica la dependencia funcional correspondiente, suele proceder de alguna de las fuentes siguientes:

- Una relación de varios a varios.
- Un atributo multivalorado de un conjunto de entidades.

En las relaciones de varios a varios cada conjunto de entidades relacionado tiene su propio esquema y hay un esquema adicional para el conjunto de relaciones. Para los atributos multivalorados se crea un esquema diferente que consta de ese atributo y de la clave primaria del conjunto de entidades (como en el caso del atributo *nombre_subordinado* del conjunto de entidades *empleado*).

El enfoque de las relaciones universales para el diseño de bases de datos relacionales parte de la suposición de que sólo hay un esquema de relación que contenga todos los atributos de interés. Este esquema único define la manera en que los usuarios y las aplicaciones interactúan con la base de datos.

7.8.2 Denominación de los atributos y de las relaciones

Una característica deseable del diseño de bases de datos es la **asunción de un rol único**, lo que significa que cada nombre de atributo tiene un significado único en toda la base de datos. Esto evita que se utilice el mismo atributo para indicar cosas diferentes en esquemas diferentes. Por ejemplo, puede que, de otra manera, se considerara el empleo del atributo *número* para el número de préstamo en el esquema *préstamo* y para el número de cuenta en el esquema *cuenta*. La reunión de una relación del esquema *préstamo* con otra de *cuenta* carece de significado ("información de los pares préstamo-cuenta en los que coincide el número de préstamo y el de cuenta"). Aunque los usuarios y los desarrolladores de aplicaciones pueden trabajar con esmero para garantizar el empleo del *número* correcto en cada circunstancia, tener nombres de atributo diferentes para el número de préstamo y para el de cuenta sirve para reducir los errores de los usuarios. De hecho, se ha observado la suposición de un rol único en los diseños de bases de datos de este libro, y su aplicación es una buena práctica general que conviene seguir.

Aunque es una buena idea hacer que los nombres de los atributos incompatibles sean diferentes, si los atributos de relaciones diferentes tienen el mismo significado, puede ser conveniente emplear el mismo nombre de atributo. Por ejemplo, se han utilizado los nombres de atributo *id_clente* e *id_empleado* en los conjuntos de entidades *cliente* y *empleado* (y en sus relaciones). Si deseáramos generalizar esos conjuntos de entidades mediante la creación del conjunto de entidades *persona*, habría que renombrar el atributo. Por tanto, aunque no se tenga actualmente una generalización de *cliente* y de *empleado*, si se prevé esa posibilidad es mejor emplear el mismo nombre en los dos conjuntos de relaciones (y en sus relaciones).

Aunque, técnicamente, el orden de los nombres de los atributos en los esquemas no tiene ninguna importancia, es costumbre relacionar en primer lugar los atributos de la clave primaria. Esto facilita la lectura de los resultados predeterminados (como los generados por **select ***).

En los esquemas de bases de datos de gran tamaño los conjuntos de relaciones (y los esquemas derivados) se suelen denominar mediante la concatenación de los nombres de los conjuntos de entidades a los que hacen referencia, quizás con guiones o caracteres de subrayado intercalados. En este libro se han utilizado algunos nombres de este tipo, por ejemplo, *sucursal_cuenta* y *sucursal_préstamo*. Se han utilizado los nombres *prestatario* o *impositor* en lugar de otros concatenados de mayor longitud como *cliente*

_préstamo o *cliente_cuenta*. Esto resultaba aceptable, ya que no es difícil recordar las entidades asociadas a unas pocas relaciones. No siempre se pueden crear nombres de relaciones mediante la mera concatenación; por ejemplo, la relación jefe o trabaja-para entre empleados no tendría mucho sentido si se llamara *empleado_empleado*. De manera parecida, si hay varios conjuntos de relaciones posibles entre un par de conjuntos de entidades, los nombres de las relaciones deben incluir componentes adicionales para identificar cada relación.

Las diferentes organizaciones tienen costumbres diferentes para la denominación de las entidades. Por ejemplo, a un conjunto de entidades de clientes se le puede denominar *cliente* o *clientes*. En los diseños de las bases de datos de este libro se ha decidido utilizar la forma singular. Es aceptable tanto el empleo del singular como el del plural, siempre y cuando la convención se utilice de manera consistente en todas las entidades.

A medida que los esquemas aumentan de tamaño, con un número creciente de relaciones, el empleo de una denominación consistente de los atributos, de las relaciones y de las entidades facilita mucho la vida de los diseñadores de bases de datos y de los programadores de aplicaciones.

7.8.3 Desnormalización para el rendimiento

A veces, los diseñadores de bases de datos escogen un esquema que tiene información redundante; es decir, que no está normalizado. Utilizan la redundancia para mejorar el rendimiento de aplicaciones concretas. La penalización sufrida por no emplear un esquema normalizado es el trabajo adicional (en términos de tiempos de codificación y de ejecución) de mantener consistentes los datos redundantes.

Por ejemplo, supóngase que hay que mostrar el nombre del titular junto con el número de cuenta y con el saldo cada vez que se accede a la cuenta. En el esquema normalizado esto exige una reunión de *cuenta* con *impositor*.

Una alternativa al cálculo de la reunión sobre la marcha es almacenar una relación que contenga todos los atributos de *cuenta* y de *impositor*. Esto hace más rápida la visualización de la información de la cuenta. Sin embargo, la información del saldo de la cuenta se repite para cada uno de los titulares, y la aplicación debe actualizar todas las copias cada vez que se actualice el saldo. El proceso de tomar un esquema normalizado y hacer que no esté normalizado se denomina **desnormalización**, y los diseñadores lo utilizan para ajustar el rendimiento de los sistemas para que den soporte a las operaciones críticas en el tiempo.

Una opción mejor, soportada hoy en día por muchos sistemas de bases de datos, es emplear el esquema normalizado y, además, almacenar la reunión de *cuenta* e *impositor* en forma de vista materializada (recuérdese que las vistas materializadas son vistas cuyo resultado se almacena en la base de datos y se actualiza cuando se actualizan las relaciones utilizadas en ellas). Al igual que la desnormalización, el empleo de las vistas materializadas supone sobrecargas de espacio y de tiempo; sin embargo, presenta la ventaja de que conservar actualizadas las vistas es labor del sistema de bases de datos, no del programador de la aplicación.

7.8.4 Otros problemas del diseño

Hay algunos aspectos del diseño de bases de datos que la normalización no aborda y, por tanto, pueden llevar a un mal diseño de la base de datos. Los datos relativos al tiempo o a intervalos temporales presentan varios de esos problemas. A continuación se ofrecen algunos ejemplos; evidentemente, conviene evitar esos diseños.

Considérese una base de datos empresarial, en la que se desea almacenar los beneficios de varias compañías a lo largo de varios años. Se puede utilizar la relación *beneficios* (*id_empresa, año, importe*) para almacenar la información sobre los beneficios. La única dependencia funcional de esta relación es *id_empresa, año* → *importe*, y se halla en la FNBC.

Un diseño alternativo es el empleo de varias relaciones, cada una de las cuales almacena los beneficios de un año diferente. Supóngase que los años que nos interesan son 2000, 2001 y 2002; se tendrán, entonces, relaciones de la forma *beneficios_2000*, *beneficios_2001* y *beneficios_2002*, todas las cuales se hallan en el esquema (*id_empresa, beneficios*). En este caso, la única dependencia funcional de cada relación será *id_empresa* → *beneficios*, por lo que esas relaciones también se hallan en la FNBC.

No obstante, este diseño alternativo es, claramente, una mala idea—habría que crear una relación nueva cada año, y también habría que escribir consultas nuevas todos los años, para tener en cuenta cada nueva relación. Las consultas también serían más complicadas, ya que probablemente tendrían que hacer referencia a muchas relaciones.

Otra manera de representar esos mismos datos es tener una sola relación *año_empresa* (*id_empresa*, *beneficios_2000*, *beneficios_2001*, *beneficios_2002*). En este caso, las únicas dependencias funcionales van de *id_empresa* hacia los demás atributos y, una vez más, la relación se halla en la FNBC. Este diseño también es una mala idea, ya que tiene problemas parecidos a los del diseño anterior—es decir, habría que modificar el esquema de la relación y escribir consultas nuevas cada año. Las consultas también serían más complicadas, ya que puede que tuvieran que hacer referencia a muchos atributos.

Las representaciones como las de la relación *año_empresa*, con una columna para cada valor de cada atributo, se denominan de **referencias cruzadas**; se emplean mucho en las hojas de cálculo, en los informes y en las herramientas de análisis de datos. Aunque esas representaciones resultan útiles para mostrárselas a los usuarios, por las razones que se acaban de dar, no resultan deseables en el diseño de bases de datos. Se han propuesto extensiones del SQL para pasar los datos de la representación relacional normal a la de referencias cruzadas, para su visualización.

7.9 Modelado de datos temporales

Supóngase que en el banco se conservan datos que no sólo muestran la dirección actual de cada cliente, sino también todas las direcciones anteriores de las que el banco tenga noticia. Se pueden formular consultas como "Averiguar todos los clientes que vivían en Vigo en 1981". En ese caso, puede que se tengan varias direcciones por cliente. Cada dirección tiene asociadas una fecha de comienzo y otra de finalización, que indican el periodo en que el cliente residió en esa dirección. Se puede utilizar un valor especial para la fecha de finalización, por ejemplo, nulo, o un valor que se halle claramente en el futuro, como 31/12/9999, para indicar que el cliente sigue residiendo en esa dirección.

En general, los **datos temporales** son datos que tienen asociado un intervalo de tiempo durante el cual son **válidos**[4]. Se utiliza el término **instantánea** de los datos para indicar su valor en un momento determinado. Por tanto, una instantánea de los datos de los clientes muestra el valor de todos los atributos de los clientes, como la dirección, en un momento concreto.

El modelado de datos temporales es una cuestión interesante por varios motivos. Por ejemplo, supóngase que se tiene una entidad *cliente* con la que se desea asociar una dirección que varía con el tiempo. Para añadir información temporal a una dirección hay que crear un atributo multivalorado, cada uno de cuyos valores es un valor compuesto que contiene una dirección y un intervalo de tiempo. Además de los valores de los atributos que varían con el tiempo, puede que las propias entidades tengan un periodo de validez asociado. Por ejemplo, la entidad cuenta puede tener un periodo de validez desde la fecha de apertura hasta la de cancelación. Las relaciones también pueden tener asociados periodos de validez. Por ejemplo, la relación *impositor* entre un cliente y una cuenta puede registrar el momento en que el cliente pasó a ser titular de la cuenta. Por tanto, habría que añadir intervalos de validez a los valores de los atributos, a las entidades y a las relaciones. La adición de esos detalles a los diagramas E-R hace que sean muy difíciles de crear y de comprender. Ha habido varias propuestas para extender la notación E-R para que especifique de manera sencilla que un atributo o una relación varía con el tiempo, pero no hay ninguna norma aceptada al respecto.

Cuando se realiza un seguimiento del valor de los datos a lo largo del tiempo, puede que dejen de cumplirse dependencias funcionales que se suponía que se cumplían, como

$$id_cliente \rightarrow calle_cliente\ ciudad_cliente$$

En su lugar, se cumpliría la restricción siguiente (expresada en castellano): "*id_cliente* sólo tiene un valor de *calle_cliente* y de *ciudad_cliente* para cada momento *t* dado".

Las dependencias funcionales que se cumplen en un momento concreto se denominan dependencias funcionales temporales. Formalmente, la **dependencia funcional temporal** $X \xrightarrow{T} Y$ se cumple en el

4. Hay otros modelos de datos temporales que distinguen entre **periodo de validez** y **momento de la transacción**; el último registra el momento en que se registró un hecho en la base de datos. Para simplificar se prescinde de estos detalles.

esquema de relación R si, para todos los ejemplares legales r de R, todas las instantáneas de r satisfacen la dependencia funcional $X \rightarrow Y$.

Se puede extender la teoría del diseño de bases de datos relacionales para que tenga en cuenta las dependencias funcionales temporales. Sin embargo, el razonamiento con las dependencias funcionales normales ya resulta bastante difícil, y pocos diseñadores están preparados para trabajar con las dependencias funcionales temporales.

En la práctica, los diseñadores de bases de datos recurren a enfoques más sencillos para diseñar las bases de datos temporales. Un enfoque empleado con frecuencia es diseñar toda la base de datos (incluidos los diseños E-R y relacional) ignorando las modificaciones temporales (o, lo que es lo mismo, tomando sólo una instantánea en consideración). Tras esto, el diseñador estudia las diferentes relaciones y decide las que necesitan que se realice un seguimiento de su variación temporal.

El paso siguiente es añadir información sobre los periodos de validez a cada una de esas relaciones, añadiendo como atributos el momento de inicio y el de finalización. Por ejemplo, supóngase que se tiene la relación

asignatura (*id_asignatura*, *denominación_asignatura*)

que asocia la denominación de cada asignatura con la asignatura correspondiente, que queda identificada mediante su identificador de asignatura. La denominación de la asignatura puede variar con el tiempo, lo cual se puede tener en cuenta añadiendo un rango de validez; el esquema resultante sería:

asignatura (*id_asignatura*, *denominación_asignatura*, *comienzo, final*)

Un ejemplar de esta relación puede tener dos registros (CS101, "Introducción a la programación", 01/01/1985, 31/12/2000) y (CS101,"Introdución a C", 01/01/2001, 31/12/9999). Si se actualiza la relación para cambiar la denominación de la asignatura a "Introducción a Java", se actualizaría la fecha "31/12/9999" a la correspondiente al momento hasta el que fue válido el valor anterior ("Introducción a C"), y se añadiría una tupla nueva que contendría la nueva denominación ("Introducción a Java"), con la fecha de inicio correspondiente.

Si otra relación tuviera una clave externa que hiciera referencia a una relación temporal, el diseñador de la base de datos tendría que decidir si la referencia se hace a la versión actual de los datos o a los datos de un momento concreto. Por ejemplo, una relación que registre la asignación de aulas actual para cada asignatura puede hacer referencia de manera implícita al valor temporal actual asociado con cada *id_asignatura*. Por otro lado, los registros del expediente de cada estudiante deben hacer referencia a la denominación de la asignatura en el momento en que la cursó ese estudiante. En este último caso, la relación que hace la referencia también debe almacenar la información temporal, para identificar cada registro concreto de la relación *asignatura*.

La clave primaria original de una relación temporal ya no identificaría de manera unívoca a cada tupla. Para solucionar este problema se pueden añadir a la clave primaria los atributos de fecha de inicio y de finalización. Sin embargo, persisten algunos problemas:

- Es posible almacenar datos con intervalos que se solapen, pero la restricción de clave primaria no puede detectarlo. Si el sistema soporta un tipo nativo *periodo de validez*, puede detectar y evitar esos intervalos temporales que se solapan.
- Para especificar una clave externa que haga referencia a una relación así, las tuplas que hacen la referencia tienen que incluir los atributos de momento inicial y final como parte de su clave externa, y los valores deberán coincidir con los de la tupla a la que hacen referencia. Además, si la tupla a la que hacen referencia se actualiza (y el momento final que se hallaba en el futuro se actualiza), esa actualización debe propagarse a todas las tuplas que hacen referencia a ella.

  Si el sistema soporta los datos temporales de alguna manera mejor, se puede permitir que la tupla que hace la referencia especifique un momento, en lugar de un rango temporal, y confiar en que el sistema garantice que hay una tupla en la relación a la que hace referencia cuyo periodo de validez contenga ese momento. Por ejemplo, un registro de un expediente puede especificar *id_asignatura* y un momento (por ejemplo, la fecha de comienzo de un trimestre), lo que basta para identificar el registro correcto de la relación *asignatura*.

Como caso especial frecuente, si todas las referencias a los datos temporales se hacen exclusivamente a los datos actuales, una solución más sencilla es no añadir información temporal a la relación y, en su lugar, crear la relación *historial* correspondiente que contenga esa información temporal, para los valores del pasado. Por ejemplo, en la base de datos bancaria, se puede utilizar el diseño que se ha creado, ignorando las modificaciones temporales, para almacenar únicamente la información actual. Toda la información histórica se traslada a las relaciones históricas. Por tanto, la relación *cliente* sólo puede almacenar la dirección actual, mientras que la relación *historial_cliente* puede contener todos los atributos de *cliente*, con los atributos adicionales *momento_inicial* y *momento_final*.

Aunque no se ha ofrecido ningún método formal para tratar los datos temporales, los problemas estudiados y los ejemplos ofrecidos deben ayudar al lector a diseñar bases de datos que registren datos temporales. Más adelante, en el Apartado 24.2, se tratan otros aspectos del manejo de datos temporales, incluidas las consultas temporales.

7.10 Resumen

- Se han mostrado algunas dificultades del diseño de bases de datos y el modo de diseñar de manera sistemática esquemas de bases de datos que eviten esas dificultades. Entre esas dificultades están la información repetida y la imposibilidad de representar cierta información.
- Se ha mostrado el desarrollo del diseño de bases de datos relacionales a partir de los diseños E-R, cuándo los esquemas se pueden combinar con seguridad y cuándo se deben descomponer. Todas las descomposiciones válidas deben ser sin pérdidas.
- Se han descrito las suposiciones de dominios atómicos y de primera forma normal.
- Se ha introducido el concepto de las dependencias funcionales y se ha utilizado para presentar dos formas normales, la forma normal de Boyce–Codd (FNBC) y la tercera forma normal (3NF).
- Si la descomposición conserva las dependencias, dada una actualización de la base de datos, todas las dependencias funcionales pueden verificarse a partir de las diferentes relaciones, sin necesidad de calcular la reunión de las relaciones de la descomposición.
- Se ha mostrado la manera de razonar con las dependencias funcionales. Se ha puesto un énfasis especial en señalar las dependencias que están implicadas lógicamente por conjuntos de dependencias. También se ha definido el concepto de recubrimiento canónico, que es un conjunto mínimo de dependencias funcionales equivalente a un conjunto dado de dependencias funcionales.
- Se ha descrito un algoritmo para la descomposición de las relaciones en la FNBC. Hay relaciones para las cuales no hay ninguna descomposición en la FNBC que conserve las dependencias.
- Se han utilizado los recubrimientos canónicos para descomponer las relaciones en la 3NF, que es una pequeña relajación de las condiciones de la FNBC. Las relaciones en la 3NF pueden tener alguna redundancia, pero siempre hay una descomposición en la 3NF que conserva las dependencias.
- Se ha presentado el concepto de dependencias multivaloradas, que especifican las restricciones que no pueden especificarse únicamente con las dependencias funcionales. Se ha definido la cuarta forma normal (4FN) con las dependencias multivaloradas. El Apartado C.1.1 del apéndice da detalles del razonamiento sobre las dependencias multivaloradas.
- Otras formas normales, como la FNRP y la FNDC, eliminan formas más sutiles de redundancia. Sin embargo, es difícil trabajar con ellas y se emplean rara vez. El Apéndice C ofrece detalles de estas formas normales.
- Al revisar los temas de este capítulo hay que tener en cuenta que el motivo de que se hayan podido definir enfoques rigurosos del diseño de bases de datos relacionales es que el modelo relacional de datos descansa sobre una base matemática sólida. Ésa es una de las principales ventajas del modelo relacional en comparación con los otros modelos de datos que se han estudiado.

Términos de repaso

- Modelo E-R y normalización.
- Descomposición.
- Dependencias funcionales.
- Descomposición sin pérdidas.
- Dominios atómicos.
- Primera forma normal (1FN).
- Relaciones legales.
- Superclave.
- R satisface F.
- F se cumple en R.
- Forma normal de Boyce–Codd (FNBC).
- Conservación de las dependencias.
- Tercera forma normal (3NF).
- Dependencias funcionales triviales.
- Cierre de un conjunto de dependencias funcionales.
- Axiomas de Armstrong.
- Cierre de los conjuntos de atributos.
- Restricción de F a R_i.
- Recubrimiento canónico.
- Atributos raros.
- Algoritmo de descomposición en la FNBC.
- Algoritmo de descomposición en la 3NF.
- Dependencias multivaloradas.
- Cuarta forma normal (4FN).
- Restricción de las dependencias multivaloradas.
- Forma normal de reunión por proyección (FNRP).
- Forma normal de dominios y claves (FNDC).
- Relación universal.
- Suposición de un rol único.
- Desnormalización.

Ejercicios prácticos

7.1 Supóngase que se descompone el esquema $R = (A, B, C, D, E)$ en

$$(A, B, C)$$
$$(A, D, E)$$

Demuéstrese que esta descomposición es una descomposición sin pérdidas si se cumple el siguiente conjunto F de dependencias funcionales:

$$A \rightarrow BC$$
$$CD \rightarrow E$$
$$B \rightarrow D$$
$$E \rightarrow A$$

7.2 Indíquense todas las dependencias funcionales que satisface la relación de la Figura 7.18.

7.3 Explíquese el modo en que se pueden utilizar las dependencias funcionales para indicar:

- Existe un conjunto de relaciones de uno a uno entre los conjuntos de entidades *cuenta* y *cliente*.
- Existe un conjunto de relaciones de varios a uno entre los conjuntos de entidades *cuenta* y *cliente*.

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_3

Figura 7.18 La relación del Ejercicio práctico 7.2.

7.4 Empléense los axiomas de Armstrong para probar la corrección de la regla de la unión. *Sugerencia*: utilícese la regla de la aumentatividad para probar que, si $\alpha \rightarrow \beta$, entonces $\alpha \rightarrow \alpha\beta$. Aplíquese nuevamente la regla de la aumentatividad, utilizando $\alpha \rightarrow \gamma$, y aplíquese luego la regla de la transitividad.

7.5 Empléense los axiomas de Armstrong para probar la corrección de la regla de la pseudotransitividad.

7.6 Calcúlese el cierre del siguiente conjunto F de relaciones funcionales para el esquema de relación $R = (A, B, C, D, E)$.

$$A \rightarrow BC$$
$$CD \rightarrow E$$
$$B \rightarrow D$$
$$E \rightarrow A$$

Indíquense las claves candidatas de R.

7.7 Utilizando las dependencias funcionales del Ejercicio práctico 7.6, calcúlese el recubrimiento canónico F_c.

7.8 Considérese el algoritmo de la Figura 7.19 para calcular α^+. Demuéstrese que este algoritmo resulta más eficiente que el presentado en la Figura 7.9 (Apartado 7.4.2) y que calcula α^+ de manera correcta.

7.9 Dado el esquema de base de datos $R(a, b, c)$ y una relación *r* del esquema R, escríbase una consulta SQL para comprobar si la dependencia funcional $b \rightarrow c$ se cumple en la relación *r*. Escríbase también un aserto de SQL que haga que se cumpla la dependencia funcional. Supóngase que no hay ningún valor nulo.

7.10 Sea $R_1, R_2, \ldots, R_n$ una descomposición del esquema U. Sea $u(U)$ una relación y sea $r_i = \Pi_{R_I}(u)$. Demuéstrese que

$$u \subseteq r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

7.11 Demuéstrese que la descomposición del Ejercicio práctico 7.1 no es una descomposición que conserve las dependencias.

7.12 Demuéstrese que es posible asegurar que una descomposición que conserve las dependencias en la 3FN sea una descomposición sin pérdidas garantizando que, como mínimo, un esquema contenga una clave candidata para el esquema que se está descomponiendo. *Sugerencia*: demuéstrese que la reunión de todas las proyecciones en los esquemas de la descomposición no puede tener más tuplas que la relación original.

7.13 Dese un ejemplo de esquema de relación R' y de un conjunto F' de dependencias funcionales tales que haya, al menos, tres descomposiciones sin pérdidas distintas de R' en FNBC.

7.14 Sea atributo *primo* el que aparece, como mínimo, en una clave candidata. Sean α y β conjuntos de atributos tales que se cumple $\alpha \rightarrow \beta$, pero no se cumple $\beta \rightarrow \alpha$. Sea A un atributo que no esté en α ni en β y para el que se cumpla que $\beta \rightarrow \alpha$. Se dice que A es *dependiente de manera transitiva* de α. Se puede reformular la definición de la 3FN de la manera siguiente: el esquema de relación R está en la 3FN con respecto al conjunto F de dependencias funcionales si no hay atributos no primos A en R para los cuales A sea dependiente de manera transitiva de una clave de R.

Demuéstrese que esta nueva definición es equivalente a la original.

7.15 La dependencia funcional $\alpha \rightarrow \beta$ se denomina **dependencia parcial** si hay un subconjunto propio γ de α tal que $\gamma \rightarrow \beta$. Se dice que β es *parcialmente dependiente* de α. El esquema de relación R está en la **segunda forma normal** (2FN) si cada atributo A de R cumple uno de los criterios siguientes:

- Aparece en una clave candidata.

```
resultado := ∅;
/* cuentadf es un array cuyo elemento i-ésimo contiene
   el número de atributos del lado izquierdo de la i-ésima
   DF que todavía no se sabe que estén en α+ */
for i := 1 to |F| do
   begin
      Supóngase que β → γ denota la iésima DF;
      cuentadf [i] := |β|;
   end
/* aparece es un array con una entrada por cada atributo. La
entrada del atributo A es una lista de enteros. Cada entero
i de la lista indica que A aparece en el lado izquierdo de
la i-iésima DF */
for each atributo A do
   begin
      aparece [A] := NIL;
      for i := 1 to |F| do
         begin
            Supóngase que β → γ denota la i-ésima DF;
            if A ∈ β then añadir i a aparece [A];
         end
   end
agregar (α);
return (resultado);

procedure agregar (α);
for each atributo A in α do
   begin
      if A ∉ resultado then
         begin
            resultado := resultado ∪ {A};
            for each elemento i in aparece[A] do
               begin
                  cuentafd [i] := cuentafd [i] − 1;
                  if cuentafd [i] := 0 then
                     begin
                        supóngase que β → γ denota la i-ésima FD;
                        agregar (γ);
                     end
               end
         end
   end
```

Figura 7.19 Algoritmo para calcular α^+.

- No es parcialmente dependiente de una clave candidata.

Demuéstrese que cada esquema en la 3FN se halla en la 2FN. *Sugerencia*: demuéstrese que todas las dependencias parciales son dependencias transitivas.

7.16 Dese un ejemplo de esquema de relación R y un conjunto de dependencias tales que R se halle en la FNBC, pero no en la 4FN.

Ejercicios

7.17 Explíquese lo que se quiere decir con *repetición de la información* e *imposibilidad de representación de la información*. Explíquese el motivo por el que estas propiedades pueden indicar un mal diseño de las bases de datos relacionales.

7.18 Indíquese el motivo de que ciertas dependencias funcionales se denominen dependencias funcionales *triviales*.

7.19 Utilícese la definición de dependencia funcional para argumentar que cada uno de los axiomas de Armstrong (reflexividad, aumentatividad y transitividad) es correcto.

7.20 Considérese la siguiente regla propuesta para las dependencias funcionales: si $\alpha \rightarrow \beta$ y $\gamma \rightarrow \beta$, entonces $\alpha \rightarrow \gamma$. Pruébese que esta regla *no* es correcta mostrando una relación *r* que satisfaga $\alpha \rightarrow \beta$ y $\gamma \rightarrow \beta$, pero no $\alpha \rightarrow \gamma$.

7.21 Utilícense los axiomas de Armstrong para probar la corrección de la regla de la descomposición.

7.22 Utilizando las dependencias funcionales del Ejercicio práctico 7.6, calcúlese B^+.

7.23 Demuéstrese que la siguiente descomposición del esquema R del Ejercicio práctico 7.1 no es una descomposición sin pérdidas:

$$(A, B, C)$$
$$(C, D, E)$$

Sugerencia: dese un ejemplo de una relación *r* del esquema R tal que

$$\Pi_{A,B,C}(r) \bowtie \Pi_{C,D,E}(r) \neq r$$

7.24 Indíquense los tres objetivos de diseño de las bases de datos relacionales y explíquese el motivo de que cada uno de ellos sea deseable.

7.25 Dese una descomposición sin pérdidas en la FNBC del esquema R del Ejercicio práctico 7.1.

7.26 Al diseñar una base de datos relacional, indíquese el motivo de que se pueda escoger un diseño que no esté en la FNBC.

7.27 Dese una descomposición sin pérdidas en la 3FN que conserve las dependencias del esquema R del Ejercicio práctico 7.1.

7.28 Dados los tres objetivos del diseño de bases de datos relacionales, indíquese si hay alguna razón para diseñar un esquema de base de datos que se halle en la 2FN, pero que no se halle en ninguna forma normal de orden superior (véase el Ejercicio práctico 7.15 para obtener la definición de la 2FN).

7.29 Dado el esquema relacional $r(A, B, C, D)$, ¿implica lógicamente $A \twoheadrightarrow BC$ a $A \twoheadrightarrow B$ y a $A \twoheadrightarrow C$? En caso positivo, pruébese; en caso contrario, dese un contraejemplo.

7.30 Explíquese el motivo de que la 4FN sea una forma normal más deseable que la FNBC.

Notas bibliográficas

El primer estudio de la teoría del diseño de bases de datos relacionales apareció en un artículo pionero de Codd [1970]. En ese artículo, Codd introducía también las dependencias funcionales y la primera, la segunda y la tercera formas normales.

Los axiomas de Armstrong se introdujeron en Armstrong [1974]. A finales de los años setenta se produjo un desarrollo significativo de la teoría de las bases de datos relacionales. Esos resultados se recogen en varios textos sobre la teoría de las bases de datos, como Maier [1983], Atzeni y Antonellis [1993] y Abiteboul et al. [1995].

La FNBC se introdujo en Codd [1972]. Biskup et al. [1979] dan el algoritmo que se ha utilizado para encontrar descomposiciones en la 3FN que conserven las dependencias. Los resultados fundamentales de la propiedad de la descomposición sin pérdidas aparecen en Aho et al. [1979a].

Beeri et al. [1977] dan un conjunto de axiomas para las dependencias multivaluadas y prueba que los axiomas de los autores son correctos y completos. Los conceptos de 4FN, de FNRP y de FNDC son de Fagin [1977], Fagin [1979] y de Fagin [1981], respectivamente. Véanse las notas bibliográficas del Apéndice C para tener más referencias sobre la literatura relativa a la normalización.

Jensen et al. [1994] presenta un glosario de conceptos relacionados con las bases de datos temporales. Gregersen y Jensen [1999] presenta un resumen de las extensiones del modelo E-R para el tratamiento de datos temporales. Tansel et al. [1993] trata la teoría, el diseño y la implementación de las bases de datos temporales. Jensen et al. [1996] describe las extensiones de la teoría de la dependencia a los datos temporales.

CAPÍTULO 8

Diseño y desarrollo de aplicaciones

Casi todo el uso de las bases de datos se produce desde los programas de aplicación. A su vez, casi toda la interacción de los usuarios con las bases de datos es indirecta, mediante los programas de aplicación. No resulta sorprendente, por tanto, que los sistemas de bases de datos lleven mucho tiempo soportando herramientas como los generadores de formularios y de interfaces gráficas de usuario, que ayudan a lograr el desarrollo rápido de aplicaciones que actúan de interfaz con los usuarios. En los últimos años, la red Web se ha transformado en la interfaz de usuario con las bases de datos más usada.

En la primera parte de este capítulo (Apartados 8.1 a 8.4) se estudian las herramientas y las tecnologías necesarias para crear aplicaciones de bases de datos. En concreto, se centrará la atención en las herramientas que ayudan al desarrollo de interfaces de usuario para las bases de datos. Se comenzará con una introducción a las herramientas para la creación de interfaces de formularios e informes. Posteriormente se ofrecerá una visión detallada del modo en que se desarrollan aplicaciones con interfaces basadas en Web.

Más adelante se tratarán los disparadores. Los disparadores permiten que las aplicaciones controlen los eventos (actividades) de las bases de datos y emprendan acciones cuando se produzcan determinados eventos. También ofrecen una manera de añadir reglas y acciones sin modificar el código ya existente de las aplicaciones. Los disparadores fueron un añadido de última hora a la norma de SQL. Se presentará la sintaxis de SQL tanto en la modalidad correspondiente a la norma SQL:1999 como en la de algunos sistemas comerciales.

Finalmente, se tratarán la autorización y la seguridad. Se describirán los mecanismos de autorización ofrecidos por SQL y la sintaxis para su uso. Posteriormente se estudiarán las limitaciones de los mecanismos de autorización de SQL y se presentarán otros conceptos y tecnologías necesarios para la protección de las bases de datos y de las aplicaciones.

8.1 Interfaces de usuario y herramientas

Aunque mucha gente interactúa con las bases de datos, poca usa un lenguaje de consultas para interactuar directamente con los sistemas de bases de datos. La mayor parte de las personas interactúan con los sistemas de bases de datos usando alguno de los medios siguientes:

1. Los **formularios** y las **interfaces gráficas de usuario** permiten a los usuarios introducir los valores que completan las consultas predefinidas. El sistema ejecuta las consultas, aplica el formato correspondiente y muestra el resultado al usuario. Las interfaces gráficas de usuario ofrecen una manera sencilla de usar de interactuar con los sistemas de bases de datos.
2. Los **generadores de informes** permiten que se generen informes predefinidos sobre el contenido actual de la base de datos. Los analistas o los administradores ven esos informes para tomar decisiones comerciales.

3. Las **herramientas de análisis de datos** permiten que los usuarios examinen y analicen los datos de manera interactiva.

Merece la pena destacar que estas interfaces usan lenguajes de consultas para comunicarse con los sistemas de bases de datos.

En este apartado se ofrece una visión general de los formularios, de las interfaces gráficas de usuario y de los generadores de informes. El Capítulo 18 trata las herramientas de análisis de datos con más detalle. Por desgracia, no hay ninguna norma para las interfaces de usuario y cada sistema de bases de datos suele ofrecer su propia interfaz de usuario. En este apartado se describen los conceptos básicos, sin profundizar en los detalles de ningún producto de interfaz de usuario concreto.

8.1.1 Formularios e interfaces gráficas de usuario

Las interfaces de formularios se usan mucho para introducir datos y extraer información de las bases de datos, mediante consultas predefinidas. Por ejemplo, los motores de búsqueda en la World Wide Web ofrecen formularios que se usan para introducir las palabras clave. La pulsación del botón de "remisión" hace que el motor de búsqueda ejecute la consulta usando las palabras clave introducidas y muestre el resultado al usuario.

Como ejemplo más orientado a las bases de datos, se puede establecer conexión con el sistema de matriculación de una universidad, en el que se pide que se escriban en un formulario el número de identificación y la contraseña. El sistema usa esta información para comprobar la identidad del usuario, y para extraer información, como el nombre y las asignaturas de las que se ha matriculado, de la base de datos y mostrarla. Puede que haya otros enlaces en la página Web que permitan buscar asignaturas y hallar más información sobre ellas, como puede ser el programa y la persona que las imparte.

Los programadores pueden crear formularios e interfaces gráficas de usuario usando navegadores Web como parte visible al usuario, o usando formularios y otros servicios ofrecidos por las interfaces de programación de aplicaciones (APIs, application-programmer interfaces) de los lenguajes de programación, como Java Swing, o las APIs proporcionadas con Visual Basic o con Visual C++. Los navegadores Web que soportan HTML constituyen los formularios e interfaces gráficas de usuario más usadas hoy en día. Aunque el navegador Web proporciona la parte visible al usuario para interactuar con él, el procesamiento subyacente se lleva a cabo en el servidor Web, usando tecnologías tales como los servlets de Java, las páginas de servidor de Java (Java Server Pages, JSP) o las páginas activas de servidor (Active Server Pages, ASP). En el Apartado 8.3.2 se estudiará la manera de crear formularios usando HTML y la manera de crear sistemas en segundo plano mediante los servlets de Java en el Apartado 8.4.

Hay gran variedad de herramientas que simplifican la creación de interfaces gráficas de usuario y de formularios. Estas herramientas permiten a los desarrolladores de aplicaciones crear formularios de manera declarativa sencilla, mediante los programas editores de formularios. Los usuarios pueden definir el tipo, el tamaño y el formato de cada campo del formulario mediante el editor de formularios. Se pueden asociar acciones del sistema con las acciones de los usuarios, como la escritura en un campo, la pulsación de una tecla de función del teclado o el envío de un formulario. Por ejemplo, la ejecución de una consulta para escribir los campos nombre y dirección puede asociarse con la escritura del campo número de identificación, y la ejecución de una instrucción de actualización se puede asociar con el envío de un formulario. Ejemplos de estos sistemas son Oracle Forms, Sybase PowerBuilder y Oracle HTML-DB.

Las comprobaciones de errores sencillas pueden llevarse a cabo definiendo restricciones para los campos del formulario. Por ejemplo, una restricción de un campo de fecha puede comprobar si la fecha introducida por el usuario tiene el formato correcto y se halla en el rango deseado (por ejemplo, un sistema de reservas puede exigir que la fecha no sea anterior a la fecha de hoy ni diste más de seis meses). Aunque estas restricciones se pueden comprobar al ejecutar la transacción, la detección temprana de los errores ayuda a los usuarios a corregirlos con rapidez. Los menús que indican los valores válidos que se pueden introducir en cada campo pueden ayudar a eliminar la posibilidad de muchos tipos de errores. Los desarrolladores de sistemas consideran que la posibilidad de controlar de manera declarativa estas características con la ayuda de una herramienta de desarrollo de interfaces de usuario, en lugar de crear directamente el formulario mediante lenguajes de guiones o de programación, facilita mucho su trabajo.

Compañía de suministros Acme, S.L.
Informe trimestral de ventas

Periodo: 1 de enero a 31 de marzo de 2005

Región	Categoría	Ventas	Subtotal
Norte	Hardware informático	1.000.000	
	Software informático	500.000	
	Todas las categorías		1.500.000
Sur	Hardware informático	200.000	
	Software informático	400.000	
	Todas las categorías		600.000
		Ventas totales	2.100.000

Figura 8.1 Informe con formato.

8.1.2 Generadores de informes

Los generadores de informes son herramientas para generar informes legibles a partir de las bases de datos. Integran la consulta a la base de datos con la creación de texto con formato y con gráficos resumen (como los gráficos de barras o circulares). Por ejemplo, un informe puede mostrar las ventas totales de los últimos dos meses para cada región de ventas.

El desarrollador de aplicaciones puede especificar el formato de los informes mediante los servicios de formato del generador de informes. Se pueden usar variables para almacenar parámetros como el mes y el año y para definir los campos del informe. Las tablas, los gráficos, los gráficos de barras u otros gráficos se pueden definir mediante consultas a la base de datos. Las definiciones de las consultas pueden hacer uso de los valores de los parámetros almacenados en las variables.

Una vez definida la estructura de los informes en un servicio generador de informes se puede almacenar y ejecutar en cualquier momento para generar informes. Los sistemas generadores de informes ofrecen gran variedad de servicios para estructurar el resultado tabular, como la definición de encabezados de tablas y de columnas, la visualización de los subtotales correspondientes a cada grupo de la tabla, la división automática de las tablas de gran tamaño en varias páginas y la visualización de los subtotales al final de cada página.

La Figura 8.1 es un ejemplo de informe con formato. Los datos del informe se generan mediante la agregación de la información sobre los pedidos.

Gran variedad de marcas, como Crystal Reports y Microsoft (SQL Server Reporting Services), ofrecen herramientas para la generación de informes. Varias familias de aplicaciones, como Microsoft Office, incluyen procedimientos para incrustar directamente en los documentos los resultados con formato de las consultas a la base de datos. Los servicios de generación de gráficos proporcionados por Crystal Reports, o por hojas de cálculo como Excel se pueden usar para tener acceso a las bases de datos y generar descripciones tabulares o gráficas de los datos. En los documentos de texto creados con Microsoft Word, por ejemplo, se puede incrustar uno o más de estos gráficos. La característica de Microsoft Office denominada OLE (object linking and embedding, vinculación e incrustación de objetos) se utiliza para vincular los gráficos en el documento de texto. Los gráficos se crean inicialmente a partir de datos generados ejecutando consultas a la base de datos; las consultas se pueden volver a ejecutar y los gráficos se pueden regenerar cuando sea necesario, para crear una versión actual del informe global.

Además de generar los informes estáticos, estas herramientas permiten que sean interactivos. Por ejemplo, un usuario puede "profundizar" en áreas de su interés, por ejemplo, pasar de una vista agregada que muestra las ventas totales de un año completo a las cifras de ventas mensuales un año concreto. El análisis interactivo de datos se volverá a tratar más adelante, en el Apartado 18.2.

8.2 Interfaces Web para bases de datos

World Wide Web (**Web**, para abreviar) es un sistema distribuido de información basado en el hipertexto. Las interfaces Web con las bases de datos se han vuelto muy importantes. Tras describir varios motivos para establecer interfaces con Web, se ofrece una visión general de la tecnología Web (Apartado 8.3). Posteriormente se describen algunas técnicas para la creación de interfaces Web para las bases de datos, mediante servlets y lenguajes de guiones del lado del servidor (Apartado 8.4). Este tema se completa en el Apartado 8.5 mediante la descripción de técnicas para la creación de aplicaciones Web de gran escala y la mejora de su rendimiento.

Web es importante como parte visible al usuario de las bases de datos por varios motivos: los navegadores Web ofrecen una *interfaz universal* para la información facilitada por los sistemas subyacentes ubicados en cualquier parte del mundo. La parte visible al usuario puede ejecutarse en cualquier sistema informático y no hace falta que el usuario descargue software específico para que tenga acceso a la información. Además, hoy en día casi todas las personas que pueden permitírselo tienen acceso a Web.

Con el crecimiento de los servicios de información y del comercio electrónico en Web, las bases de datos usadas para los servicios de información, la ayuda a la toma de decisiones y el procesamiento de transacciones deben estar conectados a Web. La interfaz de los formularios HTML resulta conveniente para el procesamiento de las transacciones. El usuario puede rellenar los datos de un formulario de pedido y pulsar un botón de envío para remitir un mensaje al servidor. El servidor ejecuta el programa de aplicación correspondiente al formulario de pedido y esta acción, a su vez, inicia transacciones en la base de datos. El servidor da formato al resultado de la transacción y se lo devuelve al usuario.

Otro motivo para el uso de interfaces entre las bases de datos y Web es que la presentación exclusiva de documentos estáticos (fijos) en un sitio Web presenta algunas limitaciones, aunque el usuario no realice ninguna consulta ni procese ninguna transacción:

- Los documentos Web estáticos no permiten su adaptación al usuario. Por ejemplo, puede que un periódico desee personalizar su aspecto para cada usuario, para destacar los artículos informativos que sea más probable que interesen a cada persona.
- Cuando se actualizan los datos de una organización, los documentos de Web se vuelven obsoletos si no se actualizan también. El problema es más grave si varios documentos de Web replican datos importantes y hay que actualizarlos todos.

Estos problemas pueden resolverse mediante la generación dinámica de los documentos de Web a partir de una base de datos. Cuando se solicita un documento, se ejecuta un programa en el sitio del servidor, que a su vez ejecuta consultas a la base de datos y genera el documento solicitado de acuerdo con el resultado de las consultas. Siempre que se actualizan datos importantes de la base de datos se actualizan los documentos generados de manera automática. El documento generado también puede personalizarse para el usuario de acuerdo con la información del usuario guardada en la base de datos.

Las interfaces Web ofrecen ventajas atrayentes incluso para las aplicaciones de bases de datos que sólo se usan dentro de una organización. La norma **lenguaje de marcas de hipertexto** (HyperText Markup Language, HTML) permite que el texto reciba formato de manera ordenada y que se destaque la información importante. Los **hipervínculos**, que son enlaces con otros documentos, pueden asociarse con regiones de los datos mostrados. Al pulsar en un hipervínculo se accede y muestra el documento vinculado. Los hipervínculos resultan muy útiles para explorar datos, ya que permiten que los usuarios obtengan más detalles de los datos que prefieran.

Finalmente, los navegadores actuales acceder a programas desde documentos HTML, y ejecutarlos en el navegador en modo seguro—es decir, sin dañar los datos de la computadora del usuario. Los programas pueden escribirse en lenguajes de guiones del lado del cliente, como Javascript, o ser "applets" escritas en el lenguaje Java, o animaciones escritas en lenguajes como Flash o Shockwave. Estos programas permiten la creación de interfaces de usuario sofisticadas, más de lo que es posible con HTML únicamente, interfaces que pueden usarse sin descargar ni instalar ningún software. Por ello, las interfaces Web resultan potentes y visualmente atractivas, y han eclipsado las interfaces específicas de gran variedad de aplicaciones de bases de datos.

```
<html>
<body>
<table BORDER COLS=3>
<tr> <td>C-101</td> <td>Centro</td> <td>500</td> </tr>
<tr> <td>C-102</td> <td>Navacerrada</td> <td>400</td> </tr>
<tr> <td>C-201</td> <td>Galapagar  </td> <td>900</td> </tr>
</table>
<center> La relación <i>cuenta</i></center>

<form action="ConsultaBanco" method=get>
Seleccione cuenta o préstamo e introduzca el número <br>
<select name="tipo">
    <option value="cuenta" selected>Cuenta </option>
    <option value="préstamo">Préstamo </option>
</select>
<input type=text size=5 name="número">
<input type=submit value="Enviar">
</form>
</body>
</html>
```

Figura 8.2 Texto fuente HTML.

8.3 Fundamentos de Web

En este apartado se va a repasar parte de la tecnología básica que subyace en World Wide Web, para los lectores que no estén familiarizados con ella.

8.3.1 Los localizadores uniformes de recursos

Un **localizador uniforme de recursos** (**URL, Uniform Resource Locator**) es un nombre globalmente único para cada documento al que se puede tener acceso en Web. Un ejemplo de URL es

http://www.acm.org/sigmod

La primera parte del URL indica el modo en que se puede tener acceso al documento: "http" indica que se puede tener acceso al documento mediante el protocolo de transferencia de hipertexto (HyperText Transfer Protocol, HTTP), el cual es un protocolo para transferir documentos HTML. La segunda parte indica el nombre único de una máquina con el servidor Web. El resto del URL es el nombre del camino hasta el archivo en la máquina, u otro identificador único del documento dentro de la máquina.

Muchos datos de Web se generan de manera dinámica. Un URL puede contener el identificador de un programa ubicado en la máquina servidora Web, así como los argumentos que hay que darle al programa. Un ejemplo de URL de este tipo es

http://www.google.com/search?q=silberschatz

que dice que el programa search del servidor www.google.com se debe ejecutar con el argumento q=silberschatz. El programa se ejecuta, usando el argumento dado, y devuelve un documento HTML que se envía a la interfaz.

8.3.2 El lenguaje de marcas de hipertexto

La Figura 8.2 es un ejemplo de origen de un documento HTML. La Figura 8.3 muestra la imagen que crea este documento.

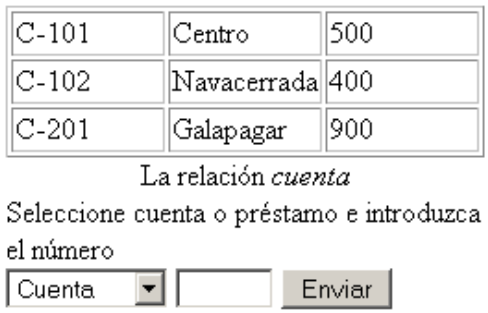


Figura 8.3 Aspecto del código HTML de la Figura 8.2.

Las figuras muestran el modo en que HTML puede mostrar una tabla y un formulario sencillo que permite que los usuarios seleccionen el tipo (cuenta o préstamo) de un menú e introduzcan un número en un cuadro de texto. HTML también soporta otros tipos de entrada. La pulsación del botón de envío hace que el programa ConsultaBanco (especificado en el campo form action) se ejecute con los valores proporcionados por el usuario para los argumentos tipo y número (especificados en los campos select e input). El programa genera un documento HTML, que se devuelve al usuario y se le muestra; se verá el modo de crear estos programas en los Apartados 8.3.4, 8.4 y 8.4.5.

HTTP define dos maneras de enviar al servidor Web los valores introducidos por los usuarios. El método get codifica los valores como parte del URL. Por ejemplo, si la página de búsqueda de Google usa un formulario con un parámetro de entrada denominado q con el método get y el usuario escribe "silberschatz" en la cadena de caracteres y envía el formulario, el navegador solicitará al servidor Web el URL siguiente:

http://www.google.com/search?q=silberschatz

El método post, por el contrario, envía una petición de la página www.google.com y envía el valor de los parámetros como parte del intercambio del protocolo HTTP entre el servidor Web y el navegador. El formulario de la Figura 8.2 especifica que el formulario utilice el método get.

Aunque el código HTML se puede crear usando un editor de texto sencillo, hay varios editores que permiten la creación directa de texto HTML usando una interfaz gráfica. Estos editores permiten insertar construcciones como los formularios, los menús y las tablas en el documento HTML a partir de un menú de opciones, en lugar de escribir manualmente el código para generar esas construcciones.

HTML soporta *hojas de estilo*, que pueden modificar las definiciones predeterminadas del modo en que se muestran las estructuras de formato HTML, así como otros atributos de visualización como el color de fondo de la página. La norma *hojas de estilo en cascada* (*cascading stylesheet*, CSS) permite usar varias veces la misma hoja de estilo para varios documentos HTML, dando un aspecto uniforme aunque distintivo a todas las páginas del sitio Web.

8.3.3 Secuencias de comandos del lado del cliente y applets

La inclusión de código ejecutable en los documentos permite que las páginas Web sean **activas** y ejecuten actividades como la animación mediante la ejecución de programas en el sitio local, en lugar de presentar simplemente textos y gráficos pasivos. El uso principal de estos programas es la interacción flexible con el usuario, más allá de la limitada capacidad de interacción proporcionada por HTML y por los formularios HTML. Además, la ejecución de programas en el sitio del cliente acelera mucho la interacción, en comparación con tener que enviar cada interacción al sitio del servidor para su procesamiento.

Un riesgo que aparece al permitir estos programas es que, si el diseño del sistema no se realiza con cuidado, el código de programa incluido en la página Web (o, de modo equivalente, en un mensaje de correo electrónico) puede realizar acciones malévolas en la computadora del usuario. Las acciones malévolas pueden ser la lectura de información privada, la eliminación y modificación de información de la computadora y hasta la toma del control de la computadora y la difusión del código a otras compu-

tadoras (por correo electrónico, por ejemplo). En los últimos años muchos virus transmitidos por correo electrónico se han difundido ampliamente de este modo.

Una de las razones por las que el lenguaje *Java* se ha hecho tan popular es que proporciona un modo seguro de ejecutar los programas en las computadoras de los usuarios. El código Java puede compilarse en "código de bytes" independiente de la plataforma, que puede ejecutarse en cualquier navegador que soporte Java. A diferencia de los programas locales, los programas de Java (applets) descargados como parte de una página Web carecen de autoridad para llevar a cabo acciones que puedan resultar destructivas. Se les permite mostrar datos en la pantalla o establecer una conexión de red para obtener más información con el servidor desde el que se ha descargado la página Web. Sin embargo, no se les permite el acceso a los archivos locales, ejecutar programas del sistema ni establecer conexiones de red con otras computadoras.

Aunque Java es un lenguaje de programación completo, existen lenguajes más sencillos, denominados **lenguajes de guiones**, que pueden enriquecer la interacción con el usuario, mientras ofrecen la misma protección que Java. Estos lenguajes proporcionan estructuras que pueden incluirse en los documentos HTML. Los **lenguajes de guiones del lado del cliente** son lenguajes diseñados para ejecutarse en el navegador Web del cliente. De entre ellos, el lenguaje *Javascript* es, con mucho, el más usado. Javascript se suele utilizar para diversas tareas. Por ejemplo, las funciones escritas en Javascript se pueden usar para llevar a cabo comprobaciones de errores (validaciones) de los datos introducidos por el usuario, como el que la cadenas de caracteres de la fecha tenga el formato correcto, o el que el valor introducido (como la edad) se halle en el rango adecuado.

Javascript puede usarse incluso para modificar de manera dinámica el código HTML que se muestra. El navegador divide el código HTML en una estructura arbórea dentro de la memoria definida por una norma denominada **modelo de objetos documento** (Document Object Model, DOM). El código Javascript puede modificar la estructura arbórea para llevar a cabo determinadas operaciones. Por ejemplo, supóngase que un usuario necesita introducir varias filas de datos como elementos de una factura que se está creando. Una tabla con cuadros de texto y otros métodos de entrada de datos en un formulario se puede usar para recoger los datos que introduce el usuario. La tabla puede tener un tamaño predeterminado pero, si se necesitan más filas, se pueden añadir pulsando un botón con la etiqueta "Añadir elemento", por ejemplo. Este botón se puede configurar para que invoque una función de Javascript que modifique el árbol DOM añadiendo a la tabla una fila adicional.

También existen lenguajes de guiones con finalidades específicas para determinadas tareas, como la animación (Flash y Shockwave de Macromedia, por ejemplo) y el modelado tridimensional (lenguaje de marcas de realidad virtual—Virtual Reality Markup Language, VRML). Los lenguajes de guiones también pueden usarse en el lado del servidor, como se verá más adelante.

8.3.4 Los servidores Web y las sesiones

Los **servidores Web** son programas que se ejecutan en la máquina servidora, que aceptan las solicitudes de los navegadores Web y devuelven los resultados en forma de documentos HTML. El navegador y el servidor Web se comunican mediante un protocolo denominado **protocolo de transferencia de hipertexto** (HyperText Transfer Protocol, HTTP). HTTP proporciona características potentes, aparte de la mera transferencia de documentos. La característica más importante es la posibilidad de ejecutar programas, con los argumentos proporcionados por el usuario, y devolver los resultados como documentos HTML.

En consecuencia, los servidores Web pueden actuar con facilidad como intermediarios para ofrecer acceso a gran variedad de servicios de información. Se pueden crear servicios nuevos mediante la creación e instalación de programas de aplicaciones que los ofrezcan. La norma **interfaz de pasarela común** (Common Gateway Interface, CGI) define el modo en que el servidor Web se comunica con los programas de las aplicaciones. Los programas de las aplicaciones suelen comunicarse con servidores de bases de datos, mediante ODBC, JDBC u otros protocolos, con objeto de obtener o guardar datos.

La Figura 8.4 muestra un servicio Web que usa una arquitectura de tres capas, con un servidor Web, un servidor de aplicaciones y un servidor de bases de datos. El uso de varios niveles de servidores aumenta la sobrecarga del sistema; la interfaz CGI inicia un nuevo proceso para atender cada solicitud, lo cual supone una sobrecarga aún mayor.

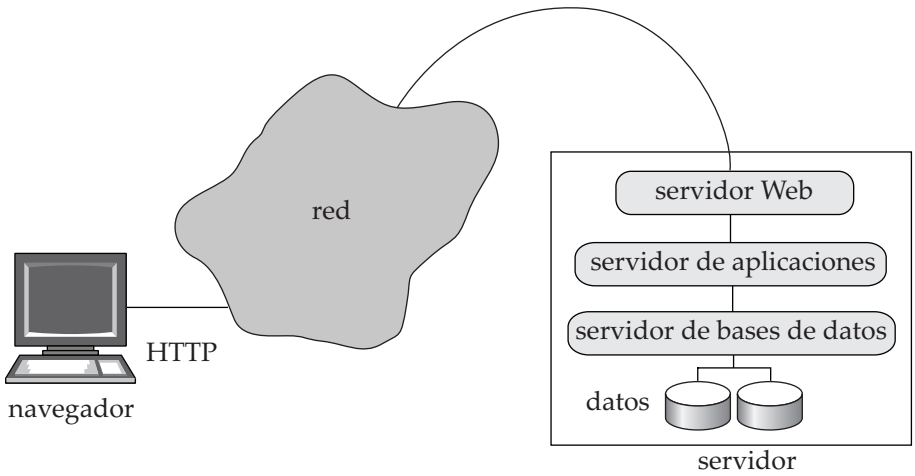


Figura 8.4 Arquitectura Web de tres capas.

La mayor parte de los servicios Web de hoy en día usan una arquitectura Web de dos capas, en la que los programas de las aplicaciones se ejecutan en el servidor Web, como en la Figura 8.5. En los apartados siguientes se estudiarán con más detalle los sistemas basados en la arquitectura de dos capas.

No existe ninguna conexión continua entre los clientes y los servidores Web; cuando un servidor Web recibe una solicitud, se crea temporalmente una conexión para enviar la solicitud y recibir la respuesta del servidor Web. Pero se cierra la conexión, y la siguiente solicitud llega mediante otra nueva. A diferencia de esto, cuando un usuario inicia una sesión en una computadora, o se conecta a una base de datos mediante ODBC o JDBC, se crea una sesión y en el servidor y en el cliente se conserva la información de la sesión hasta que ésta concluye—información como el identificador de usuario y sus opciones de sesión que haya definido. Una razón de peso para que HTTP sea **sin conexión** es que la mayor parte de las computadoras presentan un número limitado de conexiones simultáneas que pueden aceptar, por lo que se podría superar ese límite y denegarse el servicio a otros usuarios. Con un servicio sin conexión, ésta se interrumpe en cuanto se satisface una solicitud, lo que deja las conexiones disponibles para otras solicitudes.

La mayor parte de los servicios de información basados en Web necesitan información sobre las sesiones para permitir una interacción significativa con el usuario. Por ejemplo, los servicios suelen restringir el acceso a la información y, por tanto, necesitan autentificar a los usuarios. La autenticación debe hacerse una vez por sesión, y las interacciones posteriores de la sesión no deben exigir que se repita la autenticación.

Para implementar sesiones a pesar del cierre de las conexiones hay que almacenar información adicional en el cliente y devolverla con cada solicitud de la sesión; el servidor usa esta información para identificar que la solicitud forma parte de la sesión del usuario. En el servidor también hay que guardar información adicional sobre la sesión.

Esta información adicional se suele conservar en el cliente en forma de **cookie**; una cookie no es más que un pequeño fragmento de texto, con un nombre asociado, que contiene información de identificación. Por ejemplo, google.com puede definir una cookie con el nombre **prefs** que codifique las preferencias definidas por el usuario, como el idioma preferido y el número de respuestas mostradas por página. En cada solicitud de búsqueda **google.com** puede recuperar la cookie denominada prefs del navegador del usuario y mostrar el resultado de acuerdo con las preferencias especificadas. Sólo se le permite a cada dominio (sitio Web) que recupere las cookies que ha definido, no las definidas por otros dominios, por lo que sus nombres se pueden reutilizar en otros dominios.

Con objeto de realizar un seguimiento de las sesiones de usuario, una aplicación puede generar un identificador de sesión (generalmente un número aleatorio que no se esté usando como identificador de sesión) y enviar una cookie denominada (por ejemplo) idsesión que contenga ese identificador de sesión. El identificador de sesión también se almacena localmente en el servidor. Cuando llega una solicitud, el servidor de aplicaciones solicita al cliente la cookie denominada idsesión. Si el cliente no posee la cookie

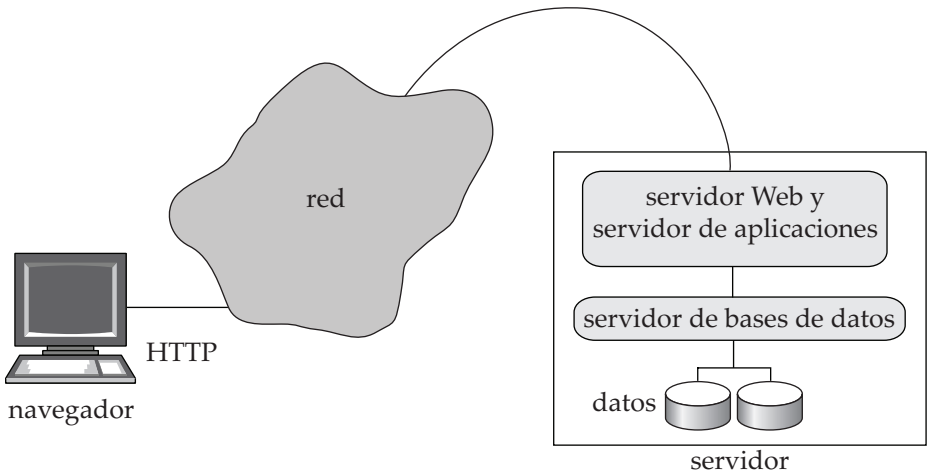


Figura 8.5 Arquitectura Web de dos capas.

almacenada, o devuelve un valor que no se halla registrado como identificador de sesión válido en el servidor, la aplicación concluye que la solicitud no forma parte de una sesión actual. Si el valor de la cookie coincide con el de un identificador de sesión almacenado, la solicitud se identifica como parte de una sesión abierta.

Si una aplicación necesita identificar de manera segura a los usuarios, puede definir la cookie sólo después de autentificar al usuario; por ejemplo, se puede autentificar al usuario sólo cuando se hayan enviado un nombre de usuario y una contraseña válidos[1].

Para las aplicaciones que no exigen un nivel elevado de seguridad, como los sitios de noticias abiertos al público, las cookies se pueden almacenar de manera permanente en el navegador y en el servidor; identifican al usuario en visitas posteriores al mismo sitio, sin necesidad de que escriba ninguna información de identificación. Para las aplicaciones que exijan un nivel de seguridad más elevado, el servidor puede invalidar (eliminar) la sesión tras un periodo de inactividad o cuando el usuario la cierre (generalmente los usuarios cierran la sesión pulsando un botón de cierre de sesión, que remite un formulario de cierre de sesión, cuya acción es invalidar la sesión actual). La invalidación de la sesión consiste simplemente en eliminar el identificador de la sesión de la lista de sesiones activas del servidor de aplicaciones.

8.4 Servlets y JSP

En la arquitectura Web de dos capas las aplicaciones se ejecutan como parte del propio servidor Web. Un modo de implementar esta arquitectura es cargar los programas Java en el servidor Web. La especificación **servlet** de Java define una interfaz de programación de aplicaciones para la comunicación entre el servidor Web y los programas de aplicaciones. La clase HttpServlet de Java implementa la especificación servlet de la API; las clases **servlet** usadas para implementar funciones concretas se definen como subclases de esta clase[2]. A menudo se usa la palabra *servlet* para hacer referencia a un programa (y a una clase) de Java que implementa la interfaz servlet. La Figura 8.6 muestra un ejemplo de código servlet; en breve se explicará con detalle.

1. El identificador de usuario puede guardarse en la parte del cliente, en una cookie denominada, por ejemplo, idusuario. Estas cookies se pueden usar para aplicaciones con un nivel de seguridad bajo, como los sitios Web gratuitos que identifican a sus usuarios. No obstante, para las aplicaciones que exigen un nivel de seguridad más elevado, este mecanismo genera un riesgo: los usuarios malintencionados pueden modificar el valor de las cookies en el navegador y luego pueden suplantar a otros usuarios. La definición de una cookie (denominada idsesión, por ejemplo) con un identificador de sesión generado aleatoriamente (a partir de un espacio de números de gran tamaño) hace muy improbable que ningún usuario pueda suplantar como un usuario diferente (es decir, que pretenda ser otro usuario). Los identificadores de sesión generados de manera secuencial, por otro lado, sí son susceptibles de suplantación.
2. La interfaz de los servlets también puede soportar solicitudes que no sean HTTP, aunque el ejemplo elegido sólo usa HTTP.

El código del servlet (es decir, el programa de Java que implementa la interfaz del servlet) se carga en el servidor Web cuando se inicia el servidor, o cuando el servidor recibe una solicitud HTTP remota para ejecutar un servlet concreto. La tarea del servlet es procesar esa solicitud, lo cual puede suponer acceder a una base de datos para recuperar la información necesaria, y generar dinámicamente una página HTML para devolvérsela al navegador del cliente.

8.4.1 Un ejemplo de servlet

Los servlets se emplean a menudo para generar de manera dinámica respuestas a las solicitudes de HTTP. Pueden tener acceso a datos proporcionados mediante formularios HTML, aplicar la "lógica comercial" para decidir la respuesta que deben dar y generar el resultado HTML que se devolverá al navegador.

La Figura 8.6 muestra un ejemplo de código de servlet para implementar el formulario de la Figura 8.2. El servlet se denomina ConsultaBancoServlet, mientras que el formulario especifica que action="ConsultaBanco". Se debe indicar al servidor Web que este servlet se va a utilizar para tratar las solicitudes de ConsultaBanco. El formulario especifica que se usa el mecanismo HTTP get para transmitir los parámetros. Por tanto, se invoca el método doGet() del servlet, que se define en el código.

Cada solicitud da lugar a una nueva hebra en la que se ejecuta la llamada, por lo que se pueden tratar en paralelo varias solicitudes. Los valores de los menús del formulario y de los campos de entrada de la página Web, así como las cookies, se comunican a un objeto de la clase HttpServletRequest que se crea para la solicitud, y la respuesta a la solicitud se comunica a un objeto de la clase HttpServletResponse.

El método doGet() del ejemplo extrae los valores del tipo y del número del parámetro mediante request.getParameter(), y los utiliza para realizar una consulta a la base de datos. El código para tener acceso a la base de datos no se muestra; hay que consultar el Apartado 4.5.2 para conseguir detalles del modo de uso de JDBC para tener acceso a las bases de datos. El sistema devuelve el resultado de la consulta al solicitante imprimiéndolos en formato HTML en la respuesta del objeto HttpServletResponse.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ConsultaBancoServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        String type = request.getParameter("type");
        String number = request.getParameter("number");
        ... código para buscar el saldo del préstamo/cuenta ...
        ... uso de JDBC para comunicarse con la base de datos ..
        ... se da por supuesto que el valor se guarda en la variable saldo

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HEAD><TITLE> Resultado de la consulta</TITLE></HEAD>");
        out.println("<BODY>");
        out.println("Saldo de " + type + number + " = " + saldo);
        out.println("</BODY>");
        out.close();
    }
}
```

Figura 8.6 Ejemplo de código de servlet.

8.4.2 Sesiones de los servlets

Recuérdese que la interacción entre el navegador y el servidor Web carece de estado. Es decir, cada vez que el navegador formula una solicitud al servidor, necesita conectarse al servidor, solicitar alguna información y desconectarse del servidor. Se pueden emplear cookies para reconocer que una solicitud dada procede de la misma sesión de navegador que otra anterior. Sin embargo, las cookies constituyen un mecanismo de bajo nivel y los programadores necesitan una abstracción mejor para tratar con las sesiones.

La API del servlet ofrece un método para realizar el seguimiento de las sesiones y almacenar la información relacionada con ellas. La invocación del método getSession(false) de la clase HttpServletRequest recupera el objeto HttpSession correspondiente al navegador que envió la solicitud. Un valor del argumento de true habría especificado que había que crear un nuevo objeto sesión si la solicitud fuera nueva. Cuando se invoca el método getSession(), el servidor pide primero al cliente que devuelva una cookie con un nombre concreto.

Si el cliente no tiene ninguna cookie con ese nombre, o devuelve un valor que no corresponde al de ninguna sesión abierta, la solicitud no forma parte de ninguna sesión abierta. En ese caso, el servlet puede dirigir al usuario a la página de inicio de sesión, que puede permitir que el usuario presente su nombre de usuario y su contraseña. El servlet correspondiente a la página de inicio de sesión puede comprobar que la contraseña coincida con la del usuario (por ejemplo, buscando la información de autenticación en la base de datos).

Se pueden utilizar otros métodos de autenticación de usuarios. Si el usuario se autentifica correctamente, el servlet de inicio de sesión ejecuta getSession(cierto), que devuelve un nuevo objeto sesión. Para crear una nueva sesión el servidor Web ejecuta internamente las tareas siguientes: definir una cookie (denominada, por ejemplo, idsesión) con un identificador de sesión como valor asociado en el navegador cliente, crear un nuevo objeto sesión y asociar el valor del identificador de sesión con el objeto sesión.

El código del servlet también puede almacenar y buscar pares (nombre—atributo, valor) en el objeto HttpSession, para mantener el estado entre varias solicitudes de una misma sesión. Por ejemplo, el servlet del inicio de sesión puede almacenar el identificador de usuario del usuario como parámetro de la sesión ejecutando el método session.setAttribute("idusuario", idusuario) sobre el objeto sesión (en el que la variable de Java idususuario contiene el identificador del usuario), una vez autentificado el usuario y creado el objeto sesión.

Si la solicitud formara parte de una sesión abierta, el navegador habría devuelto el valor de la cookie y el servidor Web habría devuelto el objeto sesión correspondiente. El servlet puede recuperar entonces los parámetros de la sesión, como el identificador de usuario, del objeto sesión ejecutando el método session.getAttribute("idusuario"). Si el atributo idusuario no está definido, la función devuelve un valor nulo, lo que indica que el usuario del cliente no se ha autentificado.

8.4.3 Ciclo de vida de los servlets

El ciclo de vida de cada servlet está controlado por el servidor Web en el que se ha implantado. Cuando hay una solicitud de un cliente para un servlet concreto, el servidor comprueba primero si existe algún ejemplar de ese servlet. Si no existe, el servidor Web carga la clase del servlet en la máquina virtual de Java (Java virtual machine, JVM) y crea un ejemplar de la clase del servlet. Además, el servidor llama al método init() para inicializar el ejemplar del servlet. Téngase en cuenta que cada ejemplar del servlet sólo se inicializa una vez, cuando se carga.

Tras asegurarse de que existe el ejemplar del servlet, el servidor invoca el método service del servlet, con un objeto request y un objeto response como parámetros. De manera predeterminada, el servidor crea una hebra nueva para ejecutar el método service; por tanto, se pueden ejecutar en paralelo varias solicitudes al servlet, sin tener que esperar que solicitudes anteriores completen su ejecución. El método service llama a doGet o a doPost, según corresponda.

Cuando ya no sea necesario, se puede cerrar el servlet llamando al método destroy(). El servidor se puede configurar para que cierre de manera automática el servlet si no se le han hecho solicitudes en un periodo de tiempo determinado; este periodo es un parámetro del servidor que se puede definir como se considere adecuado para la aplicación.

8.4.4 Soporte de los servlets

El programa de Java de Sun servletrunner ofrece un soporte mínimo para la ejecución de servlets y es una manera rápida de comenzar a trabajar con los servlets. Puede recibir solicitudes en el puerto que se especifique, invocar a los servlets correspondientes y devolver la respuesta al cliente.

Muchos servidores de aplicaciones ofrecen soporte predeterminado para los servlets. Entre ellos están el servidor de aplicaciones de código abierto JBoss, el servidor Web de Java de Sun, el servidor Enterprise de Netscape, el servidor de aplicaciones Weblogic de BEA, el servidor de aplicaciones de Oracle y el servidor de aplicaciones WebSphere de IBM. Uno de los motores de servlets independientes más populares es el servidor Tomcat del proyecto Jakarta de Apache, que es un proyecto libre de código abierto. Estos servidores de aplicaciones ofrecen gran variedad de servicios adicionales, aparte del soporte básico para servlets proporcionado por servletrunner. Por ejemplo, si se modifica el código de un servlet, pueden detectarlo y volver a compilar y a cargar el servlet de manera transparente. Como ejemplo adicional, muchos servidores de aplicaciones permiten que el servidor se ejecute en varias máquinas para mejorar el rendimiento y encaminar las solicitudes a la copia adecuada. También ofrecen funcionalidad para controlar el estado del servidor de aplicaciones, incluidas las estadísticas de rendimiento. Muchos servidores de aplicaciones soportan también la plataforma Java 2 Enterprise Edition (J2EE), que ofrece soporte y APIspara gran variedad de tareas, como el manejo de objetos, el procesamiento en paralelo en varios servidores de aplicaciones y el manejo de datos en XML (XML se describe más adelante, en el Capítulo 10).

8.4.5 Secuencias de comandos en el lado del servidor

La escritura incluso de una mera aplicación Web en un lenguaje de programación como Java o C supone una tarea que consume bastante tiempo y necesita muchas líneas de código y programadores familiarizados con las complejidades del lenguaje. Un enfoque alternativo, el de los **guiones en el lado del servidor**, ofrece un método mucho más sencillo para la creación de múltiples aplicaciones. Los lenguajes de guiones ofrecen estructuras que pueden incluirse en los documentos HTML. En los guiones en el lado del servidor, antes de entregar una página Web, el servidor ejecuta las secuencias incluidas en el contenido HTML de la página. Cada secuencia, al ejecutarse, puede generar texto que se añade a la página (o que incluso puede eliminar contenido de la página). El código fuente de los guiones se elimina de la página, de modo que puede que el cliente ni siquiera se dé cuenta de que la página contenía originalmente código. Puede que el guión ejecutado contenga código SQL que se ejecute en una base de datos.

En los últimos años han aparecido varios lenguajes de guiones, como Server-Side Javascript de Netscape, JScript de Microsoft, Java Server Pages (JSP) de Sun, el preprocesador de hipertexto PHP (PHP Hypertext Preprocessor, PHP), el lenguaje de marcas de ColdFusion (ColdFusion Markup Language, CFML) y DTML de Zope. De hecho, incluso es posible incluir en las páginas HTML código escrito en lenguajes de guiones más antiguos como VBScript, Perl y Python. Por ejemplo, Active Server Pages (ASP) de Microsoft soporta VBScript y JScript incorporados. Estos lenguajes soportan características parecidas, pero se diferencian en el estilo de programación y en la facilidad con la que se pueden crear aplicaciones sencillas.

A continuación se describe brevemente **Java Server Pages** (**JSP**), un lenguaje de guiones que permite que los programadores de HTML mezclen HTML estático con el generado de manera dinámica. El motivo es que, en muchas páginas Web dinámicas, la mayor parte del contenido sigue siendo estático (es decir, se halla presente el mismo contenido siempre que se genera la página). El contenido dinámico de la página Web (que se genera, por ejemplo, en función de los parámetros de los formularios) suele ser una pequeña parte de la página. La creación de estas páginas mediante la escritura de código de servlet da lugar a grandes cantidades de HTML que se codifican como cadenas de Java. Por el contrario, JSP permite que el código Java se incorpore al HTML estático; el código Java incorporado genera la parte dinámica de la página. Los guiones de JSP realmente se traducen en código de servlet que luego se compila, pero se le ahorra al programador de la aplicación el problema de tener que escribir gran parte del código para crear el servlet.

La Figura 8.7 muestra el texto fuente de una página HTML que incluye un guión JSP. El código Java se distingue del de HTML circundante por estar encerrado entre $<\%\ldots\%>$. El guión llama a re-

quest.getParameter() para obtener el valor del atributo nombre. En función de ese valor, el guión decide lo que se debe escribir después de "Hola". Un ejemplo más realista puede llevar a cabo acciones más complejas, como buscar valores en una base de datos usando JDBC.

JSP también soporta el concepto de *biblioteca de etiquetas* (*tag library*), que permite el uso de etiquetas que se parecen mucho a las etiquetas HTML, pero se interceptan en el servidor y se sustituyen por el código HTML correspondiente. JSP ofrece un conjunto estándar de etiquetas que definen variables y controlan el flujo (iteradores, si-entonces-si_no), junto con un lenguaje de expresiones basado en Javascript (pero interpretado en el servidor). El conjunto de etiquetas es extensible, y ya se han implementado varias bibliotecas de etiquetas. Por ejemplo, existe una biblioteca de etiquetas que soporta la visualización paginada de conjuntos de datos de gran tamaño y una biblioteca que simplifica la visualización y el análisis de fechas y de horas. Véanse las notas bibliográficas para conocer más referencias de información sobre las bibliotecas de etiquetas de JSP.

8.5 Creación de aplicaciones Web de gran tamaño

Al crear aplicaciones Web, gran parte del esfuerzo de programación se dedica a la interfaz de usuario, en lugar de a las tareas relacionadas con la base de datos. En la primera parte de este apartado se estudian modos de reducir el esfuerzo de programación dedicado a esta tarea. Más adelante se describen algunas técnicas para mejorar el rendimiento de las aplicaciones.

8.5.1 Creación de interfaces Web

A continuación se describen varias técnicas para reducir el esfuerzo de programación para la creación de la interfaz de usuario.

Muchas estructuras HTML se generan mejor usando correctamente las funciones definidas de Java, en lugar de escribiéndolas como parte del código de cada página Web. Por ejemplo, los formularios de direcciones suelen necesitar un menú que contenga los nombres de países o de estados. En lugar de escribir un largo código HTML para crear el menú necesario cada vez que se utilice, es preferible definir una función que genere el menú y llamar a esa función siempre que sea necesario.

A menudo los menús se generan mejor a partir de datos de la base de datos, como una tabla que contenga los nombres de los países o de los estados. La función que genera el menú ejecuta una consulta a la base de datos y rellena el menú usando el resultado de la consulta. Añadir luego un país o un estado sólo exige una modificación de la base de datos, no del código de la aplicación. Este enfoque tiene el posible inconveniente de exigir mayor interacción con la base de datos, pero estas sobrecargas se pueden minimizar guardando en la memoria caché del servidor de aplicaciones el resultado de la consulta.

Los formularios para la introducción de fechas y de horas, o las introducciones de datos que exigen validación, se generan también mejor llamando a funciones definidas de manera adecuada. Esas funciones pueden generar código Javascript para llevar a cabo la validación en el generador.

```
<html>
<head> <title> Hola </title> </head>
<body>
<H1>
 < % if (request.getParameter("nombre") == null)
   { out.println("Hola, mundo"); }
   else { out.println("Hola, " + request.getParameter("nombre")); }
 %>
</H1>
</body>
</html>
```

Figura 8.7 Texto fuente de HTML con un guión JSP.

La visualización de conjuntos de resultados de consultas es una tarea habitual de muchas aplicaciones de bases de datos. Se puede crear una función genérica que tome una consulta SQL (o ResultSet) como argumento y muestre las tuplas del resultado de la consulta (o ResultSet) en forma tabular. Las llamadas a metadatos JDBC se pueden emplear para obtener información como el número de columnas y el nombre y el tipo de las columnas del resultado de la consulta; esa información se usa luego para mostrar ese mismo resultado.

Para manejar las situaciones en las que el resultado de la consulta es de gran tamaño se puede permitir la *paginación* del resultado en las funciones de visualización. La función puede mostrar un número fijo de registros en cada página y ofrecer controles para pasar a las páginas anteriores o siguientes o saltar hasta una página concreta del resultado.

Por desgracia no hay una API de Java estándar (muy usada) para que las funciones lleven a cabo las tareas de interfaz de usuario que se han descrito. La creación de esa biblioteca puede ser un proyecto de programación interesante.

8.5.2 Active Server Pages de Microsoft

Active Server Pages (ASP) de Microsoft, y su versión más reciente, Active Server Pages.NET (ASP.NET), son una alternativa a JSP/Java muy utilizada. ASP.NET es similar a JSP en el sentido de que el código de lenguajes como Visual Basic o C# se puede incorporar en HTML. Además, ASP.NET ofrece gran variedad de controles (comandos de guiones) que se interpretan en el servidor y generan HTML que se envía al cliente. Esos controles pueden simplificar de manera significativa la creación de interfaces Web. A continuación se ofrece una breve visión general de las ventajas que ofrecen estos controles.

Por ejemplo, controles como los menús desplegables y los cuadros de lista se pueden asociar con objetos DataSet. El objeto DataSet es parecido al objeto ResultSet de JDBC, y se suele crear mediante la ejecución de una consulta a la base de datos. El contenido del menú HTML se genera posteriormente a partir del contenido del objeto DataSet; por ejemplo, la consulta puede recuperar el nombre de todos los departamentos de una organización en el DataSet y el menú asociado contendría esos nombres. Por tanto, los menús que dependen del contenido de la base de datos se pueden crear de manera cómoda con muy poca programación.

Es posible añadir controles de validación para crear campos de entrada de datos; estos campos especifican de manera declarativa las restricciones de validez tales como rangos de valores o forzar la introducción de valores. El servidor crea el código HTML correspondiente combinado con JavaScript para llevar a cabo la validación en el navegador del usuario. Es posible asociar con cada control de validación los mensajes de error que se deben mostrar cuando se introducen datos no válidos.

Es posible especificar que las acciones de los usuarios tengan una acción asociada en el servidor. Por ejemplo, el control de un menú puede especificar que la selección de un valor de un menú tenga una acción asociada en la parte del servidor (esto se implementa mediante código JavaScript generado por el servidor). Es posible asociar con la acción del servidor el código Visual Basic o C# que muestra los datos correspondientes al valor seleccionado. Por tanto, la selección de un valor de un menú puede hacer que los datos asociados de la página se actualicen, sin necesidad de que el usuario pulse el botón de envío.

El control DataGrid ofrece una manera muy cómoda de mostrar el resultado de las consultas. Se asocia un control DataGrid con cada objeto DataSet, que suele ser el resultado de una consulta. El servidor genera código HTML que muestra el resultado de la consulta en forma de tabla. Los encabezados de las columnas se generan de manera automática a partir de los metadatos del resultado de la consulta. Además, los controles DataGrids presentan varias características, como la paginación, y permiten que el usuario ordene el resultado de las columnas elegidas. Todo el código HTML y la funcionalidad de la parte del servidor para la implementación de estas características los genera de manera automática el servidor. El control DataGrid permite incluso que los usuarios editen los datos y devuelvan las modificaciones al servidor. El desarrollador de la aplicación puede especificar una función, que se ejecutará cuando se edite una fila, que puede llevar a cabo la actualización de la base de datos.

Visual Studio de Microsoft ofrece una interfaz gráfica de usuario para la creación de páginas de ASP que usan estas características, lo que reduce aún más el esfuerzo de programación.

Véanse las notas bibliográficas para tener referencias de más información sobre ASP.NET.

8.5.3 Mejora del rendimiento de las aplicaciones

Son millones de personas las que pueden tener acceso a los sitios Web en todo el mundo, con tasas de millares de solicitudes por segundo, o incluso mayores, en el caso de los sitios más populares. Garantizar que las solicitudes se atiendan con tiempos de respuesta bajos supone un importante desafío para los desarrolladores de los sitios Web.

Se usan técnicas de almacenamiento caché de varios tipos para aprovechar las características comunes de las diversas transacciones. Por ejemplo, supóngase que el código de la aplicación para la atención de cada solicitud de usuario necesita contactar con una base de datos mediante JDBC. La creación de cada nueva conexión JDBC puede tardar varios milisegundos, por lo que la apertura de una conexión nueva para cada solicitud de usuario no es una buena idea si hay que permitir altas tasas transaccionales.

La **agrupación de conexiones** se usa para reducir esta sobrecarga, que funciona de la manera siguiente. El gestor del grupo de conexiones (parte del servidor de aplicaciones) crea un grupo (es decir, un conjunto) de conexiones abiertas ODBC o JDBC. En lugar de abrir una conexión nueva con la base de datos, el código que atiende las solicitudes de los usuarios (generalmente un servlet) pide (solicita) una conexión al grupo de conexiones y se la devuelve cuando el código (servlet) completa el procesamiento. Si el grupo no tiene conexiones sin usar cuando se le solicita una, se abre una conexión nueva con la base de datos (procurando no superar el número máximo de conexiones que puede soportar el sistema de bases de datos de manera concurrente). Si hay muchas conexiones abiertas que no se han usado durante algún tiempo, el gestor del grupo de conexiones puede cerrar algunas de las conexiones abiertas con la base de datos. Muchos servidores de aplicaciones, y los controladores ODBC y JDBC más recientes, ofrecen un gestor de grupo de conexiones incorporado.

Un error habitual que cometen muchos programadores al crear aplicaciones Web es olvidarse de cerrrar las conexiones JDBC abiertas (o, lo que es lo mismo, cuando se usa la agrupación de conexiones, olvidarse de devolver la conexión al grupo de conexiones). Cada solicitud abre entonces una conexión nueva con la base de datos, y ésta alcanza pronto el límite de conexiones que puede tener abiertas simultáneamente. Estos problemas no suelen manifestarse en las pruebas a pequeña escala, ya que las bases de datos suelen permitir centenares de conexiones abiertas, sino que sólo se manifiestan con un uso intensivo. Algunos programadores dan por supuesto que las conexiones, como la memoria asignada a los programas de Java, son basura que se recoge de manera automática. Por desgracia, esto no sucede, y los programadores son responsables del cierre de las conexiones que han abierto.

Puede que algunas solicitudes den como resultado que se vuelva a enviar exactamente la misma consulta a la base de datos. El coste de la comunicación con la base de datos puede reducirse enormemente guardando en la caché los resultados de consultas anteriores y volviendo a usarlos, siempre y cuando no haya cambiado el resultado de la consulta en la base de datos. Algunos servidores Web soportan este almacenamiento en la caché del resultado de las consultas.

Se pueden reducir aún más los costes guardando en la caché la página Web final que se envía en respuesta a las consultas. Si llega una consulta nueva exactamente con los mismos parámetros que una consulta anterior y la página Web resultante se halla en la caché, puede volver a usarse para evitar el coste de volver a calcular la página. El almacenamiento en la caché se puede hacer en el nivel de los fragmentos de las páginas Web, que luego se combinan para crear páginas Web completas.

Los resultados de las consultas y las páginas Web guardados en la caché son modalidades de las vistas materializadas. Si la base de datos subyacente se modifica, pueden descartarse o volver a calcularse o, incluso, actualizarse de modo incremental, como en el mantenimiento de las vistas materializadas (que se describe más adelante, en el Apartado 14.5). Algunos sistemas de bases de datos (como SQL Server de Microsoft) ofrecen una manera de que el servidor de aplicaciones registre las consultas a la base de datos y obtenga una **notificación** de la base de datos cuando se modifique el resultado de alguna consulta. Este mecanismo de notificación se puede usar para garantizar que los resultados de consultas guardados en la caché del servidor de aplicaciones estén actualizados.

8.6 Disparadores

Un **disparador** es una instrucción que el sistema ejecuta de manera automática como efecto secundario de la modificación de la base de datos. Para diseñar un mecanismo disparador es necesario:

1. Especificar las condiciones en las que se va a ejecutar el disparador. Esto se descompone en un *evento* que provoca la comprobación del disparador y una *condición* que se debe cumplir para que se ejecute el disparador.
2. Especificar las *acciones* que se van a realizar cuando se ejecute el disparador.

Este modelo de disparadores se denomina modelo **evento-condición-acción** de los disparadores.

La base de datos almacena los disparadores como si fuesen datos normales, por lo que son persistentes y están accesibles para todas las operaciones de la base de datos. Una vez que se introduce un disparador en la base de datos, el sistema de bases de datos asume la responsabilidad de ejecutarlo cada vez que se produzca el evento especificado y se satisfaga la condición correspondiente.

8.6.1 Necesidad de los disparadores

Los disparadores son mecanismos útiles para alertar a los usuarios o para iniciar de manera automática ciertas tareas cuando se cumplen determinadas condiciones. A modo de ejemplo, supóngase que, en lugar de permitir saldos de cuenta negativos, el banco trata los descubiertos dejando a cero el saldo de la cuenta y creando un préstamo por el importe del descubierto. El banco da a este préstamo un número de préstamo idéntico al número de cuenta que ha tenido el descubierto. En este ejemplo la condición para ejecutar el disparador es una actualización de la relación *cuenta* que da lugar a un valor negativo de *saldo*. Supóngase que Santos retiró cierta cantidad de dinero de una cuenta que dio lugar a que el saldo de la cuenta fuera negativo y que t denota la tupla de cuenta con un valor negativo de *saldo*. Las acciones que hay que emprender son:

- Insertar una nueva tupla s en la relación *préstamo* con

  $$s[\textit{número_préstamo}] = t[\textit{número_cuenta}]$$
  $$s[\textit{nombre_sucursal}] = t[\textit{nombre_sucursal}]$$
  $$s[\textit{importe}] = -t[\textit{saldo}]$$

  Obsérvese que, dado que t[*saldo*] es negativo, hay que cambiar el signo de t[*saldo*] para obtener el importe del préstamo—un número positivo.

- Insertar una nueva tupla u en la relación *prestatario* con

  $$u[\textit{nombre_cliente}] = \text{“Santos”}$$
  $$u[\textit{número_préstamo}] = t[\textit{número_cuenta}]$$

- Dejar t[*saldo*] a 0.

Como ejemplo adicional del uso de disparadores, supóngase un almacén que desee mantener unas existencias mínimas de cada producto; cuando el nivel de existencias de un producto cae por debajo del nivel mínimo, se debe realizar un pedido de manera automática. Ésta es la manera en que esta regla empresarial se puede implementar mediante disparadores: al actualizar el nivel de existencias de cada producto el disparador debe comparar ese nivel con el nivel de existencias mínimo del producto y, si se halla en el mínimo o por debajo de él, se añade un nuevo pedido a la relación *pedidos*.

Obsérvese que los sistemas de disparadores en general no pueden realizar actualizaciones fuera de la base de datos y, por tanto, en el ejemplo de reposición de las existencias, no se puede usar un disparador para realizar directamente un pedido al exterior. En vez de eso, se añade un pedido a la relación *pedidos*, como en el ejemplo del inventario. Se debe crear un proceso del sistema separado que se esté ejecutando constantemente que explore periódicamente la relación *pedidos* y formule los pedidos. Este proceso del sistema también debe tener en cuenta las tuplas de la relación *pedidos* que se han procesado y el momento de formulación de cada pedido. El proceso también debe realizar un seguimiento de las entregas de los pedidos y alertar a los gestores en caso de condiciones excepcionales, tales como los retrasos en las entregas. Algunos sistemas de bases de datos ofrecen soporte predeterminado para el envío de correo electrónico desde las consultas y los disparadores de SQL, usando el enfoque mencionado.

```
create trigger disparador_descubierto after update on cuenta
referencing new row as fila_nueva
for each row
when fila_nueva.saldo < 0
begin atomic
   insert into prestatario
          (select nombre_cliente, número_cuenta
           from impositor
           where fila_nueva.número_cuenta = impositor.número_cuenta);
    insert into préstamo values
          (fila_nueva.número_cuenta, fila_nueva.nombre_sucursal, − fila_nueva.saldo);
   update cuenta set saldo = 0
          where cuenta.número_cuenta = fila_nueva.número_cuenta
end
```

Figura 8.8 Ejemplo de la sintaxis SQL:1999 para los disparadores.

8.6.2 Los disparadores en SQL

Los sistemas de bases de datos basados en SQL emplean mucho los disparadores, aunque antes de SQL:1999 no eran parte de la norma de SQL. Por desgracia, cada sistema de bases de datos implementó su propia sintaxis para los disparadores, lo que produjo incompatibilidades. En la Figura 8.8 se describe la sintaxis de SQL:1999 para los disparadores (que es parecida a la sintaxis del sistema de bases de datos DB2 de IBM y a la de Oracle).

Esta definición de disparador especifica que el disparador se inicia *después* de la ejecución de cualquier actualización de la relación *cuenta*. Cada instrucción de actualización SQL puede actualizar varias tuplas de la relación, y la cláusula **for each row** del código del disparador se itera luego de manera explícita para cada fila actualizada. La cláusula **referencing new row as** crea la variable *fila_nueva* (denominada **transition variable**), que almacena el valor de la fila actualizada después de la actualización.

La instrucción **when** especifica una condición, en este caso $fila_nueva.saldo < 0$. El sistema ejecuta el resto del cuerpo del disparador sólo para las tuplas que satisfacen esa condición. La cláusula **begin atomic** ... **end** sirve para reunir varias instrucciones de SQL en una sola instrucción compuesta. Las dos instrucciones **insert** de la estructura **begin** ... **end** realizan las tareas específicas de creación de las nuevas tuplas de las relaciones *prestatario* y *préstamo* para que representen el nuevo préstamo. La instrucción **update** sirve para volver a dejar a cero el saldo de la cuenta tras su valor negativo anterior.

El evento y las acciones del disparador pueden adoptar muchas formas:

- El *evento* del disparador puede ser **insert** o **delete**, en lugar de **update**.

  Por ejemplo, la acción ante el borrado de una cuenta puede ser comprobar si los titulares de la cuenta tienen otras cuentas y, si no las tienen, borrarlos de la relación *impositor*. Se deja al lector la definición de este disparador como ejercicio (Ejercicio práctico 8.5).

  Como ejemplo adicional, si se inserta un nuevo *impositor*, la acción del disparador puede ser enviar una carta de bienvenida al impositor. Evidentemente, el disparador no puede provocar directamente esta acción fuera de la base de datos pero, en vez de eso, sí que puede añadir una nueva tupla a una relación que almacene las direcciones a las que se deben enviar las cartas de bienvenida. Un proceso diferente examinará esta relación e imprimirá las cartas que hay que enviar.

  Muchos sistemas de bases de datos soportan gran variedad de eventos de disparadores, como puede ser el que un usuario (aplicación) inicie una sesión en la base de datos (es decir, que abra una conexión), el que se cierre el sistema o el que se realicen modificaciones en la configuración del sistema.

- Para las actualizaciones el disparador puede especificar las columnas cuya actualización provoca su ejecución. Por ejemplo, si la primera línea del disparador de descubiertos se sustituyera por

create trigger *disparador_descubierto* **after update of** *saldo* **on** *cuenta*

el disparador se ejecutaría sólo cuando se actualizase *saldo*; las actualizaciones del resto de los atributos no provocarían su ejecución.

- La cláusula **referencing old row as** se puede utilizar para crear una variable que almacene el valor anterior de una fila actualizada o borrada. La cláusula **referencing new row as** se puede usar con las inserciones además de con las actualizaciones.

- Los disparadores se pueden activar antes (**before**) del evento (insert/delete/update) en lugar de después (**after**) del evento.

  Estos disparadores pueden servir como restricciones adicionales que pueden impedir actualizaciones no válidas. Por ejemplo, si no se desea permitir descubiertos, se puede crear un disparador **before** que compruebe si el nuevo saldo es negativo y, en caso de serlo, retroceda la transacción. Aunque se podría haber usado con este propósito un disparador **after**, su uso haría que primero se llevase a cabo la actualización y que luego se retrocediera la transacción.

  Como ejemplo adicional, supóngase que el valor del campo de número telefónico de una tupla insertada está vacío, lo que indica la ausencia de número de teléfono. Se puede definir un disparador que sustituya ese valor por un valor **nulo**. Se puede emplear la instrucción **set** para realizar estas modificaciones.

  create trigger *disparador_hacer_nulo* **before update on** *r*
  referencing new row as *fila_nueva*
  for each row
  when *fila_nueva.número_teléfono* = ' '
  set *fila_nueva.número_teléfono* = **null**

- En lugar de llevar a cabo una acción por cada fila afectada, se puede realizar una sola acción para toda la instrucción de SQL que ha causado la inserción, el borrado o la actualización. Para hacerlo se usa la cláusula **for each statement** en lugar de **for each row**.

  Las cláusulas **referencing old table as** o **referencing new table as** se pueden emplear para hacer referencia a tablas temporales (denominadas *tablas de transición*) que contengan todas las filas afectadas. Las tablas de transición no se pueden emplear con los disparadores **before**, pero sí con los disparadores **after**, independientemente de si son disparadores de instrucciones (**statement**) o de filas (**row**).

  Cada instrucción SQL se puede usar para llevar a cabo varias acciones con base en las tablas de transición.

- Los disparadores se pueden habilitar y deshabilitar; de manera predeterminada, al crearlos se habilitan, pero se pueden deshabilitar empleando **alter trigger** *nombre_disparador* **disable** (algunas bases de datos utilizando una sintaxis alternativa, como **disable trigger** *nombre_disparador*). Los disparadores deshabilitados se pueden volver a habilitar. Los disparadores también se pueden descartar, lo que los elimina de manera permanente, usando la instrucción **drop trigger** *nombre_disparador*.

Volviendo al ejemplo de inventario del almacén, supóngase que se tienen las siguientes relaciones:

- *existencias(producto, nivel)*.
- *nivel_mínimo(producto, nivel)*, que denota la cantidad mínima que se debe mantener de cada producto.
- *nuevo_pedido(producto, cantidad)*, que denota la cantidad de producto que se debe pedir cuando su nivel cae por debajo del mínimo.
- *pedidos(producto, cantidad)*, que denota la cantidad de producto que se debe pedir.

Obsérvese que se ha tenido mucho cuidado a la hora de realizar pedidos sólo cuando su cantidad cae desde un nivel superior al mínimo a otro inferior. Si sólo se comprueba si el nuevo valor después de la actualización está por debajo del mínimo, se puede realizar erróneamente un pedido cuando ya se ha realizado uno. Se puede usar el disparador mostrado en la Figura 8.9 para volver a pedir el producto.

Muchos sistemas de bases de datos ofrecen implementaciones no normalizadas de los disparadores, o implementan sólo parte de las características de los disparadores. Por ejemplo, muchos de los sistemas de bases de datos no implementan la cláusula **before**, y usan la palabra clave **on** en lugar de **after**. Puede que no implementen la cláusula **referencing**. En su lugar, puede que especifiquen las tablas de transición mediante las palabras clave **inserted** o **deleted**. La Figura 8.10 muestra la manera en que se escribiría el disparador de los descubiertos de las cuentas en SQL Server de Microsoft. Consúltese el manual del sistema de bases de datos que se esté usando para obtener más información sobre las características de los disparadores que soporta.

8.6.3 Cuándo no deben emplearse los disparadores

Existen muchas aplicaciones para los disparadores, como las que se acaban de ver en el Apartado 8.6.2, pero algunas se manejan mejor con técnicas alternativas. Por ejemplo, en el pasado, los diseñadores de sistemas empleaban los disparadores para realizar resúmenes de datos. Se empleaban disparadores bajo la inserción, borrado o actualización de una relación *empleado* que contenía los atributos *sueldo* y *departamento* para mantener el sueldo total de cada departamento. Sin embargo, muchos sistemas de bases de datos actuales soportan las vistas materializadas (véase el Apartado 3.9.1), que ofrecen una forma mucho más sencilla de realizar los resúmenes de datos. Los diseñadores también empleaban mucho los disparadores para las réplicas de las bases de datos; utilizaban disparadores bajo la inserción, borrado o actualización de las relaciones para registrar las modificaciones en relaciones denominadas **cambio** o **delta**. Un proceso diferente copiaba las modificaciones a la réplica (copia) de la base de datos, y el sistema ejecutaba los cambios en la réplica. Sin embargo, los sistemas de bases de datos modernos proporcionan características predeterminadas para la réplica de bases de datos, lo que hace que los disparadores resulten innecesarios para las réplicas en la mayor parte de los casos.

De hecho, muchas aplicaciones de los disparadores, incluyendo el ejemplo de los descubiertos, se pueden sustituir mediante el uso adecuado de los procedimientos almacenados. Por ejemplo, supóngase que las actualizaciones del atributo *saldo* de *cuenta* sólo se realizan mediante un procedimiento almacenado concreto. Ese procedimiento, a su vez, comprobará si el saldo es negativo y llevará a cabo las acciones correspondientes al desencadenador del descubierto. Los programadores deben tener cuidado con no actualizar directamente el valor de *saldo*, sino actualizarlo solamente mediante el procedimiento almacenado; esto se puede garantizar si no se proporciona a la aplicación (o al usuario) autorización

```
create trigger disparador_nuevo_pedido after update of cantidad on existencias
referencing old row as fila_vieja, new row as fila_nueva
for each row
when fila_nueva.nivel <= (select nivel
                          from nivel_mínimo
                          where nivel_minimo.producto = fila_vieja.producto)
and fila_vieja.nivel > (select nivel
                        from nivel_mínimo
                        where nivel_mínimo.producto = fila_vieja.producto)
begin
        insert into pedidos
                (select producto, cantidad
                 from nuevo_pedido
                 where nuevo_pedido.producto = fila_vieja.producto)
end
```

Figura 8.9 Ejemplo de disparador para hacer un nuevo pedido de un producto.

```
create trigger disparador_descubierto on cuenta
for update
as
if inserted.saldo < 0
begin
   insert into prestatario
          (select nombre_cliente, número_cuenta
           from impositor, inserted
           where inserted.número_cuenta = impositor.número_cuenta)
    insert into préstamo values
          (inserted.número_cuenta, inserted.nombre_sucursal, − inserted.saldo)
   update cuenta set saldo = 0
          from cuenta, inserted
          where cuenta.número_cuenta = inserted.número_cuenta
end
```

Figura 8.10 Ejemplo de disparador en la sintaxis de SQL Server de Microsoft.

para actualizar el atributo *saldo*, pero se le proporciona autorización para ejecutar el procedimiento almacenado asociado. Una encapsulación parecida se puede usar para sustituir el disparador *nuevo_pedido* mediante un procedimiento almacenado.

Otro problema con los disparadores es la ejecución no pretendida de la acción disparada cuando se cargan datos de una copia de seguridad, o cuando las actualizaciones de la base de datos de un sitio se replican en un sitio de copia de seguridad. En esos casos, la acción ya se ha ejecutado y, generalmente, no se debe volver a ejecutar. Al cargar los datos, los disparadores se pueden deshabilitar de manera explícita. Para los sistemas de réplica de copia de seguridad que puede que tengan que sustituir el sistema principal, hay que deshabilitar los disparadores en un principio y habilitarlos cuando el sitio de copia de seguridad sustituye al principal en el procesamiento. Como alternativa, algunos sistemas de bases de datos permiten que los disparadores se especifiquen como **no para réplica**, lo que garantiza que no se ejecuten en el sitio de copia de seguridad durante la réplica de la base de datos. Otros sistemas de bases de datos ofrecen una variable del sistema que denota que la base de datos es una réplica en la que se están repitiendo las acciones de la base de datos; el cuerpo del disparador debe examinar esta variable y salir si es verdadera. Ambas soluciones eliminan la necesidad de habilitar y deshabilitar de manera explícita los disparadores.

Los disparadores se deben escribir con sumo cuidado, dado que un error en un disparador detectado en tiempo de ejecución provoca el fallo de la instrucción de inserción, borrado o actualización que inició el disparador. Además, la acción de un disparador puede activar otro disparador. En el peor de los casos, esto puede dar lugar a una cadena infinita de disparos. Por ejemplo, supóngase que un disparador de inserción sobre una relación tiene una acción que provoca otra (nueva) inserción sobre la misma relación. La acción de inserción dispara otra acción de inserción, y así hasta el infinito. Generalmente, los sistemas de bases de datos limitan la longitud de las cadenas de disparadores (por ejemplo, a dieciséis o treinta y dos), y consideran que las cadenas de disparadores de mayor longitud son un error.

Los disparadores se denominan a veces *reglas*, o *reglas activas*, pero no se deben confundir con las reglas de Datalog (véase el Apartado 5.4), que son realmente definiciones de vistas.

8.7 Autorización en SQL

Se ha visto el conjunto básico de privilegios de SQL en el Apartado 4.3, incluidos los privilegios **delete**, **insert**, **select** y **update**.

Además de estas formas de privilegios para el acceso a los datos, se pueden conceder (conceptualmente) a los usuarios diferentes tipos de autorización para la modificación del esquema de la base de datos:

- Autorización para crear nuevas relaciones.

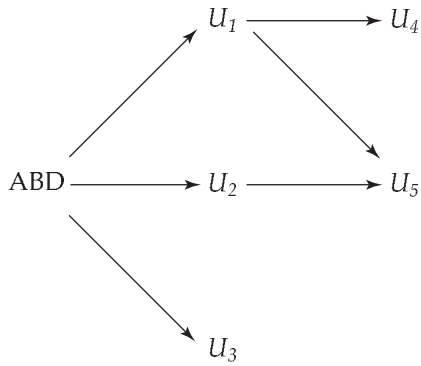


Figura 8.11 Grafo de concesión de autorizaciones.

- Autorización para añadir atributos a las relaciones o para eliminar atributos de las relaciones.
- Autorización para descartar relaciones.

La norma de SQL especifica un mecanismo de autorización primitivo para el esquema de la base de datos. Sólo el propietario del esquema puede llevar a cabo modificaciones del mismo. Por tanto, las modificaciones del esquema—como la creación o eliminación de relaciones, la adición o el descarte de atributos de las relaciones y la adición o descarte de índices—sólo puede ejecutarlas el propietario del esquema. Varias implementaciones de las bases de datos tienen mecanismos de autorización más potentes para los esquemas de las bases de datos, parecidos a los ya estudiados, pero esos mecanismos no están normalizados.

SQL también incluye un privilegio **references** que permite que los usuarios declaren claves externas al crear las relaciones. Inicialmente, puede parecer que no hay ninguna razón para impedir que los usuarios creen claves externas que hagan referencia a otras relaciones. Sin embargo, recuérdese que las restricciones de clave externa restringen las operaciones de borrado y de actualización de la relación a la que se hace referencia. Supóngase que U_1 crea una clave externa en la relación r que hace referencia al atributo *nombre_sucursal* de la relación *sucursal* y luego inserta una tupla en r correspondiente a la sucursal de Navacerrada. Ya no es posible borrar la sucursal de Navacerrada de la relación *sucursal* sin modificar también la relación r. Por tanto, la definición de una clave externa por U_1 restringe la actividad futura de otros usuarios; en consecuencia, el privilegio **references** es necesario.

El privilegio **references** sobre s también es necesario para crear restricciones **check** sobre una relación r si alguna restricción tiene una subconsulta que hace referencia a la relación s.

SQL define el privilegio **execute**; este privilegio autoriza a los usuarios a ejecutar funciones o procedimientos. Por tanto, sólo los usuarios que tienen el privilegio **execute** sobre la función $f()$ pueden llamarla (bien directamente, bien desde una consulta de SQL).

SQL también incluye el privilegio **usage**, que autoriza a los usuarios a usar el dominio especificado (recuérdese que los dominios se corresponden con el concepto de tipo de los lenguajes informáticos y pueden ser definidos por los usuarios).

La forma de autoridad definitiva es la dada al administrador de la base de datos. El administrador de la base de datos puede autorizar nuevos usuarios, reestructurar la base de datos, etc. Esta forma de autorización es análoga a la de **superusuario** u operador de un sistema operativo.

8.7.1 Concesión de privilegios

El usuario al que se le ha concedido alguna forma de autorización puede estar autorizado a transmitir esa autorización a otros usuarios. Sin embargo, hay que tener cuidado con el modo en que se puede transmitir esa autorización entre los usuarios para asegurar que la misma pueda retirarse en el futuro.

Considérese, a modo de ejemplo, la concesión de la autorización de actualización sobre la relación *préstamo* de la base de datos bancaria. Supóngase que, inicialmente, el administrador de la base de datos concede la autorización de actualización sobre *préstamo* a los usuarios U_1, U_2 y U_3, que, a su vez, pueden transmitirla a otros usuarios. La transmisión de la autorización de un usuario a otro puede representarse

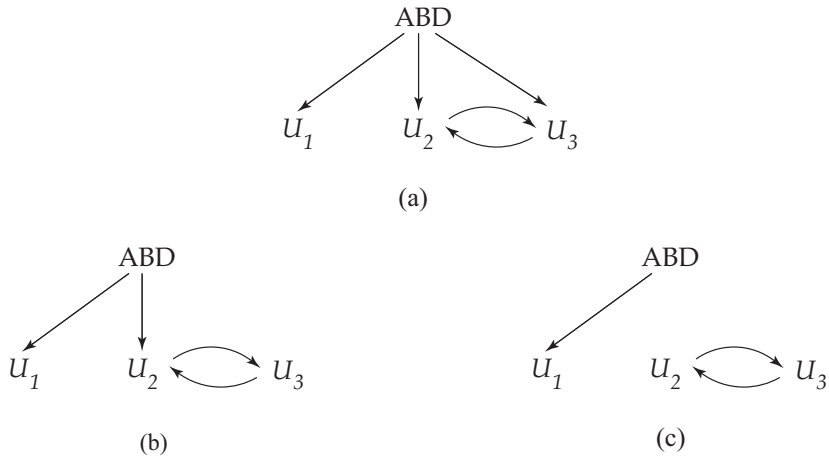


Figura 8.12 Intento de eludir la retirada de autorizaciones.

mediante un **grafo de autorización**. Los nodos de este grafo son los usuarios. El grafo incluye un arco $U_i \rightarrow U_j$ si el usuario U_i concede la autorización de actualización sobre *préstamo* a U_j. La raíz del grafo es el administrador de la base de datos. En el grafo de ejemplo de la la Figura 8.11 se puede observar que tanto U_1 como U_2 conceden autorización al usuario U_5; sólo U_1 concede autorización a U_4.

Un usuario tiene autorización *si y sólo si* hay un camino desde la raíz del grafo de autorización (el nodo que representa al administrador de la base de datos) hasta el nodo que representa a ese usuario.

Supóngase que el administrador de la base de datos decide revocar la autorización del usuario U_1. Como U_4 ha recibido la autorización de U_1, también hay que revocar esa autorización. Sin embargo, a U_5 le concedieron la autorización tanto U_1 como U_2. Dado que el administrador de la base de datos no ha revocado la autorización sobre *préstamo* a U_2, U_5 conserva la autorización de actualización sobre *préstamo*. Si U_2 revocara la autorización a U_5, U_5 perdería esa autorización.

Un par de usuarios taimados puede intentar eludir las reglas de retirada de autorizaciones concediéndose autorización mutuamente, como se muestra en la Figura 8.12a. Si el administrador de la base de datos retira la autorización a U_2, U_2 la conserva mediante U_3, como se muestra en la Figura 8.12b. Si a continuación se retira la autorización a U_3, U_3 parece conservarla mediante U_2, como se muestra en la Figura 8.12.c. Sin embargo, cuando el administrador retira la autorización a U_3, los arcos de U_3 a U_2 y de U_2 a U_3 ya no forman parte de ningún camino que comience en el administrador de la base de datos. Se exige que todos los arcos de los grafos de autorización sean parte de algún camino que comience en el administrador de la base de datos. Los arcos entre U_2 y U_3 se han borrado y el grafo de autorización resultante queda como se muestra en la Figura 8.13.

8.7.2 Concesión de privilegios en SQL

En el Apartado 4.3 se ha visto la sintaxis básica de SQL para la concesión y retirada de los privilegios. Recuérdese que la instrucción **grant** se usa para conceder autorizaciones. Su forma básica es:

grant <lista de privilegios> **on** <nombre de la relación o de la vista> **to** <lista de usuarios o de roles>

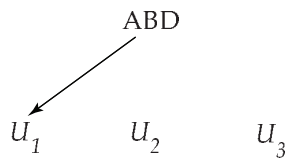


Figura 8.13 Grafo de autorización.

La *lista de privilegios* permite conceder varios privilegios en una sola instrucción. El concepto de roles se trata más adelante, en el Apartado 8.7.3.

La siguiente instrucción **grant** concede a los usuarios U_1, U_2 y U_3 el privilegio **select** sobre la relación *cuenta*:

grant select on *cuenta* **to** U_1, U_2, U_3

Se puede conceder el privilegio **update** sobre todos los atributos de la relación o sólo sobre algunos. Si se incluye el privilegio **update** en una sentencia **grant**, la lista de atributos sobre los que se va a conceder la autorización de actualización aparece opcionalmente entre paréntesis justo detrás de la palabra clave **update**. Si se omite la lista de atributos, se concede el privilegio de actualización sobre todos los atributos de la relación.

Esta instrucción **grant** concede a los usuarios U_1, U_2 y U_3 autorización de actualización sobre el atributo *importe* de la relación *préstamo*:

grant update (*importe*) **on** *préstamo* **to** U_1, U_2, U_3

El privilegio **insert** también puede especificar una lista de atributos; las inserciones que se hagan en la relación sólo deben especificar esos atributos, el sistema dará a los restantes atributos valores predeterminados (si se han definido para cada atributo) o los definirá como nulos. El nombre de usuario **public** hace referencia a todos los usuarios actuales y futuros del sistema. Por tanto, los privilegios concedidos a **public** se conceden de manera implícita a todos los usuarios actuales y futuros.

El privilegio de SQL **references** se concede sobre atributos concretos de manera parecida a la del privilegio **update**. La siguiente instrucción **grant** permite que el usuario U_1 cree relaciones que hagan referencia a la clave *nombre_sucursal* de la relación *sucursal* como clave externa:

grant references (*nombre_sucursal*) **on** *sucursal* **to** U_1

De manera predeterminada, los usuarios o roles a los que se les conceden privilegios no están autorizados a concedérselos a otros usuarios o roles. Si se desea conceder un privilegio y permitir que el receptor se lo conceda a otros usuarios, hay que añadir la cláusula **with grant option** a la instrucción **grant** correspondiente. Por ejemplo, si se desea conceder a U_1 el privilegio **select** sobre *sucursal* y permitir que U_1 les conceda a otros ese privilegio, hay que escribir

grant select on *sucursal* **to** U_1 **with grant option**

El creador de un objeto (relación/vista/rol) tiene todos los privilegios sobre ese objeto, incluido el privilegio de conceder privilegios a otros.

8.7.3 Roles

Considérese un banco que tiene muchos cajeros. Cada cajero debe tener el mismo tipo de autorización para el mismo conjunto de relaciones. Cada vez que se contrata a un cajero nuevo hay que concederle todas esas autorizaciones una a una.

Un esquema mejor sería especificar las autorizaciones que deben tener los cajeros e identificar por separado a los usuarios de la base de datos que son cajeros. El sistema puede usar estos dos fragmentos de información para determinar las autorizaciones de cada persona que sea cajera. Cuando se contrate a una nueva persona como cajero, se le debe asignar un identificador de usuario e identificarlo como cajero. No es necesario volver a especificar cada uno de los permisos concedidos a los cajeros.

El concepto de **roles** captura este esquema. En la base de datos se crea un conjunto de roles. Las autorizaciones se pueden conceder a los roles exactamente igual que se conceden a los usuarios. Se concede un conjunto de roles (que puede estar vacío) que está autorizado a desempeñar a cada usuario de la base de datos.

En la base de datos bancaria entre los ejemplos de roles están *cajero*, *director_sucursal*, *interventor* y *administrador_del_sistema*.

Una alternativa menos preferible sería crear el identificador de usuario *cajero* y permitir que cada cajero se conectase a la base de datos usando ese identificador. El problema con este esquema es que no sería posible identificar exactamente al cajero que ha realizado una determinada transacción, lo que puede provocar problemas de seguridad. El uso de los roles tiene la ventaja de exigir a los usuarios que se conecten con su propio identificador de usuario.

Cualquier autorización que se pueda conceder a los usuarios se puede conceder a los roles. Los roles se conceden a los usuarios igual que las autorizaciones. Y, como otras autorizaciones, también se les puede conceder a los usuarios autorización para conceder un rol concreto a otros. Por tanto, se puede conceder a los directores de las sucursales autorización para conceder el rol *cajero*.

En SQL:1999 los roles se pueden crear de la manera siguiente:

create role *cajero*

Se pueden conceder privilegios a los roles igual que a los usuarios, como se muestra en esta instrucción:

grant select on *cuenta*
to *cajero*

Se pueden conceder roles a los usuarios y a otros roles, como muestran estas instrucciones:

grant *cajero* **to** Martín
create role *director*
grant *cajero* **to** *director*
grant *director* **to** María

Por tanto, los privilegios de los usuarios o roles consisten en:

- Todos los privilegios concedidos directamente al usuario o rol.
- Todos los privilegios concedidos a los roles que se han concedido al usuario o rol.

Téngase en cuenta que puede haber cadenas de roles; por ejemplo, se puede conceder el rol *empleado* a todos los *cajeros*. A su vez, el rol *cajero* se concede a todos los *director*es. Por tanto, el rol *director* hereda todos los privilegios concedidos a los roles *empleado* y *cajero*, además de los concedidos directamente a *director*.

Las acciones ejecutadas por las sesiones tienen todos los privilegios concedidos directamente usuario que las ha abierto, así como todos los privilegios concedidos a los roles que se hayan concedido (directa o indirectamente, a través de otros roles) a ese usuario. Por tanto, si al usuario Martín se le ha concedido el rol de *director*, las acciones ejecutadas por el usuario Martín tienen todos los privilegios concedidos a *director*, además de los privilegios concedidos a *cajero* si se ha concedido el rol de *cajero* al rol de *director*.

Además del concepto de usuario (actual) de la sesión, SQL también tiene el concepto del rol actual asociado a la sesión. De manera predeterminada, el rol actual asociado con una sesión es nulo (excepto en algunos casos especiales). El rol actual asociado con la sesión se puede definir ejecutando **set role** *nombre_rol*. El rol especificado debe haberse concedido al usuario, o la instrucción **set role** fallará.

Cuando se conceden privilegios se tratan de manera predeterminada como si los hubiera concedido el usuario actual, es decir, el concedente es el usuario actual. Para conceder privilegios con el concedente definido como el rol actual asociado con una sesión se puede añadir la cláusula **granted by current_role** a la instrucción grant, siempre y cuando el rol actual no sea nulo. El motivo de que se especifique que el concedente de un privilegio es un rol concreto quedará claro más adelante, cuando se estudie la retirada de los privilegios.

8.7.4 Retirada de los privilegios

Para retirar autorizaciones se usa la instrucción **revoke**. Adopta una forma casi idéntica a la de **grant**:

revoke <lista de privilegios> **on** <nombre de la relación o de la vista>
from <lista de usuarios o de roles> [**restrict** | **cascade**]

Por tanto, para retirar los privilegios concedidos anteriormente hay que escribir

revoke select on *sucursal* **from** U_1, U_2, U_3
revoke update (*importe*) **on** *préstamo* **from** U_1, U_2, U_3
revoke references (*nombre_sucursal*) **on** *sucursal* **from** U_1

Como se ha visto en el Apartado 8.7.1, la retirada de privilegios de un usuario o rol puede provocar que otros usuarios o roles también pierdan esos privilegios. Este comportamiento se denomina *retirada en cascada*. En la mayor parte de los sistemas de bases de datos la retirada en cascada es el comportamiento predeterminado; por tanto, se puede omitir la palabra clave **cascade**, como se ha hecho en los ejemplos anteriores. La instrucción **revoke** puede especificar también **restrict**:

revoke select on *sucursal* **from** U_1, U_2, U_3 **restrict**

En este caso, el sistema devuelve un error si hay retiradas en cascada y no lleva a cabo la acción de retirada. La instrucción **revoke** siguiente sólo retira la opción grant, no el privilegio **select** propiamente dicho:

revoke grant option for select on *sucursal* **from** U_1

La retirada en cascada es inapropiada en muchas situaciones. Supóngase que María tiene el rol de *director*, concede el rol de *cajero* a Martín y luego se le retira el rol de *directora* a María (quizás porque haya dejado la empresa); Martín sigue trabajando para la empresa y debería conservar su rol de *cajero*.

Para abordar esta situación SQL:1999 permite que los roles, en vez de los usuarios, concedan privilegios. Supóngase que la concesión del rol de *cajero* (u otros privilegios) a Martín se realiza con el concedente configurado con el rol de *director* (usando la cláusula **granted by current_role** que se ha visto antes, con el rol actual definido como *director*), en lugar de que el concedente sea la usuaria María. Por tanto, la retirada de roles o privilegios (incluido el rol de *director*) a María no tiene como consecuencia la retirada de los privilegios que tenía el concedente configurado con el rol de *director*, aunque María fuera la usuaria que ejecutara la concesión; por tanto, Martín conserva el rol de *cajero*, incluso tras la retirada de los privilegios a María.

8.7.5 Autorización en vistas, funciones y procedimientos

En el Capítulo 3 se introdujo el concepto de *vista* como medio de ofrecer al usuario un modelo personalizado de la base de datos. Las vistas pueden ocultar datos que el usuario no necesite ver. La capacidad de las vistas de ocultar datos sirve tanto para simplificar el uso del sistema como para mejorar la seguridad. Las vistas simplifican el uso del sistema porque restringen la atención de los usuarios a los datos de interés. Aunque se puede negar a un usuario el acceso directo a una relación, se le puede permitir a ese usuario el acceso a parte de esa relación mediante una vista. Por tanto, una combinación de seguridad en el nivel relacional y de seguridad en el nivel de las vistas limita el acceso de los usuarios precisamente a los datos que necesitan.

En el ejemplo bancario, considérese un empleado que necesite conocer el nombre de todos los clientes que tienen concedido un préstamo en cada sucursal. Este empleado no está autorizado a ver información relativa a los préstamos concretos que pueda tener cada cliente. Por tanto, se debe denegar a ese empleado el acceso directo a la relación *préstamo*. Pero, si tiene que tener acceso a la información que necesita, se le debe conceder acceso a la vista *préstamo_cliente*, que sólo consta de los nombres de los clientes y de las sucursales en las que tienen concedido algún préstamo. Esta vista se puede definir en SQL de la manera siguiente:

create view *préstamo_cliente* **as**
(**select** *nombre_sucursal, nombre_cliente*
from *prestatario, préstamo*
where *prestatario.número_préstamo = préstamo.número_préstamo*)

Supóngase que el empleado formula la siguiente consulta de SQL:

select *
from *préstamo_cliente*

Evidentemente, el empleado está autorizado a ver el resultado de esta consulta. Sin embargo, cuando el procesador de consultas la traduce en una consulta a las relaciones reales de la base de datos, produce una consulta sobre *prestatario* y sobre *préstamo*. Por tanto, el sistema debe comprobar la autorización de la consulta del empleado antes de comenzar a procesarla.

El usuario que crea una vista no recibe necesariamente todos los privilegios sobre esa vista. Sólo recibe los privilegios que no le dan ninguna autorización adicional respecto de las que ya tenía. Por ejemplo, no se le puede dar la autorización **update** a un usuario sobre una vista si no tiene esa autorización sobre las relaciones usadas para crear esa vista. Si un usuario crea una vista sobre la que no se le puede conceder ninguna autorización, el sistema denegará la solicitud de creación de esa vista. En el ejemplo de la vista *préstamo_cliente*, el creador de la vista debe tener la autorización **read** tanto sobre la relación *prestatario* como sobre la relación *préstamo*.

El privilegio **execute** se puede conceder sobre una función o sobre un procedimiento y permite que el usuario los ejecute. De manera predeterminada, al igual que con las vistas, las funciones y los privilegios tienen todos los privilegios que tenía su creador. En efecto, la función o el procedimiento se ejecutan como si los invocara el usuario que los creó. El usuario actual de la sesión se configura como creador de la función o del procedimiento mientras se ejecutan.

Aunque este comportamiento resulta adecuado en muchas ocasiones, no siempre lo es. En SQL:2003, si la definición de la función tiene la cláusula adicional **sql security invoker**, se ejecuta con los privilegios del usuario que la invoca, en vez de hacerlo con los de su **definidor**. Esto permite la creación de bibliotecas de funciones que se pueden ejecutar bajo la misma autorización que tiene el usuario que las invoca.

8.7.6 Limitaciones de la autorización de SQL

Las normas actuales de SQL para la autorización tienen algunas deficiencias. Por ejemplo, supóngase que se desea que todos los estudiantes sean capaces de ver sus propias notas, pero no las del resto. La autorización debe estar en el nivel de las tuplas, lo cual no es posible según las normas para autorización de SQL.

Más aún, con el crecimiento de Web, los accesos a bases de datos provienen sobre todo de los servidores de aplicaciones Web. Puede que los usuarios finales no tengan identificadores de usuario individuales en la base de datos y, realmente, puede que haya sólo un identificador de usuario en la base de datos que corresponda a todos los usuarios del servidor de aplicaciones.

La tarea de autorización recae entonces sobre el servidor de aplicaciones; se soslaya todo el esquema de autorización de SQL. La ventaja es que la aplicación puede implementar autorizaciones más pormenorizadas, como las de las tuplas individuales. Éstos son los problemas:

- El código para comprobar la autorización se entremezcla con el resto del código de la aplicación.
- La implementación de la autorización mediante el código de las aplicaciones, en lugar de especificarla declarativamente en SQL, hace que sea difícil garantizar la ausencia de agujeros de seguridad. Debido a un descuido, puede que alguno de los programas de aplicación no compruebe la autorización, lo que permitiría el acceso de usuarios no autorizados a datos confidenciales. La verificación de que todos los programas de aplicación realizan todas las comprobaciones de autorización implica la lectura de todo el código del servidor de aplicaciones, una tarea formidable en sistemas de gran tamaño.

Algunos sistemas de bases de datos ofrecen mecanismos para las autorizaciones pormenorizadas. Por ejemplo, la **base de datos virtual privada** (Virtual Private Database, VPD) de Oracle permite que el administrador del sistema asocie funciones con relaciones; la función devuelve un predicado que se debe añadir a las consultas que utilicen la relación (se pueden definir funciones diferentes para las

relaciones que se estén actualizando). Por ejemplo, la función para la relación *cuenta* puede devolver un predicado como

número_cuenta **in**
(**select** *número_cuenta*
from *impositor*
where *impositor.nombre* = *contextosistema.id_usuario*())

Este predicado se añade a la cláusula **where** de cada consulta que utilice la relación *cuenta*. En consecuencia (suponiendo que el nombre del impositor coincida con el id_usuario de la base de datos), cada usuario de la base de datos sólo puede ver las tuplas correspondientes a las cuentas de las que es titular. Por tanto, VPD ofrece autorización en el nivel de cada una de las filas de la relación y, en consecuencia, se dice que es un mecanismo de *autorización en el nivel de las filas*.

Hay que ser consciente de que añadir ese predicado puede cambiar el significado de una consulta de manera importante. Por ejemplo, si un usuario escribe una consulta para averiguar el saldo medio de las cuentas, acabará teniendo el promedio de los saldos de sus propias cuentas.

Para tratar las aplicaciones Web en las que la aplicación se conecta con la base de datos usando un solo identificador de usuario, Oracle también permite que las aplicaciones definan el identificador de usuario de la conexión. Véanse en las notas bibliográficas las indicaciones de más información sobre la VPD de Oracle.

Puede que las diferentes provisiones que pueden hacer los sistemas de bases de datos para la autorización sigan sin ofrecer protección suficiente para los datos más delicados. En esos casos, los datos se pueden almacenar de manera cifrada. El cifrado se describe con más detalle en el Apartado 8.8.1.

8.7.7 Trazas de auditoría

Muchas aplicaciones de bases de datos seguras exigen que se mantenga una **traza de auditoría**. Una traza de auditoría es un registro histórico de todas las modificaciones (inserciones, borrados o actualizaciones) de la base de datos, junto con información sobre el usuario que realizó la modificación y el momento en que se produjo.

La traza de auditoría mejora la seguridad de varias formas. Por ejemplo, si se averigua que el saldo de una cuenta es incorrecto, puede que el banco desee revisar todas las actualizaciones realizadas en esa cuenta para descubrir las actualizaciones incorrectas (o fraudulentas), así como las personas que las realizaron. El banco puede entonces usar la traza de auditoría para seguir la pista de todas las actualizaciones realizadas por esas personas con objeto de encontrar otras actualizaciones incorrectas o fraudulentas.

Es posible crear trazas de auditoría definiendo los disparadores adecuados sobre las actualizaciones de las relaciones (usando variables definidas por el sistema que identifican el nombre de usuario y la hora). Sin embargo, muchos sistemas de bases de datos proporcionan mecanismos predeterminados para crear trazas de auditoría que son más cómodos de usar. Los detalles de la creación de las trazas de auditoría varían de unos sistemas de bases de datos a otros y se deben consultar los manuales de los diferentes sistemas para conocer los detalles.

8.8 Seguridad de las aplicaciones

En la seguridad de los datos de las aplicaciones es necesario afrontar amenazas y problemas aparte de los que gestiona la autorización de SQL. Por ejemplo, hay que proteger los datos mientras se transmiten; puede que haga falta proteger los datos de los intrusos que logran superar la seguridad del sistema operativo; y puede que los datos tengan complejas restricciones de intimidad que superen las que pueden aplicar las bases de datos. Estos y otros problemas relacionados se abordan en este apartado.

8.8.1 Técnicas de cifrado

Existe un enorme número de técnicas para el cifrado de los datos. Puede que las técnicas de cifrado sencillas no proporcionen la seguridad adecuada, dado que para los usuarios no autorizados puede ser sencillo romper el código. Como ejemplo de una técnica de cifrado débil considérese la sustitución de cada carácter por el siguiente en el alfabeto. Por tanto,

Navacerrada

se transforma en

Ñbwbdfssbeb

Si un usuario no autorizado sólo lee "Ñbwbdfssbeb" probablemente no tenga la información suficiente para romper el código. Sin embargo, si el intruso ve gran número de nombres de sucursales cifrados, puede usar los datos estadísticos referentes a la frecuencia relativa de los caracteres para averiguar el tipo de sustitución realizado (por ejemplo, en español la *E* es la más frecuente, seguida por *A, O, L, S, N*, etc.).

Una buena técnica de cifrado posee las propiedades siguientes:

- Resulta relativamente sencillo para los usuarios autorizados cifrar y descifrar los datos.
- No depende de lo poco conocido que sea el algoritmo sino de un parámetro del algoritmo denominado *clave de cifrado*.
- Es extremadamente difícil para un intruso determinar la clave de cifrado.

Un enfoque, la *norma de cifrado de datos* (Data Encryption Standard, DES), publicado en 1977, realiza tanto la sustitución de los caracteres como su reordenación con base en la clave de cifrado. Para que este esquema funcione se debe dotar a los usuarios autorizados de la clave de cifrado mediante un mecanismo seguro. Este requisito es una debilidad importante, ya que el esquema no es más seguro que el mecanismo por el que se transmite la clave de cifrado. La norma DES se reafirmó en 1983, 1987 y, de nuevo, en 1993. No obstante, se reconoció en 1993 que la debilidad de DES había alcanzado un punto en el que hacía falta seleccionar una nueva norma denominada **norma de cifrado avanzado** (Advanced Encryption Standard, AES). En el año 2000, se seleccionó el **algoritmo Rijndael** (denominado así por sus inventores, V. Rijmen y J. Daemen) para que fuera la norma AES. Este algoritmo se eligió por su nivel significativamente más fuerte de seguridad y la facilidad relativa de su implementación en los sistemas informáticos actuales, así como en dispositivos como las tarjetas inteligentes. Al igual que la norma DES, el algoritmo Rijndael es un algoritmo de clave compartida (o de clave simétrica) en el que los usuarios autorizados comparten una clave.

El **cifrado de clave pública** es un esquema alternativo que evita parte de los problemas que se afrontan con el DES. Se basa en dos claves; una *clave pública* y otra *privada*. Cada usuario U_i tiene una clave pública E_i y una clave privada D_i. Todas las claves públicas están publicadas: cualquiera puede verlas. Cada clave privada sólo la conoce el usuario al que pertenece. Si el usuario U_1 desea guardar datos cifrados, los cifra usando la clave pública E_1. Descifrarlos exige la clave privada D_1.

Como la clave de cifrado de cada usuario es pública, se puede intercambiar información de manera segura usando este esquema. Si el usuario U_1 desea compartir los datos con U_2, los codifica usando E_2, la clave pública de U_2. Dado que sólo el usuario U_2 conoce la manera de descifrar los datos, la información se transmite de manera segura.

Para que el cifrado de clave pública funcione debe haber un esquema de cifrado que pueda hacerse público sin permitir fácilmente que se descubra el esquema de descifrado. En otros términos, dada la clave pública, debe resultar difícil deducir la clave privada. Existe un esquema de ese tipo y se basa en estas condiciones:

- Hay un algoritmo eficiente para comprobar si un número es primo.
- No se conoce ningún algoritmo eficiente para encontrar los factores primos de un número.

Para los fines de este esquema los datos se tratan como conjuntos de enteros. Se crea la clave pública calculando el producto de dos números primos grandes: P_1 y P_2. La clave privada consiste en el par (P_1, P_2). El algoritmo de descifrado no se puede usar con éxito si sólo se conoce el producto P_1P_2; necesita el valor de P_1 y de P_2. Dado que todo lo que se publica es el producto P_1P_2, los usuarios no autorizados necesitan poder factorizar P_1P_2 para robar los datos. Eligiendo P_1 y P_2 suficientemente grandes (por encima de 100 cifras) se puede hacer el coste de la factorización de P_1P_2 prohibitivamente elevado (del orden de años de tiempo de cálculo, incluso en las computadoras más rápidas).

En las notas bibliográficas se hace referencia a los detalles del cifrado de clave pública y a la justificación matemática de las propiedades de esta técnica.

Pese a que el cifrado de clave pública que usa el esquema descrito es seguro, también es costoso en cuanto a cálculo. Un esquema híbrido usado para proteger las comunicaciones es el siguiente: las claves DES se intercambian mediante un esquema de cifrado de clave pública y posteriormente se usa el cifrado DES para los datos transmitidos.

8.8.2 Soporte del cifrado en las bases de datos

Muchos sistemas de archivos y de bases de datos de hoy en día soportan el cifrado de los datos. Ese cifrado protege los datos de quien pueda tener acceso a los datos pero no sea capaz de tener acceso a la clave de descifrado. En el caso del cifrado del sistema de archivos, los datos que se cifran suelen ser archivos de gran tamaño y directorios que contienen información sobre los archivos.

En el contexto de las bases de datos, el cifrado se puede llevar a cabo en diferentes niveles. En el nivel inferior, los bloques de disco que contienen datos de la base de datos se pueden cifrar, usando una clave disponible para el software del sistema de bases de datos. Cuando se recupera un bloque del disco, primero se descifra y luego se usa de la manera habitual. Este cifrado en el nivel de los bloques del disco protege contra los atacantes que pueden tener acceso al contenido del disco pero no disponen de la clave de cifrado. También tiene la ventaja de necesitar sobrecargas de tiempo y de espacio relativamente bajas. Por ejemplo, si hay que proteger contra el robo de la propia computadora los datos de la base de datos de una computadora portátil, se puede usar este cifrado. La clave de descifrado debería facilitarla el usuario cada vez que se reiniciara el software de bases de datos. De manera parecida, quien tenga acceso a las cintas de copia de seguridad de una base de datos no tendrá acceso a los datos contenidos en esas copias de seguridad si no conoce la clave de descifrado.

En los sistemas de bases de datos compartidos, no se puede usar el cifrado de los bloques del disco para proteger los datos de otros usuarios privilegiados, como los administradores de la base de datos, que pueden formular consultas a la base de datos. Para proteger los datos contra este tipo de accesos, se debe aplicar el cifrado *antes* de que los datos lleguen a la base de datos. La aplicación debe cifrar los datos antes de enviarlos a la base de datos. Varios sistemas de bases de datos proporcionan APIs para el cifrado que ofrecen ese soporte para las columnas especificadas. Se puede usar una sola clave para todas las columnas cifradas y para todas las filas de una columna dada. El uso de una clave diferente para cada fila no resulta factible, ya que dificultaría mucho la tarea de gestión de las claves.

El almacenamiento seguro de las claves de cifrado es otro problema relacionado. Si se almacenan en forma de archivo del sistema operativo, quien pueda romper la seguridad del sistema operativo podrá tener acceso a las claves. Algunos sistemas operativos ofrecen *almacenamiento seguro*; es decir, sólo permiten que recupere cada clave la aplicación que la almacenó. La propia aplicación se puede identificar mediante un valor de asociación de su ejecutable, por lo que los atacantes que sustituyan la aplicación por una copia modificada no podrán tener acceso a la clave.

El cifrado de valores pequeños, como los identificadores o los nombres, se complica por la posibilidad de los ataques de diccionario, especialmente si la clave de cifrado está disponible públicamente. Por ejemplo, si se cifran los campos fecha-de-nacimiento, el atacante que intente descifrar un valor cifrado concreto e puede intentar cifrar todas las fechas de nacimiento posibles hasta que encuentre una cuyo valor cifrado coincida con e. Estos ataques se pueden disuadir añadiendo bits adicionales aleatorios al final del valor antes de su cifrado (y eliminándolos tras su descifrado). Estos bits adicionales (a veces denominados *bits de sal*) ofrecen buena protección contra los ataques de diccionario.

8.8.3 Autenticación

La autenticación hace referencia a la tarea de comprobar la identidad de las personas o del software que se conectan a la base de datos. La forma más sencilla consta de una contraseña secreta que se debe presentar cuando se abre una conexión con la base de datos.

La autenticación basada en contraseñas la usan mucho los sistemas operativos y las bases de datos. Sin embargo, el uso de contraseñas tiene algunos inconvenientes, especialmente en las redes. Si un espía es capaz de "oler" los datos que se envían por la red, puede ser capaz de averiguar la contraseña cuando se envíe por la red. Una vez que el husmeador tiene un usuario y una contraseña, se puede conectar a la base de datos fingiendo que es el usuario legítimo.

8.8.3.1 Sistemas de respuesta por desafío

Un esquema más seguro es el sistema de **respuesta por desafío**. El sistema de bases de datos envía una cadena de desafío al usuario. El usuario cifra la cadena de desafío usando una contraseña secreta como clave de cifrado y devuelve el resultado. El sistema de bases de datos puede verificar la autenticidad del usuario descifrando la cadena con la misma contraseña secreta, y comparando el resultado con la cadena de desafío original. Este esquema garantiza que las contraseñas no circulan por la red.

Los sistemas de clave pública se pueden usar para el cifrado en los sistemas de respuesta por desafío. El sistema de bases de datos cifra la cadena de desafío usando la clave pública del usuario y se la envía al usuario. Éste descifra la cadena con su clave privada y devuelve el resultado al sistema de bases de datos. El sistema de bases de datos comprueba entonces la respuesta. Este esquema tiene la ventaja añadida de no almacenar la contraseña secreta en la base de datos, donde podrían verla los administradores del sistema.

Guardar la clave privada del usuario en una computadora (aunque sea una computadora personal) tiene el riesgo de que, si se pone en peligro la seguridad de la computadora, se puede revelar la clave al atacante, que se podrá hacer pasar por el usuario. Las **tarjetas inteligentes** ofrecen una solución a este problema. En las tarjetas inteligentes la clave se puede guardar en un chip incorporado; el sistema operativo de la tarjeta inteligente garantiza que no se pueda leer nunca la clave, pero permite que se envíen datos a la tarjeta para cifrarlos o descifrarlos usando la clave privada[3].

8.8.3.2 Firmas digitales

Otra aplicación interesante del cifrado de clave pública está en las **firmas digitales** para comprobar la autenticidad de los datos; las firmas digitales desempeñan el rol electrónico de las firmas físicas en los documentos. La clave privada se usa para firmar los datos y los datos firmados se pueden hacer públicos. Cualquiera puede comprobarlos con la clave pública, pero nadie puede haber generado los datos firmados sin tener la clave privada. Por tanto, se pueden **autentificar** los datos; es decir, se puede comprobar que fueron creados realmente por la persona que afirma haberlos creado.

Además, las firmas digitales también sirven para garantizar el **no repudio**. Es decir, en el caso de que la persona que creó los datos afirmara posteriormente que no los creó (el equivalente electrónico de afirmar que no se ha firmado un talón) se puede probar que esa persona tiene que haber creado los datos (a menos que su clave privada haya caído en manos de otros).

8.8.3.3 Certificados digitales

En general, la autenticación es un proceso de doble sentido, en el que cada miembro del par de entidades que interactúa se autentifica a sí mismo frente al otro. Esta autenticación por parejas es necesaria aunque un cliente entre en contacto con un sitio Web, para evitar que los sitios malévolos se hagan pasar por sitios Web legales. Esa suplantación se puede llevar a cabo, por ejemplo, si se pone en peligro la seguridad de los enrutadores de la red y los datos se desvían al sitio malévolo.

Para que el usuario pueda estar seguro de que está interactuando con el sitio Web auténtico debe tener la clave pública de ese sitio. Esto plantea el problema de hacer que el usuario consiga la clave pública

3. Las tarjetas inteligentes también ofrecen otra funcionalidad, como es la posibilidad de almacenar dinero digitalmente y realizar pagos, lo cual no es relevante en este contexto.

(si se almacena en el sitio Web, el sitio malévolo puede proporcionar una clave diferente y el usuario no tendría manera de comprobar si la clave pública proporcionada es auténtica). La autenticación se puede conseguir mediante un sistema de **certificados digitales**, en el que las claves públicas están firmadas por una agencia de certificación, cuya clave pública es bien conocida. Por ejemplo, las claves públicas de las autoridades de certificación raíz se almacenan en los navegadores Web estándar. Los certificados emitidos por ellas se pueden comprobar mediante las claves públicas almacenadas.

Un sistema de dos niveles supondría una carga excesiva de creación de certificados para las autoridades de certificación raíz, por lo que se usa en su lugar un sistema de varios niveles, con una o más autoridades de certificación raíz y un árbol de autoridades de certificación por debajo de cada raíz. Cada autoridad (que no sea autoridad raíz) tiene un certificado digital emitido por su autoridad principal.

El certificado digital emitido por la autoridad de certificación A consiste en una clave pública C_A y un texto cifrado E que se puede descifrar mediante la pública C_A. El texto cifrado contiene el nombre de la parte a la que se le ha emitido el certificado y su clave pública C_c. En caso de que la autoridad de certificación A no sea autoridad de certificación raíz, el texto cifrado también contiene el certificado digital emitido a A por su autoridad de certificación principal; este certificado autentifica la propia clave C_A (ese certificado puede, a su vez, contener un certificado de otra autoridad de certificación principal, y así sucesivamente). Para comprobar un certificado, se descifra el texto cifrado E mediante la clave pública y, si A no es autoridad raíz, la clave pública C_A se comprueba de manera recursiva usando el certificado digital contenido en E; la recursividad termina cuando se llega a un certificado emitido por la autoridad raíz. La comprobación de los certificados establece la cadena mediante la cual se autentifica cada sitio concreto y proporciona el nombre y la clave pública autentificada de ese sitio.

Los certificados digitales se usan mucho para autentificar los sitios Web ante los usuarios, para evitar que sitios malévolos suplanten a otros sitios Web. En el protocolo HTTPS (la versión segura del protocolo HTTP) el sitio proporciona su certificado digital al navegador, que lo muestra entonces al usuario. Si el usuario acepta el certificado, el navegador usa la clave pública proporcionada para cifrar los datos. Los sitios malévolos pueden tener acceso al certificado, pero no a la clave privada y, por tanto, no podrá descifrar los datos enviados por el navegador. Sólo el sitio auténtico, que tiene la clave privada correspondiente, puede descifrar los datos enviados por el navegador. Hay que tener en cuenta que los costes del cifrado y descifrado con la clave pública y la privada son mucho más elevados que los de cifrado y descifrado mediante claves privadas simétricas. Para reducir los costes de cifrado, HTTPS crea realmente una clave simétrica de un solo uso tras la autenticación y la usa para cifrar los datos durante el resto de la sesión.

Los certificados digitales también se pueden usar para autentificar a los usuarios. El usuario debe remitir al sitio un certificado digital que contenga su clave pública; el sitio comprueba que el certificado haya sido firmado por una autoridad de confianza. Entonces se puede usar la clave pública del usuario en un sistema de respuesta por desafío para garantizar que ese usuario posee la clave privada correspondiente, lo que autentifica al usuario.

8.8.3.4 Autenticación centralizada

Cuando los usuarios tienen acceso a varios sitios Web, suele resultar molesto para ellos tener que autentificarse en cada sitio por separado, a menudo con contraseñas diferentes para cada sitio. Hay sistemas que permiten que el usuario se autentifique ante un servicio central de autenticación y los demás sitios Web pueden autentificar al usuario mediante ese sitio Web; así, se puede usar la misma contraseña para tener acceso a varios sitios.

Los sistemas de **inicio de sesión único**, además, permiten al usuario autentificarse una sola vez (generalmente introduciendo una contraseña) para que varias aplicaciones puedan comprobar la identidad del usuario mediante el servicio central de autenticación sin necesidad de que vuelva a autentificarse. Estos mecanismos de inicio de sesión único se usan desde hace mucho tiempo en los sistemas operativos distribuidos, como Kerberos, y se dispone de varias implementaciones para las aplicaciones Web. Véanse las notas bibliográficas para obtener más información.

Además de autentificar a los usuarios, los servicios centrales de autenticación pueden ofrecer otros servicios, como proporcionar a la aplicación información sobre los usuarios como nombre, correo electrónico y domicilio. Esto elimina la necesidad de introducir esta información por separado en cada

aplicación. Los sistemas de directorios como LDAP y el Directorio Activo (Active Directory), y los sistemas de autenticación como el servicio Passport de Microsoft, ofrecen mecanismos para autentificar a los usuarios y proporcionar información sobre ellos.

8.8.4 Protección de las aplicaciones

Hay muchas maneras de poner en peligro la seguridad de las aplicaciones, aunque el sistema de bases de datos en sí mismo sea seguro. A continuación se describen varios posibles agujeros de seguridad y la manera de protegerse de ellos.

En los ataques de **inyección de SQL** el atacante logra hacer que una aplicación ejecute una consulta de SQL creada por él. Estos ataques funcionan de la manera siguiente. Considérese el texto fuente del formulario mostrado en la Figura 8.2. Supóngase que el servlet correspondiente, mostrado en la Figura 8.6, crea una cadena de caracteres de consulta de SQL usando la expresión de Java siguiente:

"select saldo from cuenta where número_cuenta ='" + número + "'"

donde número es una variable que contiene la cadena de caracteres introducida por el usuario. Un atacante malévolo que utilice el formulario Web puede escribir una cadena de caracteres como "';<alguna instrucción de SQL >;", donde <alguna instrucción de SQL > denota cualquier instrucción de SQL que desee el atacante, en lugar de un número de cuenta válido. El servlet creará y enviará la siguiente cadena de caracteres

select *saldo* **from** *cuenta* **where** *número_cuenta* = ' '; <alguna instrucción de SQL >;

La comilla insertada por el atacante cierra la cadena de caracteres, el siguiente punto y coma termina la consulta y el texto siguiente, insertado por el atacante, se interpreta como si fuera una consulta de SQL. Por tanto, el usuario malévolo ha logrado insertar una instrucción arbitraria de SQL, que la aplicación ejecuta. La instrucción puede provocar daños significativos, ya que puede eludir todas las medidas de seguridad implementadas en el código de la aplicación.

Para evitar estos ataques es mejor usar instrucciones preparadas para ejecutar las consultas de SQL. Al definir los parámetros de las consultas preparadas, JDBC añade de manera automática caracteres de escape para que la comilla proporcionada por el usuario ya no pueda terminar la cadena de caracteres. De manera equivalente, se puede aplicar una función que añada caracteres de escape a las cadenas de caracteres que introducen los usuarios antes de concatenarlos con las consultas de SQL, en lugar de usar instrucciones preparadas.

Otro problema con el que deben enfrentarse los desarrolladores de aplicaciones es el almacenaje de las contraseñas en texto claro en el código de las aplicaciones. Por ejemplo, los programas como los guiones de JSP suelen contener contraseñas en texto claro. Si estos guiones se almacenan en directorios accesibles para los servidores Web, los usuarios externos pueden tener acceso al código fuente del guión y, por tanto, a la contraseña de la cuenta de la base de datos usada por la aplicación. Para evitar estos problemas muchos servidores de aplicaciones ofrecen mecanismos para guardar las contraseñas en forma cifrada; el servidor las descifra antes de pasárselas a la base de datos. Esta característica elimina la necesidad de almacenar las contraseñas como texto claro en los programas de aplicación.

Como medida adicional contra las contraseñas de bases de datos puestas en peligro, muchos sistemas de bases de datos permiten que se restrinja el acceso a la base de datos a un conjunto dado de direcciones de Internet. Los intentos de conectarse con la base de datos desde otras direcciones de Internet se rechazan.

8.8.5 Intimidad

En un mundo en el que se una cantidad creciente de datos personales está disponible en línea, la gente está cada vez más preocupada por la intimidad de sus datos. Por ejemplo, la mayor parte de la gente desea que sus datos médicos personales sigan siendo confidenciales y no se revelen en público. No obstante, los datos médicos deben ponerse a disposición de los médicos y de los técnicos médicos de emergencias que tratan al paciente. Muchos países disponen de legislación sobre la intimidad de esos datos, que definen el momento en que pueden revelarse y la persona a la que se pueden revelar. La

violación de la legislación sobre intimidad puede dar lugar en algunos países a sanciones penales. Las aplicaciones que tienen acceso a esos datos privados deben elaborarse con cuidado, teniendo en cuenta la legislación sobre intimidad.

Por otro lado, los datos privados agregados pueden desempeñar un papel importante en muchas tareas, como la detección de los efectos secundarios de las drogas o la detección de la propagación de una epidemia. La manera de poner esos datos a disposición de los investigadores que desempeñan esas labores sin poner en peligro la intimidad de las personas es un problema importante del mundo real. Como ejemplo, supóngase que un hospital oculta el nombre del paciente pero proporciona al investigador su fecha de nacimiento y su código postal (que pueden resultar útiles al investigador). Esos dos elementos de información simplemente pueden usarse en muchos casos para identificar de manera unívoca al paciente (usando información de una base de datos externa), poniendo en peligro su intimidad. En esta situación concreta, una solución sería dar el año de nacimiento, pero no la fecha completa, junto con el código postal, lo que puede ser suficiente para el investigador. Esto no proporcionaría información suficiente para identificar de manera unívoca a la mayor parte de las personas[4]. Como ejemplo adicional, los sitios Web suelen reunir datos personales como la dirección, el teléfono, la dirección de correo electrónico y la información sobre la tarjeta de crédito. Esta información puede ser necesaria para llevar a cabo una transacción como la compra de un producto de una tienda. No obstante, puede que el cliente no desee que esa información se ponga a disposición de otras organizaciones, o puede que desee que parte de esa información (como el número de la tarjeta de crédito) se borre pasado cierto tiempo como manera de impedir que caiga en manos no autorizadas en caso de que se produzca un fallo de seguridad. Muchos sitios Web permiten que los clientes especifiquen sus preferencias de intimidad, y luego deben asegurarse de que esas preferencias se respeten.

8.9 Resumen

- La mayor parte de los usuarios interactúa con las bases de datos mediante formularios e interfaces gráficas de usuario, y hay muchas herramientas para simplificar la creación de esas interfaces. Los generadores de informes son herramientas que ayudan a crear informes legibles para las personas a partir del contenido de la base de datos.
- Los navegadores Web se han constituido en la interfaz de usuario con las bases de datos más usada. HTML ofrece la posibilidad de definir interfaces que combinan hipervínculos con formularios. Los navegadores Web se comunican con los servidores Web mediante el protocolo HTTP. Los servidores Web pueden pasar solicitudes a los programas de aplicación y devolver el resultado al navegador.
- Hay varios lenguajes de guiones del lado del cliente—Javascript es el más usado—que proporcionan una mayor interacción con el usuario en el extremo del navegador.
- Los servidores Web ejecutan programas de aplicación para implementar la funcionalidad deseada. Las servlets son un mecanismo muy usado para escribir programas de aplicación que se ejecutan como parte del proceso del servidor Web, para reducir las sobrecargas. También hay muchos lenguajes de guiones del lado del servidor que interpreta el servidor Web y proporcionan la funcionalidad de los programas de aplicación como parte del servidor Web.
- Los disparadores definen las acciones que se deben ejecutar automáticamente cuando se producen ciertos eventos. Los disparadores tienen muchas aplicaciones, como la implementación de reglas de negocio, la auditoría del registro histórico e incluso realizar acciones fuera del sistema de bases de datos. Aunque los disparadores sólo se han añadido recientemente a la norma de SQL como parte de SQL:1999 la mayor parte de los sistemas de bases de datos los implementan desde hace tiempo.

4. Para la gente extremadamente anciana, que es relativamente escasa, incluso el año de nacimiento junto el código postal puede resultar suficiente para identificar a una persona concreta, por lo que se puede proporcionar un rango de valores, como puede ser ochenta o más años, en lugar de la edad real de las personas mayores de ochenta años.

- Se puede autorizar a los usuarios a los que se les haya concedido algún tipo de autorización a transmitírsela a otros usuarios. No obstante, hay que tomar precauciones respecto a la manera en que se pueden transmitir las autorizaciones de unos usuarios a otros si se desea garantizar que esas autorizaciones se puedan retirar en el futuro.
- Los roles facilitan la asignación de conjuntos de privilegios a los usuarios de acuerdo con el rol que esos usuarios desempeñen en la organización.
- Los mecanismos de autorización de SQL son poco detallados y de valor limitado para las aplicaciones que trabajan con gran número de usuarios. Se han desarrollado extensiones para proporcionar control de acceso en el nivel de las filas y para trabajar con gran número de usuarios de aplicaciones, pero todavía no están normalizadas.
- El cifrado desempeña un papel fundamental en la protección de la información y en la autenticación de los usuarios y de los sitios Web. Los sistemas de respuesta por desafío se suelen usar para autentificar a los usuarios. Los certificados digitales desempeñan un papel fundamental en la autenticación de los sitios Web.
- Los desarrolladores de aplicaciones deben prestar mucha atención a la seguridad, para evitar los ataques de inyección de SQL y otros ataques de usuarios malévolos.
- La protección de la intimidad de los datos es una importante labor para las aplicaciones de bases de datos. Muchos países tienen disposiciones legales sobre el mantenimiento de la intimidad de determinadas clases de datos, como los datos médicos.

Términos de repaso

- Formularios.
- Interfaces gráficas de usuario.
- Generadores de informes.
- Interfaces Web con las bases de datos.
- Lenguaje de marcas de hipertexto (HyperTextMarkup Language, HTML).
- Hipervínculos.
- Localizador uniforme de recursos (Uniform resource locator, URL).
- Secuencias de comandos del lado del cliente.
- Javascript.
- Modelo de objetos documento (Document Object Model, DOM).
- Applets.
- Lenguaje de guiones del lado del cliente.
- Servidores Web.
- Sesión.
- Protocolo de transferencia de hipertexto (HyperText Transfer Protocol, HTTP).
- Interfaz de pasarela común (Common Gateway Interface, CGI).
- Sin conexión.
- Cookie.
- Servlets.
- Sesiones de servlets.
- JSP.
- Sesiones de comandos del lado del servidor.
- Agrupación de conexiones.
- ASP.NET.
- Disparador.
- Modelo evento-condición-acción.
- Disparadores **before** y **after**.
- Variables y tablas de transición.
- Autorización.
- Privilegios.
- Privilegio de conceder privilegios
- Opción **grant**.
- Roles.
- Retirada de privilegios.
- Autorización sobre las vistas.
- Autorización **execute**.
- Privilegios **invoker**.
- Autorización en el nivel de las filas.
- Traza de auditoría.
- Cifrado.
- Cifrado de clave pública.
- Autenticación.

- Respuesta por desafío.
- Firmas digitales.
- Certificados digitales.
- Autenticación centralizada.
- Inicio de sesión único.
- Inyección de SQL.
- Intimidad.

Ejercicios prácticos

8.1 ¿Cuál es la razón principal de que los servlets den mejor rendimiento que los programas que usan la interfaz de pasarela común (common gateway interface, CGI), pese a que los programas de Java suelen ejecutarse más lentamente que los programas de C o de C++?

8.2 Indíquense algunas de las ventajas y de los inconvenientes de los protocolos sin conexión frente a los protocolos que mantienen las conexiones.

8.3 Indíquense tres maneras en que se puede usar el almacenamiento en caché para acelerar el rendimiento de los servidores Web.

8.4 Considérese la vista *sucursal_cliente* definida como sigue:

```
create view sucursal_cliente as
    select nombre_sucursal, nombre_clienten
    from impositor, cuenta
    where impositor.número_cuenta = cuenta.número_cuenta
```

Supóngase que la vista está *materializada*; es decir, que la vista se calcula y se almacena. Escríbanse disparadores para *mantener* la vista, es decir, mantenerla actualizada según las inserciones y los borrados en *impositor* o en *cuenta*. No hay que preocuparse de las actualizaciones.

8.5 Escríbase un disparador de SQL para realizar la acción siguiente: cuando se borre una cuenta, comprobar para cada titular de la cuenta si tiene otras cuentas y, si no, borrarlo de la relación *impositor*.

8.6 Supóngase que alguien suplanta a una empresa y obtiene un certificado de una autoridad emisora de certificados. ¿Cuál es el efecto sobre las cosas (como los pedidos de compras o los programas) certificadas por la empresa suplantada y sobre las cosas certificadas por otras empresas?

8.7 Quizás los elementos de datos más importantes de cualquier sistema de bases de datos sean las contraseñas que controlan el acceso a la base de datos. Propóngase un esquema para guardar de manera segura las contraseñas. Es necesario asegurarse de que el esquema permita al sistema comprobar las contraseñas proporcionadas por los usuarios que intenten iniciar una sesión en el mismo.

Ejercicios

8.8 Escríbase un servlet y el código HTML asociado para la aplicación siguiente, muy sencilla: se permite que los usuarios remitan un formulario que contiene un valor, por ejemplo, n, y deben obtener una respuesta que contenga n símbolos "*".

8.9 Escríbase un servlet y el código HTML asociado para la aplicación sencilla siguiente: se permite que los usuarios remitan un formulario que contiene un número, por ejemplo, n, y deben obtener una respuesta que indique el número de veces que se ha remitido anteriormente el valor n. El número de veces que se ha remitido anteriormente el valor se debe almacenar en una base de datos.

8.10 Escríbase un servlet que autentifique a los usuarios (de acuerdo con los nombres de usuario y las contraseñas almacenadas en una relación de una base de datos) y defina una variable de sesión denominada *idusuario* tras la autenticación.

8.11 ¿Qué son los ataques de inyección de SQL? Explíquese cómo funcionan y las precauciones que se deben adoptar para evitarlos.

8.12 Escríbase pseudocódigo para gestionar un grupo de conexiones. El pseudocódigo debe incluir una función que cree el grupo (proporcionando una cadena de caracteres de conexión a la base de datos, el nombre de usuario de la base de datos y la contraseña como parámetros), una función que solicite las conexiones al grupo, una función que libere las conexiones para el grupo y una función que cierre el grupo de conexiones.

8.13 Supóngase que hay dos relaciones r y s, tales que la clave externa B de r hace referencia a la clave primaria A de s. Descríbase la manera en que puede usarse el mecanismo de los disparadores para implementar la opción **on delete cascade** cuando se borra una tupla de s.

8.14 La ejecución de un disparador puede hacer que se dispare otra acción. La mayor parte de los sistemas de bases de datos imponen un límite sobre la profundidad máxima de anidamiento. Explíquese el motivo de la imposición de ese límite.

8.15 Explíquese el motivo de que, cuando un director, por ejemplo, María, concede una autorización, esa autorización deba concederla el rol de director, en vez de la usuaria María.

8.16 Supóngase que el usuario A, que tiene todas las autorizaciones sobre la relación r, concede a **public** la autorización de selección sobre la relación r con la opción grant. Supóngase que el usuario B concede luego a A la autorización de selección sobre r. ¿Provoca esto un ciclo en el grafo de autorización? Explíquese el motivo.

8.17 Confecciónese una lista de los problemas de seguridad de los bancos. Para cada elemento de la lista, indíquese si ese problema afecta a la seguridad física, a la personal, a la del sistema operativo o a la de las bases de datos.

8.18 Los sistemas de bases de datos que guardan cada relación en un archivo diferente del sistema operativo pueden usar los esquemas de seguridad y de autorización del sistema operativo en lugar de definir un esquema especial propio. Indíquense una ventaja y un inconveniente de ese enfoque.

8.19 El mecanismo VPD de Oracle implementa la seguridad en el nivel de las filas añadiendo predicados a la cláusula where de cada consulta. Dese un ejemplo de predicado que pueda usarse para implementar la seguridad en el nivel de las filas y tres consultas con las propiedades siguientes:

- **a.** En el primer caso, la consulta con el predicado añadido da el mismo resultado que la consulta original.
- **b.** En el segundo caso, la consulta con el predicado añadido da un resultado que es siempre un subconjunto del resultado de la consulta original.
- **c.** En el tercer caso, la consulta con el predicado añadido da respuestas incorrectas.

8.20 Indíquense dos ventajas de cifrar los datos guardados en una base de datos.

8.21 Supóngase que se desea crear una traza de auditoría de las modificaciones de la relación *cuenta*.

- **a.** Defínanse los disparadores para crear una traza de auditoría y registrar la información en una relación denominada, por ejemplo, *traza_cuenta*. La información registrada debe incluir el identificador de usuario (supóngase que la función *id_usuario* () proporciona esa información) y una marca de tiempo, además de los valores antiguos y nuevos. También hay que proporcionar el esquema de la relación *traza_cuenta*.
- **b.** ¿Puede la implementación anterior garantizar que las actualizaciones realizadas por administradores de la bases de datos malévolos (o por alguien que consiga la contraseña del administrador) se hallen en la traza de auditoría? Explíquese la respuesta.

8.22 Los hackers pueden lograr hacer creer al usuario que su sitio Web es realmente un sitio Web en el que el usuario confía (como el de un banco o el de una tarjeta de crédito). Esto puede lograrse mediante mensajes engañosos de correo electrónico o, incluso, mediante la entrada en la infraestructura de la red y el redireccionamiento del tráfico de red destinado a, por ejemplo, mibanco.com, hacia el sitio de los hackers. Si se introduce el nombre de usuario y la contraseña en el sitio de los hackers, ese sitio puede registrarlo y usarlo posteriormente para entrar en la cuenta en el sitio ver-

dadero. Cuando se usa una URL como https://mibanco.com, se usa el protocolo HTTPS para evitar esos ataques. Explíquese la manera en que el protocolo puede usar los certificados digitales para comprobar la autenticidad del sitio.

8.23 Explíquese lo que es el sistema de autenticación de respuesta por desafío. ¿Por qué es más seguro que los sistemas tradicionales basados en las contraseñas?

Sugerencias de proyectos

Cada uno de los proyectos que aparecen a continuación es un proyecto de grandes dimensiones, que puede ser un proyecto de un semestre de duración realizado por un grupo de estudiantes. La dificultad de cada proyecto puede ajustarse fácilmente añadiendo o eliminando características.

Proyecto 8.1 Considérese el esquema E-R del Ejercicio práctico 6.4 (Capítulo 6), que representa la información de los equipos de una liga. Diséñese e impleméntese un sistema basado en Web para introducir, actualizar y examinar los datos.

Proyecto 8.2 Diséñese e impleméntese un sistema de carro de la compra que permita a los compradores reunir artículos en un carro de la compra (se puede decidir la información que se proporciona de cada artículo) y comprarlos todos juntos. Se puede extender y usar el esquema E-R del Ejercicio 6.21 del Capítulo 6. Conviene comprobar la disponibilidad del artículo y tratar los artículos no disponibles del modo que se considere adecuado.

Proyecto 8.3 Diséñese e impleméntese un sistema basado en Web para registrar la información sobre las matrículas y las notas de los estudiantes para los cursos de una universidad.

Proyecto 8.4 Diséñese e impleméntese un sistema que permita el registro de la información de rendimiento de las asignaturas (concretamente, las notas otorgadas a cada estudiante en cada trabajo o examen de cada asignatura, y el cálculo de una suma (ponderada) de las notas para obtener las notas totales de la asignatura). El número de trabajos o exámenes no debe estar predefinido; es decir, se pueden añadir más trabajos o exámenes en cualquier momento. El sistema también debe soportar la clasificación, permitiendo que se especifiquen separadores para varias notas.

Puede que también se desee integrarlo en el sistema de matrícula de los estudiantes del Proyecto 8.3 (que quizás implemente otro equipo de proyecto).

Proyecto 8.5 Diséñese e impleméntese un sistema basado en Web para la reserva de aulas de una universidad. Se debe soportar la reserva periódica (días y horas fijas cada semana para un semestre completo). También debe soportarse la cancelación de clases concretas de una reserva periódica.

Puede que también se desee integrarlo con el sistema de matrícula de estudiantes del Proyecto 8.3 (que quizás implemente otro equipo de proyecto) de modo que las aulas puedan reservarse para asignaturas y las cancelaciones de una clase o las reservas de clases adicionales puedan anotarse en una sola interfaz, se reflejen en la reserva de aulas y se comuniquen a los estudiantes por correo electrónico.

Proyecto 8.6 Diséñese e impleméntese un sistema para gestionar exámenes objetivos (con varias respuestas posibles) en línea. Se debe soportar el aporte distribuido de preguntas (por los profesores ayudantes, por ejemplo), la edición de las preguntas por quien esté a cargo de la asignatura y la creación de exámenes a partir del conjunto de preguntas disponible. También se deben poder suministrar los exámenes en línea, bien a una hora fija para todos los estudiantes o a cualquier hora pero con un límite de tiempo desde el comienzo hasta el final (se pueden soportar las dos opciones o sólo una de ellas) y dar a los estudiantes información sobre su puntuación al final del tiempo concedido.

Proyecto 8.7 Diséñese e impleméntese un sistema para gestionar el servicio de correo electrónico de los clientes. El correo entrante va a un buzón común. Hay una serie de agentes de atención al cliente que responden el correo electrónico. Si el mensaje es parte de una serie de respuestas (que se siguen mediante el campo responder a del correo electrónico) es preferible que responda el mensaje el mismo agente que lo haya respondido anteriormente. El sistema debe realizar un seguimiento de

todo el correo entrante y de las respuestas, de modo que los agentes puedan ver el historial de preguntas de cada cliente antes de responder a los mensajes.

Proyecto 8.8 Diséñese e impleméntese un mercado electrónico sencillo en el que los artículos puedan clasificarse para su compra o venta en varias categorías (que deben formar una jerarquía). Puede que también se deseen soportar los servicios de alerta, en los que los usuarios pueden registrar su interés por los artículos de una categoría concreta, quizás con otras restricciones añadidas, sin anunciar su interés de manera pública; después, cuando alguno de esos artículos se ofrece a la venta, se les notifica.

Proyecto 8.9 Diséñese e impleméntese un sistema de grupos de noticias basado en Web. Los usuarios deben poder suscribirse a los grupos de noticias y explorar los artículos de esos grupos. El sistema hace un seguimiento de los artículos que el usuario ha leído, de modo que no vuelvan a mostrarse. También hay que permitir la búsqueda de artículos antiguos.

Es posible que también se desee ofrecer un servicio de calificación de los artículos, de modo que los artículos con puntuación elevada se destaquen, lo que permite al lector ocupado saltarse los artículos con baja puntuación.

Proyecto 8.10 Diséñese e impleméntese un sistema basado en Web para gestionar una clasificación deportiva. Se registra mucha gente y puede que se les dé alguna clasificación inicial (quizás con base en sus resultados anteriores). Cualquier usuario puede retar a un partido a cualquier otro y las clasificaciones se ajustan de acuerdo con los resultados.

Un sistema sencillo para ajustar las clasificaciones se limita a pasar al ganador por delante del perdedor en la clasificación, en caso de que el ganador estuviera por detrás. Se puede intentar inventar sistemas de ajustes de la clasificación más complicados.

Proyecto 8.11 Diséñese e impleméntese un servicio de listados de publicaciones. El servicio debe permitir la introducción de información sobre las publicaciones, como pueden ser título, autores, año en que apareció la publicación, páginas, etc. Los autores deben ser una entidad separada con atributos como nombre, institución, departamento, correo electrónico, dirección y página Web.

La aplicación debe soportar varias vistas de los mismos datos. Por ejemplo, se deben facilitar todas las publicaciones de un autor dado (ordenadas por año, por ejemplo), o todas las publicaciones de los autores de una institución o de un departamento dados. También se debe soportar la búsqueda por palabras clave, tanto en la base de datos global como en cada una de las vistas.

Proyecto 8.12 Una tarea frecuente en cualquier organización es la recogida de información estructurada de un grupo de personas. Por ejemplo, puede que un director necesite pedir a los empleados que introduzcan sus planes de vacaciones, un profesor puede que desee obtener la respuesta de los estudiantes a un tema concreto, el estudiante que organiza un evento puede que desee permitir que otros estudiantes se matriculen en él o alguien puede desear organizar una votación en línea sobre un tema concreto.

Créese un sistema que permita a los usuarios crear con facilidad eventos de reunión de información. Al crear un evento, su creador debe definir quién tiene derecho a participar; para ello, el sistema debe conservar la información de los usuarios y permitir predicados que definan subconjuntos de usuarios. El creador del evento debe poder especificar conjuntos de datos (con tipos, valores predeterminados y controles de validación) que tendrán que proporcionar los usuarios. El evento debe tener una fecha límite asociada y la posibilidad de enviar recordatorios a los usuarios que todavía no hayan remitido su información. Se puede dar al creador del evento la opción de hacer que se cumpla la fecha límite de manera automática con base en una fecha u hora especificada o que decida iniciar una sesión y declarar que se ha superado la fecha límite. Se deben generar estadísticas sobre los datos enviados—para ello, se permitirá al creador del evento que cree resúmenes sencillos de la información aportada. El creador del evento puede elegir hacer públicos parte de los resúmenes, visibles a todos los usuarios, de manera continua (por ejemplo, el número de personas que ha respondido) o una vez superada la fecha límite (por ejemplo, la puntuación promedio obtenida).

Proyecto 8.13 Créese una biblioteca de funciones para simplificar la creación de interfaces Web. Hay que implementar, como mínimo, las funciones siguientes: una función que muestre el conjunto de resultados JDBC (en formato tabular), funciones que creen diferentes tipos de datos de texto y numéricos (con criterios de validación como el tipo de entrada y un rango opcional, aplicado en el cliente mediante el código Javascript correspondiente), funciones que introduzcan los valores de la fecha y de la hora (con valores predeterminados) y funciones que creen elementos de menú de acuerdo con un conjunto de resultados. Para obtener más nota se debe permitir al usuario configurar parámetros de estilo, como los colores y las fuentes, y proporcionar soporte a la paginación en las tablas (se pueden usar parámetros ocultos para los formularios para especificar la página que se debe mostrar). Créese una aplicación de bases de datos de ejemplo para ilustrar el uso de estas funciones.

Proyecto 8.14 Diséñese e impleméntese un sistema de agenda multiusuario basado en Web. El sistema debe realizar un seguimiento de las citas de cada persona, con eventos de ocurrencia múltiple, como las reuniones semanales, eventos compartidos (en los que las actualizaciones realizadas por el creador del evento se reflejan en las agendas de todos los que comparten ese evento). Proporciónese la notificación de los eventos por correo electrónico. Para obtener más nota se debe implementar un servicio Web que pueda usar un programa de recordatorios que se ejecute en la máquina cliente.

Notas bibliográficas

La información sobre los servlets, incluidos los tutoriales, las especificaciones de las normas y el software, está disponible en java.sun.com/products/servlet. La información sobre JSP está disponible en java.sun.com/products/jsp. La información sobre las bibliotecas de etiquetas de JSP también se puede encontrar en ese URL. La información sobre la estructura .NET y sobre el desarrollo de aplicaciones Web mediante ASP.NET se puede hallar en msdn.microsoft.com.

Las propuestas originales de asertos y disparadores para SQL se estudian en Astrahan et al. [1976], Chamberlin et al. [1976] y Chamberlin et al. [1981]. Melton y Simon [2001], Melton [2002] y Eisenberg y Melton [1999] ofrecen un tratamiento del nivel de los libros de texto de SQL:1999, incluido el tratamiento de los asertos y de los disparadores en SQL:1999.

Se puede encontrar más información sobre la base de datos privada virtual (Virtual Private Database, VPD) de Oracle, que proporciona, entre otras características, autorizaciones detalladas, se puede encontrar en www.oracle.com/technology/deploy/security/index.html. Las autorizaciones detalladas se examinan también en Rizvi et al. [2004].

Atreya et al. [2002] proporciona un tratamiento del nivel de los libros de texto de las firmas digitales, incluidos los certificados digitales X.509 y de la infraestructura de clave pública. La información sobre el sistema de inicio de sesión único Pubcookie se puede encontrar en www.pubcookie.org.

Herramientas

El desarrollo de aplicaciones Web necesita varias herramientas de software como los servidores de aplicaciones, los compiladores y los editores de lenguajes de programación como Java o C#, y otras herramientas opcionales como los servidores Web. A continuación se relacionan algunas de las herramientas mejor conocidas: el SDK de Java de Sun (java.sun.com), el sistema Tomcat de Apache (jakarta.apache.org), que soporta los servlets y JSP, el servidor Web Apache (apache.org), el servidor de aplicaciones JBoss (jboss.org), las herramientas ASP.NET de Microsoft (msdn.microsoft.com/asp.net/), WebSphere de IBM (www.software.ibm.com), Resin de Caucho (www.caucho.com), los productos Coldfusion y JRun de Allaire (www.allaire.com) y Zope (www.zope.org). Algunos, como Tomcat y el servidor Web de Apache, son gratuitos para cualquier tipo de uso, algunos son gratuitos para su uso no comercial o personal, mientras que otros son de pago. Véanse los respectivos sitios Web para obtener más información.

PARTE 3

Bases de datos orientadas a objetos y XML

Varias áreas de aplicaciones para los sistemas de bases de datos se hallan limitadas por las restricciones del modelo de datos relacional. En consecuencia, los investigadores han desarrollado varios modelos de datos basados en enfoques orientados a objetos para tratar con esos dominios de aplicaciones.

El modelo relacional orientado a objetos, que se describe en el Capítulo 9, combina características del modelo relacional y del modelo orientado a objetos. Este modelo proporciona el rico sistema de tipos de los lenguajes orientados a objetos combinado con las relaciones como base del almacenamiento de los datos. Aplica la herencia a las relaciones, no sólo a los tipos. El modelo de datos relacional orientado a objetos permite una migración fácil desde las bases de datos relacionales, lo que resulta atractivo para las diferentes marcas de bases de datos relacionales. En consecuencia, la norma SQL:1999 incluye varias características orientadas a objetos en su sistema de tipos, mientras que sigue utilizando el modelo relacional como modelo subyacente.

El término base de datos orientada a objetos se utiliza para describir los sistemas de bases de datos que soportan el acceso directo a los datos desde lenguajes de programación orientados a objetos, sin necesidad de lenguajes de consultas relacionales como interfaz de las bases de datos. El Capítulo 9 también proporciona una breve visión general de las bases de datos orientadas a objetos.

El lenguaje XML se diseñó inicialmente como un modo de añadir información de marcas a los documentos de texto, pero ha adquirido importancia debido a sus aplicaciones en el intercambio de datos. XML permite representar los datos mediante una estructura anidada y, además, ofrece una gran flexibilidad en su estructuración, lo que resulta importante para ciertos tipos de datos no tradicionales. El Capítulo 10 describe el lenguaje XML y a continuación presenta diferentes formas de expresar las consultas sobre datos representados en XML, incluido el lenguaje de consultas XQuery para XML, que está adquiriendo una gran aceptación.

CAPÍTULO 9

Bases de datos basadas en objetos

Las aplicaciones tradicionales de las bases de datos consisten en tareas de procesamiento de datos, como la gestión bancaria y de nóminas, con tipos de datos relativamente sencillos, que se adaptan bien al modelo relacional. A medida que los sistemas de bases de datos se fueron aplicando a un rango más amplio de aplicaciones, como el diseño asistido por computadora y los sistemas de información geográfica, las limitaciones impuestas por el modelo relacional se convirtieron en un obstáculo. La solución fue la introducción de bases de datos basadas en objetos, que permiten trabajar con tipos de datos complejos.

9.1 Visión general

El primer obstáculo al que se enfrentan los programadores que usan el modelo relacional de datos es el limitado sistema de tipos soportado por el modelo relacional. Los dominios de aplicación complejos necesitan tipos de datos del mismo nivel de complejidad, como las estructuras de registros anidados, los atributos multivalorados y la herencia, que los lenguajes de programación tradicionales soportan. La notación E-R y las notaciones E-R extendidas soportan, de hecho, estas características, pero hay que trasladarlas a tipos de datos de SQL más sencillos. El **modelo de datos relacional orientado a objetos** extiende el modelo de datos relacional ofreciendo un sistema de tipos más rico que incluye tipos de datos complejos y orientación a objetos. Hay que extender de manera acorde los lenguajes de consultas relacionales, en especial SQL, para que puedan trabajar con este sistema de tipos más rico. Estas extensiones intentan conservar los fundamentos relacionales—en especial, el acceso declarativo a los datos—mientras extienden la potencia de modelado. Los **sistemas de bases de datos relacionales basadas en objetos**, es decir, los sistemas de bases de datos basados en el modelo objeto-relación, ofrecen un medio de migración cómodo para los usuarios de las bases de datos relacionales que deseen usar características orientadas a objetos.

El segundo obstáculo es la dificultad de acceso a los datos de la base de datos desde los programas escritos en lenguajes de programación como C++ o Java. La mera extensión del sistema de tipos soportado por las bases de datos no resulta suficiente para resolver completamente este problema. Las diferencias entre el sistema de tipos de las bases de datos y el de los lenguajes de programación hace más complicados el almacenamiento y la recuperación de los datos, y se debe minimizar. Tener que expresar el acceso a las bases de datos mediante un lenguaje (SQL) que es diferente del lenguaje de programación también hace más difícil el trabajo del programador. Es deseable, para muchas aplicaciones, contar con estructuras o extensiones del lenguaje de programación que permitan el acceso directo a los datos de la base de datos, sin tener que pasar por un lenguaje intermedio como SQL.

El término **lenguajes de programación persistentes** hace referencia a las extensiones de los lenguajes de programación existentes que añaden persistencia y otras características de las bases de datos usando el sistema de tipos nativo del lenguaje de programación. El término **sistemas de bases de datos orientadas a objetos** se usa para hacer referencia a los sistemas de bases de datos que soportan sistemas de

tipos orientados a objetos y permiten el acceso directo a los datos desde los lenguajes de programación orientados a objetos usando el sistema de tipos nativo del lenguaje.

En este capítulo se explicará primero el motivo del desarrollo de los tipos de datos complejos. Luego se estudiarán los sistemas de bases de datos relacionales orientados a objetos; el tratamiento se basará en las extensiones relacionales orientadas a objetos añadidas a la versión SQL:1999 de la norma de SQL. La descripción se basa en la norma de SQL, concretamente, en el uso de características que se introdujeron en SQL:1999 y en SQL:2003. Téngase en cuenta que la mayor parte de los productos de bases de datos sólo soportan un subconjunto de las características de SQL aquí descritas. Se debe consultar el manual de usuario del sistema de bases de datos que se utilice para averiguar las características que soporta.

Luego se estudian brevemente los sistemas de bases de datos orientados a objetos que añaden el soporte de la persistencia a los lenguajes de programación orientados a objetos. Finalmente, se describen situaciones en las que el enfoque relacional orientado a objetos es mejor que el enfoque orientado a objetos, y viceversa, y se mencionan criterios para escoger entre los dos.

9.2 Tipos de datos complejos

Las aplicaciones de bases de datos tradicionales consisten en tareas de procesamiento de datos, tales como la banca y la gestión de nóminas. Dichas aplicaciones presentan conceptualmente tipos de datos simples. Los elementos de datos básicos son registros bastante pequeños y cuyos campos son atómicos, es decir, no contienen estructuras adicionales y en los que se cumple la primera forma normal (véase el Capítulo 7). Además, sólo hay unos pocos tipos de registros.

En los últimos años, ha crecido la demanda de formas de abordar tipos de datos más complejos. Considérense, por ejemplo, las direcciones. Mientras que una dirección completa se puede considerar como un elemento de datos atómico del tipo cadena de caracteres, esa forma de verlo esconde detalles como la calle, la población, la provincia, y el código postal, que pueden ser interesantes para las consultas. Por otra parte, si una dirección se representa dividiéndola en sus componentes (calle, población, provincia y código postal) la escritura de las consultas sería más complicada, pues tendrían que mencionar cada campo. Una alternativa mejor es permitir tipos de datos estructurados, que admiten el tipo *dirección* con las subpartes *calle, población, provincia* y *código_postal*.

Como ejemplo adicional, considérense los atributos multivalorados del modelo E-R. Esos atributos resultan naturales, por ejemplo, para la representación de números de teléfono, ya que las personas pueden tener más de un teléfono. La alternativa de la normalización mediante la creación de una nueva relación resulta costosa y artificial para este ejemplo.

Con sistemas de tipos complejos se pueden representar directamente conceptos del modelo E-R, como los atributos compuestos, los atributos multivalorados, la generalización y la especialización, sin necesidad de una compleja traducción al modelo relacional.

En el Capítulo 7 se definió la *primera forma normal* (1FN), (1FN) que exige que todos los atributos tengan *dominios atómicos*. Recuérdese que un dominio es *atómico* si se considera que los elementos del dominio son unidades indivisibles.

La suposición de la 1FN es natural en los ejemplos bancarios que se han considerado. No obstante, no todas las aplicaciones se modelan mejor mediante relaciones en la 1FN. Por ejemplo, en vez de considerar la base de datos como un conjunto de registros, los usuarios de ciertas aplicaciones la ven como un conjunto de objetos (o de entidades). Puede que cada objeto necesite varios registros para su representación. Una interfaz sencilla y fácil de usar necesita una correspondencia de uno a uno entre el concepto intuitivo de objeto del usuario y el concepto de elemento de datos de la base de datos.

Considérese, por ejemplo, una aplicación para una biblioteca y supóngase que se desea almacenar la información siguiente para cada libro:

- Título del libro
- Lista de autores
- Editor
- Conjunto de palabras clave

Es evidente que, si se define una relación para esta información varios dominios no son atómicos.

- **Autores**. Cada libro puede tener una lista de autores, que se pueden representar como array. Pese a todo, puede que se desee averiguar todos los libros de los que es autor Santos. Por tanto, lo que interesa es una subparte del elemento del dominio "autores".
- **Palabras clave**. Si se almacena un conjunto de palabras clave para cada libro, se espera poder recuperar todos los libros cuyas palabras clave incluyan uno o más términos dados. Por tanto, se considera el dominio del conjunto de palabras clave como no atómico.
- **Editor**. A diferencia de *palabras clave* y *autores*, *editor* no tiene un dominio que se evalúe en forma de conjunto. No obstante, se puede considerar que *editor* consta de los subcampos *nombre* y *sucursal*. Este punto de vista hace que el dominio de *editor* no sea atómico.

La Figura 9.1 muestra una relación de ejemplo, *libros*.

Por simplificar se da por supuesto que el título del libro lo identifica de manera unívoca[1]. Se puede representar, entonces, la misma información usando el esquema siguiente:

- *autores*(*título, autor, posición*)
- *palabras_clave*(*título, palabra_clave*)
- *libros4*(*título, nombre_editor, sucursal_editor*)

El esquema anterior satisface la 4NF. La Figura 9.2 muestra la representación normalizada de los datos de la Figura 9.1.

Aunque la base de datos de libros de ejemplo puede expresarse de manera adecuada sin necesidad de usar relaciones anidadas, su uso lleva a un modelo más fácil de comprender. El usuario típico o el programador de sistemas de recuperación de la información piensa en la base de datos en términos de libros que tienen conjuntos de autores, como los modelos de diseño que no se hallan en la 1FN. El diseño de la 4FN exige consultas que reúnan varias relaciones, mientras que los diseños que no se hallan en la 1FN hacen más fáciles muchos tipos de consultas.

Por otro lado, en otras situaciones puede resultar más conveniente usar una representación en la primera forma normal en lugar de conjuntos. Por ejemplo, considérese la relación *impositor* del ejemplo bancario. La relación es de varios a varios entre *clientes* y *cuentas*. Teóricamente, se podría almacenar un conjunto de cuentas con cada cliente, o un conjunto de clientes con cada cuenta, o ambas cosas. Si se almacenaran ambas cosas, se tendría redundancia de los datos (la relación de un cliente concreto con una cuenta dada se almacenaría dos veces).

La posibilidad de usar tipos de datos complejos como los conjuntos y los arrays puede resultar útil en muchas aplicaciones, pero se debe usar con cuidado.

9.3 Tipos estructurados y herencia en SQL

Antes de SQL:1999 el sistema de tipos de SQL consistía en un conjunto bastante sencillo de tipos predefinidos. SQL:1999 añadió un sistema de tipos extenso a SQL, lo que permite los tipos estructurados y la herencia de tipos.

1. Esta suposición no se cumple en el mundo real. Los libros se suelen identificar por un número de ISBN de diez cifras que identifica de manera unívoca cada libro publicado.

título	*array_autores*	*editor*	*conjunto_palabras_clave*
		(*nombre, sucursal*)	
Compiladores	[Gómez, Santos]	(McGraw-Hill, NuevaYork)	{análisis sintáctico, análisis}
Redes	[Santos, Escudero]	(Oxford, Londres)	{Internet, Web}

Figura 9.1 Relación de libros que no está en la 1FN, *libros*.

título	*autor*	*posición*
Compiladores	Gómez	1
Compiladores	Santos	2
Redes	Santos	1
Redes	Escudero	2

autores

título	*palabra_clave*
Compiladores	análisis sintáctico
Compiladores	análisis
Redes	Internet
Redes	Web

palabras_clave

título	*nombre_editor*	*sucursal_editor*
Compiladores	McGraw-Hill	Nueva York
Redes	Oxford	Londres

libros4

Figura 9.2 Versión en la 4FN de la relación *libros*.

9.3.1 Tipos estructurados

Los tipos estructurados permiten representar directamente los atributos compuestos de los diagramas E-R. Por ejemplo, se puede definir el siguiente tipo estructurado para representar el atributo compuesto *nombre* con los atributos componentes *nombredepila* y *apellidos*:

```
create type Nombre as
    (nombredepila varchar(20),
    apellidos varchar(20))
    final
```

De manera parecida, el tipo estructurado siguiente puede usarse para representar el atributo compuesto *dirección*:

```
create type Dirección as
    (calle varchar(20),
    ciudad varchar(20),
    códigopostal varchar(9))
    not final
```

En SQL estos tipos se denominan tipos **definidos por el usuario**. La definición anterior corresponde al diagrama E-R de la Figura 6.4. Las especificaciones **final** y **not final** están relacionadas con la subtipificación, que se describirá más adelante, en el Apartado 9.3.22.[2] Ahora se pueden usar esos tipos para crear atributos compuestos en las relaciones, con sólo declarar que un atributo es de uno de estos tipos. Por ejemplo, se puede crear la tabla *cliente* de la manera siguiente:

```
create table cliente (
    nombre Nombre,
    dirección Dirección,
    fechaDeNacimiento date)
```

2. La especificación **final** de *Nombre* indica que no se pueden crear subtipos de *nombre*, mientras que la especificación **not final** de *Dirección* indica que se pueden crear subtipos de *dirección*.

Se puede tener acceso a los componentes de los atributos compuestos usando la notación "punto"; por ejemplo, *nombre.nombredepila* devuelve el componente nombre de pila del atributo nombre. El acceso al atributo *nombre* devolvería un valor del tipo estructurado *Nombre*.

También se puede crear una tabla cuyas filas sean de un tipo definido por el usuario. Por ejemplo, se puede definir el tipo *TipoCliente* y crear la tabla *cliente* de la manera siguiente:

```
create type TipoCliente as (
        nombre Nombre,
        dirección Dirección,
        fechaDeNacimiento date)
        not final
create table cliente of TipoCliente
```

Una manera alternativa de definir los atributos compuestos en SQL es usar **tipos de fila** sin nombre. Por ejemplo, la relación que representa la información del cliente se podría haber creado usando tipos de fila de la manera siguiente:

```
create table cliente (
    nombre row (nombredepila varchar(20),
                apellidos varchar(20)),
    dirección row (calle varchar(20),
                   ciudad varchar(20),
                   códigopostal varchar(9)),
    fechaDeNacimiento date)
```

Esta definición es equivalente a la anterior definición de la tabla, salvo en que los atributos *nombre* y *dirección* tienen tipos sin nombre y las filas de la tabla también tienen un tipo sin nombre.

La consulta siguiente ilustra la manera de tener acceso a los atributos componentes de los atributos compuestos. La consulta busca el apellido y la ciudad de cada cliente.

```
select nombre.apellidos, dirección.ciudad
from cliente
```

Sobre los tipos estructurados se pueden definir **métodos**. Los métodos se declaran como parte de la definición de los tipos de los tipos estructurados:

```
create type TipoCliente as (
        nombre Nombre,
        dirección Dirección,
        fechaDeNacimiento date)
        not final
method edadAFecha(aFecha date)
        returns interval year
```

El cuerpo del método se crea por separado:

```
create instance method edadAFecha (aFecha date)
    returns interval year
    for TipoCliente
begin
    return aFecha − self.fechaDeNacimiento;
end
```

Téngase en cuenta que la cláusula **for** indica el tipo al que se aplica el método, mientras que la palabra clave **instance** indica que el método se ejecuta sobre un ejemplar del tipo *Cliente*. La variable **self** hace referencia al ejemplar de *Cliente* sobre el que se invoca el método. El cuerpo del método puede contener

instrucciones procedimentales, que ya se han visto en el Apartado 4.6. Los métodos pueden actualizar los atributos del ejemplar sobre el que se ejecutan.

Los métodos se pueden invocar sobre los ejemplares de los tipos. Si se hubiera creado la tabla *cliente* del tipo *TipoCliente*, se podría invocar el método *edadAFecha* () como se ilustra a continuación, para averiguar la edad de cada cliente.

```
select nombre.apellidos, edadAFecha(current_date)
from cliente
```

En SQL:1999 se usan **funciones constructoras** para crear valores de los tipos estructurados. Las funciones con el mismo nombre que un tipo estructurado son funciones constructoras de ese tipo estructurado. Por ejemplo, se puede declarar una función constructora para el tipo *Nombre* de esta manera:

```
create function Nombre (nombredepila varchar(20), apellidos varchar(20))
returns Nombre
begin
    set self.nombredepila = nombredepila;
    set self.apellidos = apellidos;
end
```

Se puede usar **new** *Nombre* ('Martín', 'Gómez') para crear un valor del tipo *Nombre*.

Se puede crear un valor de fila haciendo una relación de sus atributos entre paréntesis. Por ejemplo, si se declara el atributo *nombre* como tipo de fila con los componentes *nombredepila* y *apellidos*, se puede crear para él este valor:

```
('Ted', 'Codd')
```

sin necesidad de función constructora.

De manera predeterminada, cada tipo estructurado tiene una función constructora sin argumentos, que configura los atributos con sus valores predeterminados. Cualquier otra función constructora hay que crearla de manera explícita. Puede haber más de una función constructora para el mismo tipo estructurado; aunque tengan el mismo nombre, deben poder distinguirse por el número y tipo de sus argumentos.

La instrucción siguiente ilustra la manera de crear una nueva tupla de la relación *Cliente*. Se da por supuesto que se ha definido una función constructora para *Dirección*, igual que la función constructora que se definió para *Nombre*.

```
insert into Cliente
    values
        (new Nombre('Martín', 'Gómez'),
        new Dirección('Calle Mayor, 20', 'Madrid', '28045'),
        date '22-8-1960')
```

9.3.2 Herencia de tipos

Supóngase que se tiene la siguiente definición de tipo para las personas:

```
create type Persona
    (nombre varchar(20),
     dirección varchar(20))
```

Puede que se desee almacenar en la base de datos información adicional sobre las personas que son estudiantes y sobre las que son profesores. Dado que los estudiantes y los profesores también son personas, se puede usar la herencia para definir en SQL los tipos estudiante y profesor:

```
create type Estudiante
    under Persona
    (grado varchar(20),
     departmento varchar(20))
create type Profesor
    under Persona
    (sueldo integer,
     departmento varchar(20))
```

Tanto *Estudiante* como *Profesor* heredan los atributos de *Persona*—es decir, *nombre* y *dirección*. Se dice que *Estudiante* y *Profesor* son subtipos de *Persona*, y *Persona* es un supertipo de *Estudiante* y de *Profesor*.

Los métodos de los tipos estructurados se heredan por sus subtipos, igual que los atributos. Sin embargo, cada subtipo puede redefinir el efecto de los métodos volviendo a declararlos, usando **overriding method** en lugar de **method** en la declaración del método.

La norma de SQL también exige un campo adicional al final de la definición de los tipos, cuyo valor es **final** o **not final**. La palabra clave **final** indica que no se pueden crear subtipos a partir del tipo dado, mientras que **not final** indica que se pueden crear subtipos. Ahora, supóngase que se desea almacenar información sobre los profesores ayudantes, que son a la vez estudiantes y profesores, quizás incluso en departamentos diferentes. Esto se puede hacer si el sistema de tipos soporta **herencia múltiple**, por la que se pueden declarar los tipos como subtipos de varios tipos. Téngase en cuenta que la norma de SQL (hasta las versiones SQL:1999 y SQL:2003, al menos) no soporta la herencia múltiple, aunque puede que sí que la soporten versiones futuras.

Por ejemplo, si el sistema de tipos soporta la herencia múltiple, se puede definir el tipo para los profesores ayudantes de la manera siguiente:

```
create type ProfesorAyudante
    under Estudiante, Profesor
```

ProfesorAyudante heredará todos los atributos de *Estudiante* y de *Profesor*. No obstante, hay un problema, ya que los atributos *nombre*, *dirección* y *departamento* están presentes tanto en *Estudiante* como en *Profesor*.

Los atributos *nombre* y *dirección* se heredan realmente de una fuente común, *Persona*. Por tanto, no hay ningún conflicto causado por heredarlos de *Estudiante* y de *Profesor*. Sin embargo, el atributo *departamento* se define por separado en *Estudiante* y en *Profesor*. De hecho, cada profesor ayudante puede ser estudiante en un departamento y profesor en otro. Para evitar conflictos entre las dos apariciones de *departamento*, se pueden rebautizar usando una cláusula **as**, como en la definición del tipo *ProfesorAyudante*:

```
create type ProfesorAyudante
    under Estudiante with (departmento as departamento_estudiante),
          Profesor with (departmento as departamento_profesor)
```

Hay que tener en cuenta nuevamente que SQL sólo soporta la herencia simple—es decir, cada tipo sólo se puede heredar de un único tipo, la sintaxis usada es como la de los ejemplos anteriores. La herencia múltiple, como en el ejemplo de *ProfesorAyudante*, *no* está soportada en SQL.

En SQL, como en la mayor parte del resto de los lenguajes, el valor de un tipo estructurado debe tener exactamente un "tipo más concreto". Es decir, cada valor debe asociarse, al crearlo, con un tipo concreto, denominado su **tipo más concreto**. Mediante la herencia también se asocia con cada uno de los supertipos de su tipo más concreto. Por ejemplo, supóngase que una entidad es tanto del tipo *Persona* como del tipo *Estudiante*. Entonces, el tipo más concreto de la entidad es *Estudiante*, ya que *Estudiante* es un subtipo de *Persona*. Sin embargo, una entidad no puede ser de los tipos *Estudiante* y *Profesor*, a menos que tenga un tipo, como *ProfesorAyudante*, que sea subtipo de *Profesor* y de *Estudiante* (lo cual no es posible en SQL, ya que SQL no soporta la herencia múltiple).

9.4 Herencia de tablas

Las subtablas de SQL se corresponden con el concepto de especialización/generalización de E-R. Por ejemplo, supóngase que se define la tabla *personas* de la manera siguiente:

create table *personas* **of** *Persona*

A continuación se pueden definir las tablas *estudiantes* y *profesores* como **subtabla**s de *personas*, de la manera siguiente:

create table *estudiantes* **of** *Estudiante*
under *personas*
create table *profesores* **of** *Profesor*
under *personas*

Los tipos de las subtablas deben ser subtipos del tipo de la tabla madre. Por tanto, todos los atributos presentes en *personas* también están presentes en las subtablas.

Además, cuando se declaran *estudiantes* y *profesores* como subtablas de *personas*, todas las tuplas presentes en *estudiantes* o en *profesores* pasan a estar también presentes de manera implícita en *personas*. Por tanto, si una consulta usa la tabla *personas*, no sólo encuentra tuplas directamente insertadas en esa tabla, sino también tuplas insertadas en sus subtablas, es decir, *estudiantes* y *profesores*. No obstante, esa consulta sólo puede tener acceso a los atributos que están presentes en *personas*.

SQL permite hallar tuplas que se encuentran en *personas* pero no en sus subtablas usando en las consultas "**only** *personas*" en lugar de *personas*. La palabra clave **only** también puede usarse en las sentencias delete y update. Sin la palabra clave **only**, la instrucción delete aplicada a una supertabla, como *personas*, también borra las tuplas que se insertaron originalmente en las subtablas (como *estudiantes*); por ejemplo, la instrucción

delete from *personas* **where** *P*

borrará todas las tuplas de la tabla *personas*, así como de sus subtablas *estudiantes* y *profesores*, que satisfagan *P*. Si se añade la palabra clave **only** a la instrucción anterior, las tuplas que se insertaron en las subtablas no se ven afectadas, aunque satisfagan las condiciones de la cláusula **where**. Las consultas posteriores a la supertabla seguirán encontrando esas tuplas.

Teóricamente, la herencia múltiple es posible con las tablas, igual que con los tipos. Por ejemplo, se puede crear una tabla del tipo *ProfesorAyudante*:

create table *profesores_ayudantes*
of *ProfesorAyudante*
under *estudiantes*, *profesores*

Como consecuencia de la declaración, todas las tuplas presentes en la tabla *profesores_ayudantes* también se hallan presentes de manera implícita en las tablas *profesores* y *estudiantes* y, a su vez, en la tabla *personas*. Hay que tener en cuenta, no obstante, que SQL no soporta la herencia múltiple de tablas.

Existen varios requisitos de consistencia para las subtablas. Antes de definir las restricciones, es necesaria una definición: se dice que las tuplas de una subtabla se **corresponden** con las tuplas de la tabla madre si tienen el mismo valor para todos los atributos heredados. Por tanto, las tuplas correspondientes representan a la misma entidad.

Los requisitos de consistencia de las subtablas son:

1. Cada tupla de la supertabla puede corresponderse, como máximo, con una tupla de cada una de sus subtablas inmediatas.
2. SQL posee una restricción adicional que hace que todas las tuplas que se corresponden entre sí deben proceder de una tupla (insertada en una tabla).

Por ejemplo, sin la primera condición, se podrían tener dos tuplas de *estudiantes* (o de *profesores*) que correspondieran a la misma persona.

La segunda condición excluye que haya una tupla de *personas* correspondiente a una tupla de *estudiantes* y a una tupla de *profesores*, a menos que todas esas tuplas se hallen presentes de manera implícita porque se haya insertado una tupla en la tabla *profesores_ayudantes*, que es subtabla tanto de *profesores* como de *estudiantes*.

Dado que SQL no soporta la herencia múltiple, la segunda condición impide realmente que ninguna persona sea a la vez profesor y estudiante. Aunque se soportara la herencia múltiple, surgiría el mismo problema si estuviera ausente la subtabla *profesores_ayudantes*. Evidentemente, resultaría útil modelar una situación en la que alguien pudiera ser a la vez profesor y estudiante, aunque no se hallara presente la subtabla *profesores_ayudantes*. Por tanto, puede resultar útil eliminar la segunda restricción de consistencia. Hacerlo permitiría que cada objeto tuviera varios tipos, sin necesidad de que tuviera un tipo más concreto.

Por ejemplo, supóngase que se vuelve a tener el tipo *Persona*, con los subtipos *Estudiante* y *Profesor*, y la tabla correspondiente *personas*, con las subtablas *profesores* y *estudiantes*. Se puede tener una tupla de *profesores* y una tupla de *estudiantes* correspondientes a la misma tupla de *personas*. No hace falta tener el tipo *ProfesorAyudante*, que es subtipo de *Estudiante* y de *Profesor*. No hace falta crear el tipo *ProfesorAyudante* a menos que se desee almacenar atributos adicionales o redefinir los métodos de manera específica para las personas que sean a la vez estudiantes y profesores.

No obstante, hay que tener en cuenta que, por desgracia, SQL prohíbe las situaciones de este tipo debido al segundo requisito de consistencia. Ya que SQL tampoco soporta la herencia múltiple, no se puede usar la herencia para modelar una situación en la que alguien pueda ser a la vez estudiante y profesor. En consecuencia, las subtablas de SQL no se pueden usar para representar las especializaciones que se solapen de los modelos E-R.

Por supuesto, se pueden crear tablas diferentes para representar las especializaciones o generalizaciones que se solapen sin usar la herencia. El proceso ya se ha descrito anteriormente, en el Apartado 6.9.5. En el ejemplo anterior, se crearían las tablas *personas*, *estudiantes* y *profesores*, de las que las tablas *estudiantes* y *profesores* contendrían el atributo de clave primaria de *Persona* y otros atributos específicos de *Estudiante* y de *Profesor*, respectivamente. La tabla *personas* contendría la información sobre todas las personas, incluidos los estudiantes y los profesores. Luego habría que añadir las restricciones de integridad referencial correspondientes para garantizar que los estudiantes y los profesores también se hallen representados en la tabla *personas*.

En otras palabras, se puede crear una implementación mejorada del mecanismo de las subtablas mediante las características de SQL ya existentes, con algún esfuerzo adicional para la definición de la tabla, así como en el momento de las consultas para especificar las reuniones para el acceso a los atributos necesarios.

Para finalizar este apartado, hay que tener en cuenta que SQL define un nuevo privilegio denominado **under**, el cual es necesario para crear subtipos o subtablas bajo otro tipo o tabla. La razón de ser de este privilegio es parecida a la del privilegio **references**.

9.5 Tipos array y multiconjunto en SQL

SQL soporta dos tipos de conjuntos: arrays y multiconjuntos; los tipos array se añadieron en SQL:1999, mientras que los tipos multiconjunto se agregaron en SQL:2003. Recuérdese que un *multiconjunto* es un conjunto no ordenado, en el que cada elemento puede aparecer varias veces. Los multiconjuntos son como los conjuntos, salvo que los conjuntos permiten que cada elemento aparezca, como mucho, una vez.

Supóngase que se desea registrar información sobre libros, incluido un conjunto de palabras clave para cada libro. Supóngase también que se deseara almacenar el nombre de los autores de un libro en forma de array; a diferencia de los elementos de los multiconjuntos, los elementos de los arrays están ordenados, de modo que se puede distinguir el primer autor del segundo, etc. El ejemplo siguiente ilustra la manera en que se pueden definir en SQL estos atributos valorados como arrays y como multiconjuntos.

```
create type Editor as
      (nombre varchar(20),
       sucursal varchar(20))
create type Libro as
      (título varchar(20),
       array_autores varchar(20) array [10],
       fecha_publicación date,
       editor Editor,
       conjunto_palabras_clave varchar(20) multiset)
create table libros of Libro
```

La primera instrucción define el tipo denominado *Editor*, que tiene dos componentes: nombre y sucursal. La segunda instrucción define el tipo estructurado *Libro*, que contiene un *título*, un *array_autores*, que es un array de hasta diez nombres de autor, una fecha de publicación, un editor (del tipo *Editor*), y un multiconjunto de palabras clave. Finalmente, se crea la tabla *libros*, que contiene las tuplas del tipo *Libro*.

Téngase en cuenta que se ha usado un array, en lugar de un multiconjunto, para almacenar el nombre de los autores, ya que el orden de los autores suele tener cierta importancia, mientras que se considera que el orden de las palabras asociadas con el libro no es significativo.

En general, los atributos multivalorados de los esquemas E-R se pueden asignar en SQL a atributos valorados como multiconjuntos; si el orden es importante, se pueden usar los arrays de SQL en lugar de los multiconjuntos.

9.5.1 Creación y acceso a los valores de los conjuntos

En SQL:1999 se puede crear un array de valores de esta manera:

array['Silberschatz', 'Korth', 'Sudarshan']

De manera parecida, se puede crear un multiconjunto de palabras clave de la manera siguiente:

multiset['computadora', 'base de datos', 'SQL']

Por tanto, se puede crear una tupla del tipo definido por la relación *libros* como:

```
('Compiladores', array['Gómez', 'Santos'], new Editor('McGraw-Hill', 'Nueva York'),
                                                    multiset['análisis sintáctico', 'análisis'])
```

En este ejemplo se ha creado un valor para el atributo *Editor* mediante la invocación a una función *constructora* para *Editor* con los argumentos correspondientes. Téngase en cuenta que esta constructora para *Editor* se debe crear de manera explícita y no se halla presente de manera predeterminada; se puede declarar como la constructora para *Nombre*, que ya se ha visto en el Apartado 9.3.

Si se desea insertar la tupla anterior en la relación *libros*, se puede ejecutar la instrucción:

```
insert into libros
values
('Compiladores', array['Gómez', 'Santos'],
                 new Editor('McGraw-Hill', 'Nueva York'),
                 multiset['análisis sintáctico', 'análisis'])
```

Se puede tener acceso a los elementos del array o actualizarlos especificando el índice del array, por ejemplo, *array_autores* [1].

9.5.2 Consulta de los atributos valorados como conjuntos

Ahora se considerará la forma de manejar los atributos que se valoran como conjuntos. Las expresiones que se valoran como conjuntos pueden aparecer en cualquier parte en la que pueda aparecer el nombre

de una relación, como las cláusulas **from**, como ilustran los siguientes párrafos. Se usará la tabla *libros* que ya se había definido anteriormente.

Si se desea averiguar todos los libros que tienen las palabras "base de datos" entre sus palabras clave se puede usar la consulta siguiente:

```
select título
from libros
where 'base de datos' in (unnest(conjunto_palabras_clave))
```

Obsérvese que se ha usado **unnest**(*conjunto_palabras_clave*) en una posición en la que SQL sin las relaciones anidadas habría exigido una subexpresión **select-from-where**.

Si se sabe que un libro concreto tiene tres autores, se puede escribir:

```
select array_autores[1], array_autores[2], array_autores[3]
from libros
where título = 'Fundamentos de bases de datos'
```

Ahora supóngase que se desea una relación que contenga parejas de la forma "título, nombre_autor" para cada libro y para cada uno de sus autores. Se puede usar esta consulta:

```
select L.título, A.autores
from libros as L, unnest(L.array_autores) as A(autores)
```

Dado que el atributo *array_autores* de *libros* es un campo que se valora como conjunto, **unnest**(*L.array_autores*) puede usarse en una cláusula **from** en la que se espere una relación. Téngase en cuenta que la variable tupla *B* es visible para esta expresión, ya que se ha definido *anteriormente* en la cláusula **from**.

Al desanidar un array la consulta anterior pierde información sobre el orden de los elementos del array. Se puede usar la cláusula **unnest with ordinality** para obtener esta información, como ilustra la consulta siguiente. Esta consulta se puede usar para generar la relación *autores*, que se ha visto anteriormente, a partir de la relación *libros*.

```
select título, A.autores, A.posición
from libros as L,
        unnest(L.array_autores) with ordinality as A(autores, posición)
```

La cláusula **with ordinality** genera un atributo adicional que registra la posición del elemento en el array. Se puede usar una consulta parecida, pero sin la cláusula **with ordinality**, para generar la relación *palabras_clave*.

9.5.3 Anidamiento y desanidamiento

La transformación de una relación anidada en una forma con menos atributos de tipo relación (o sin ellos) se denomina **desanidamiento**. La relación *libros* tiene dos atributos, *array_autores* y *conjunto_palabras_clave*, que son conjuntos, y otros dos, *título* y *editor*, que no lo son. Supóngase que se desea convertir la relación en una sola relación plana, sin relaciones anidadas ni tipos estructurados como atributos. Se puede usar la siguiente consulta para llevar a cabo la tarea:

```
select título, A.autor, editor.nombre as nombre_editor, editor.sucursal
        as sucursal_editor, P.palabras_clave
from libros as L, unnest(L.array_autores) as A(autores),
        unnest (L.conjunto_palabras_clave) as P(palabras_clave)
```

La variable *L* de la cláusula **from** se declara para que tome valores de *libros*. La variable *A* se declara para que tome valores de los autores de *array_autores* para el libro *L* y *P* se declara para que tome valores de las palabras clave del *conjunto_palabras_clave* del libro *L*. La Figura 9.1 muestra un ejemplar de la relación *libros* y la Figura 9.3 muestra la relación, denominada *libros_plana*, que es resultado de la consulta

título	*autor*	*nombre_editor*	*sucursal_editor*	*conjunto_palabras_clave*
Compiladores	Gómez	McGraw-Hill	Nueva York	análisis sintáctico
Compiladores	Santos	McGraw-Hill	Nueva York	análisis sintáctico
Compiladores	Gómez	McGraw-Hill	Nueva York	análisis
Compiladores	Santos	McGraw-Hill	Nueva York	análisis
Redes	Santos	Oxford	Londres	Internet
Redes	Escudero	Oxford	Londres	Internet
Redes	Santos	Oxford	Londres	Web
Redes	Escudero	Oxford	Londres	Web

Figura 9.3 *libros_plana*: resultado del desanidamiento de los atributos *array_autores* y *conjunto_palabras_clave* de la relación *libros*.

anterior. Téngase en cuenta que la relación *libros_plana* se halla en 1FN, ya que todos sus atributos son atómicos.

El proceso inverso de transformar una relación en la 1FN en una relación anidada se denomina **anidamiento**. El anidamiento puede realizarse mediante una extensión de la agrupación en SQL. En el uso normal de la agrupación en SQL se crea (lógicamente) una relación multiconjunto temporal para cada grupo y se aplica una función de agregación a esa relación temporal para obtener un valor único (atómico). La función **collect** devuelve el multiconjunto de valores; en lugar de crear un solo valor se puede crear una relación anidada. Supóngase que se tiene la relación en 1FN *libros_plana*, tal y como se muestra en la Figura 9.3. La consulta siguiente anida la relación en el atributo *palabras_clave*:

select *título*, *autores*, *Editor*(*nombre_editor, sucursal_editor*) **as** *editor*,
collect(*palabras_clave*) **as** *conjunto_palabras_clave*
from *libros_plana*
group by *título*, *autor*, *editor*

El resultado de la consulta a la relación *libros_plana* de la Figura 9.3 aparece en la Figura 9.4.

Si se desea anidar también el atributo autores en un multiconjunto, se puede usar la consulta:

select *título*, **collect**(*autores*) **as** *conjunto_autores*,
Editor(*nombre_editor, sucursal_editor*) **as** *editor*,
collect(*palabra_clave*) **as** *conjunto_palabras_clave*
from *libros_plana*
group by *título, editor*

Otro enfoque de la creación de relaciones anidadas es usar subconsultas en la cláusula **select**. Una ventaja del enfoque de las subconsultas es que se puede usar de manera opcional en la subconsulta una cláusula **order by** para generar los resultados en el orden deseado, lo que puede aprovecharse para crear un array. La siguiente consulta ilustra este enfoque; las palabras clave **array** y **multiset** especifican que se van a crear un array y un multiconjunto (respectivamente) a partir del resultado de las subconsultas.

título	*autor*	*editor*	*conjunto_palabras_clave*
		(*nombre_editor*, *sucursal_editor*)	
Compiladores	Gómez	(McGraw-Hill, Nueva York)	{análisis sintáctico, análisis}
Compiladores	Santos	(McGraw-Hill, Nueva York)	{análisis sintáctico, análisis}
Redes	Santos	(Oxford, Londres)	{Internet, Web}
Redes	Escudero	(Oxford, Londres)	{Internet, Web}

Figura 9.4 Una versión parcialmente anidada de la relación *libros_plana*.

```
select título,
      array( select autores
        from autores as A
        where A.título = L.título
        order by A.posición) as array_autores,
      Editor(nombre_editor, sucursal_editor) as editor,
      multiset( select palabra_clave
        from palabras_clave as P
        where P.título = L.título) as conjunto_palabras_clave,
from libros4 as L
```

El sistema ejecuta las subconsultas anidadas de la cláusula **select** para cada tupla generada por las cláusulas **from** and *where* de la consulta externa. Obsérvese que el atributo *L.título* de la consulta externa se usa en las consultas anidadas para garantizar que sólo se generen los conjuntos correctos de autores y de palabras clave para cada título.

SQL:2003 ofrece gran variedad de operadores para multiconjuntos, incluida la funcion **set**(M), que calcula una versión libre duplicada del multiconjunto M, la operación agregada **intersection**, cuyo resultado es la intersección de todos los multiconjuntos de un grupo, la operación agregada **fusion**, que devuelve la unión de todos los multiconjuntos de un grupo, y el predicado **submultiset**, que comprueba si el multiconjunto está contenido en otro multiconjunto.

La norma de SQL no proporciona ningún medio para actualizar los atributos de los multiconjuntos, salvo asignarles valores nuevos. Por ejemplo, para borrar el valor v del atributo de multiconjunto A hay que definirlo como (A **except all multiset**[v]).

9.6 Identidad de los objetos y tipos de referencia en SQL

Los lenguajes orientados a objetos ofrecen la posibilidad de hacer referencia a objetos. Los atributos de un tipo dado pueden servir de referencia para los objetos de un tipo concreto. Por ejemplo, en SQL se puede definir el tipo *Departamento* con el campo *nombre* y el campo *director*, que es una referencia al tipo *Persona*, y la tabla *departamentos* del tipo *Departamento*, de la manera siguiente:

```
create type Departmento (
      nombre varchar(20),
      director ref(Persona) scope personas
)
create table departamentos of Departmento
```

En este caso, la referencia está restringida a las tuplas de la tabla *personas*. La restricción del **ámbito** (scope) de referencia a las tuplas de una tabla es obligatoria en SQL, y hace que las referencias se comporten como las claves externas.

Se puede omitir la declaración **scope** *personas* de la declaración de tipos y hacer, en su lugar, un añadido a la instrucción **create table**:

```
create table departamentos of Departmento
      (director with options scope personas)
```

La tabla a la que se hace referencia debe tener un atributo que guarde el identificador de cada tupla. Ese atributo, denominado **atributo autorreferencial** (self-referential attribute), se declara añadiendo una cláusula **ref is** a la instrucción **create table**:

```
create table personas of Persona
ref is id_personal system generated
```

En este caso, *id_personal* es el nombre de un atributo, no una palabra clave, y la instrucción **create table** especifica que la base de datos genera de manera automática el identificador.

Para inicializar el atributo de referencia hay que obtener el identificador de la tupla a la que se va a hacer referencia. Se puede conseguir el valor del identificador de la tupla mediante una consulta. Por tanto, para crear una tupla con el valor de referencia, primero se puede crear la tupla con una referencia nula y luego definir la referencia de manera independiente:

```
insert into departamentos
    values ('CS', null)
update departamentos
    set director = (select p.id_personal
                    from personal as p
                    where nombre = 'Martín')
where nombre = 'CS'
```

Una alternativa a los identificadores generados por el sistema es permitir que los usuarios generen los identificadores. El tipo del atributo autorreferencial debe especificarse como parte de la definición de tipos de la tabla a la que se hace referencia, y la definición de la tabla debe especificar que la referencia está **generada por el usuario** (**user generated**):

```
create type Persona
    (nombre varchar(20),
     dirección varchar(20))
    ref using varchar(20)
create table personas of Persona
    ref is id_personal user generated
```

Al insertar tuplas en *personas* hay que proporcionar el valor del identificador:

```
insert into personas (id_personal, nombre, dirección) values
('01284567', 'Martín', 'Avenida del Segura, 23')
```

Ninguna otra tupla de *personas*, de sus supertablas ni de sus subtablas puede tener el mismo identificador. Por tanto, se puede usar el valor del identificador al insertar tuplas en *departamentos*, sin necesidad de más consultas para recuperarlo:

```
insert into departamentos
    values ('CS', '01284567')
```

Incluso es posible usar el valor de una clave primaria ya existente como identificador, incluyendo la cláusula **ref from** en la definición de tipos:

```
create type Persona
    (nombre varchar(20) primary key,
     dirección varchar(20))
    ref from(nombre)
create table personas of Persona
    ref is id_personal derived
```

Téngase en cuenta que la definición de la tabla debe especificar que las referencia es derivada, y debe seguir especificando el nombre de un atributo autorreferencial. Al insertar una tupla de *departamentos* se puede usar

```
insert into departamentos
    values ('CS', 'Martín')
```

En SQL:1999 las referencias se desvinculan mediante el símbolo $->$. Considérese la tabla *departamentos* que se ha definido anteriormente. Se puede usar esta consulta para averiguar el nombre y la dirección de los directores de todos los departamentos:

select *director*−>*nombre*, *director*−>*dirección*
from *departamentos*

Las expresiones como "*director*−>*nombre*" se denominan **expresiones de camino**.

Dado que *director* es una referencia a una tupla de la tabla *personas*, el atributo *nombre* de la consulta anterior es el atributo *nombre* de la tupla de la tabla *personas*. Se pueden usar las referencias para ocultar las operaciones de reunión; en el ejemplo anterior, sin las referencias, el campo *director* de *departamento* se declararía como clave externa de la tabla *personas*. Para averiguar el nombre y la dirección del director de un departamento, hace falta una reunión explícita de las relaciones *departamentos* y *personas*. El uso de referencias simplifica considerablemente la consulta.

Se puede usar la operación **deref** para devolver la tupla a la que señala una referencia y luego tener acceso a sus atributos, como se muestra a continuación.

select deref(*director*).*nombre*
from *departamentos*

9.7 Implementación de las características O-R

Los sistemas de bases de datos relacionales orientadas a objetos son básicamente extensiones de los sistemas de bases de datos relacionales ya existentes. Las modificaciones resultan claramente necesarias en muchos niveles del sistema de bases de datos. Sin embargo, para minimizar las modificaciones en el código del sistema de almacenamiento (almacenamiento de relaciones, índices, etc.), los tipos de datos complejos soportados por los sistemas relacionales orientados a objetos se pueden traducir al sistema de tipos más sencillo de las bases de datos relacionales.

Para comprender la manera de hacer esta traducción, sólo hace falta mirar al modo en que algunas características del modelo E-R se traducen en relaciones. Por ejemplo, los atributos multivalorados del modelo E-R se corresponden con los atributos valorados como multiconjuntos del modelo relacional orientado a objetos. Los atributos compuestos se corresponden grosso modo con los tipos estructurados. Las jerarquías ES del modelo E-R se corresponden con la herencia de tablas del modelo relacional orientado a objetos.

Las técnicas para convertir las características del modelo E-R en tablas, que se estudiaron en el Apartado 6.9, se pueden usar, con algunas extensiones, para traducir los datos relacionales orientados a objetos a datos relacionales en el nivel de almacenamiento.

Las subtablas se pueden almacenar de manera eficiente, sin réplica de todos los campos heredados, de una de estas maneras:

- Cada tabla almacena la clave primaria (que puede haber heredado de una tabla madre) y los atributos que se definen de localmente. No hace falta almacenar los atributos heredados (que no sean la clave primaria), se pueden obtener mediante una reunión con la supertabla, de acuerdo con la clave primaria.
- Cada tabla almacena todos los atributos heredados y definidos localmente. Cuando se inserta una tupla, sólo se almacena en la tabla en la que se inserta, y su presencia se infiere en cada una de las supertablas. El acceso a todos los atributos de las tuplas es más rápido, ya que no hace falta ninguna reunión.

  No obstante, en caso de que el sistema de tipos permita que cada entidad se represente en dos subtablas sin estar presente en una subtabla común a ambas, esta representación puede dar lugar a la réplica de información. Además, resulta difícil traducir las claves externas que hacen referencia a una supertabla en restricciones de las subtablas; para implementar de manera eficiente esas claves externas hay que definir la supertabla como vista, y el sistema de bases de datos tiene que soportar las claves externas en las vistas.

Las implementaciones pueden decidir representar los tipos arrays y multiconjuntos directamente o usar internamente una representación normalizada. Las representaciones normalizadas tienden a ocupar más espacio y exigen un coste adicional en reuniones o agrupamientos para reunir los datos en

arrays o en multiconjuntos. Sin embargo, puede que las representaciones normalizadas resulten más sencillas de implementar.

Las interfaces de programas de aplicación ODBC y JDBC se han extendido para recuperar y almacenar tipos estructurados; por ejemplo, JDBC ofrece el método getObject(), que es parecido a getString() pero devuelve un objeto Java Struct, a partir del cual se pueden extraer los componentes del tipo estructurado. También es posible asociar clases de Java con tipos estructurados de SQL, y JDBC puede realizar la conversión entre los tipos. Véase el manual de referencia de ODBC o de JDBC para obtener más detalles.

9.8 Lenguajes de programación persistentes

Los lenguajes de las bases de datos se diferencian de los lenguajes de programación tradicionales en que trabajan directamente con datos que son persistentes; es decir, los datos siguen existiendo una vez que el programa que los creó haya concluido. Las relaciones de las bases de datos y las tuplas de las relaciones son ejemplos de datos persistentes. Por el contrario, los únicos datos persistentes con los que los lenguajes de programación tradicionales trabajan directamente son los archivos.

El acceso a las bases de datos es sólo un componente de las aplicaciones del mundo real. Mientras que los lenguajes para el tratamiento de datos como SQL son bastante efectivos en el acceso a los datos, se necesita un lenguaje de programación para implementar otros componentes de las aplicaciones como las interfaces de usuario o la comunicación con otras computadoras. La manera tradicional de realizar las interfaces de las bases de datos con los lenguajes de programación es incorporar SQL dentro del lenguaje de programación.

Los **lenguajes de programación persistentes** son lenguajes de programación extendidos con estructuras para el tratamiento de los datos persistentes. Los lenguajes de programación persistentes pueden distinguirse de los lenguajes con SQL incorporado, al menos, de dos maneras:

1. En los lenguajes incorporados el sistema de tipos del lenguaje anfitrión suele ser diferente del sistema de tipos del lenguaje para el tratamiento de los datos. Los programadores son responsables de las conversiones de tipos entre el lenguaje anfitrión y SQL. Hacer que los programadores lleven a cabo esta tarea presenta varios inconvenientes:
   - El código para la conversión entre objetos y tuplas opera fuera del sistema de tipos orientado a objetos y, por tanto, tiene más posibilidades de presentar errores no detectados.
   - La conversión en la base de datos entre el formato orientado a objetos y el formato relacional de las tuplas necesita gran cantidad de código. El código para la conversión de formatos, junto con el código para cargar y descargar los datos de la base de datos, puede suponer un porcentaje significativo del código total necesario para la aplicación.

   Por el contrario, en los lenguajes de programación persistentes, el lenguaje de consultas se halla totalmente integrado con el lenguaje anfitrión y ambos comparten el mismo sistema de tipos. Los objetos se pueden crear y guardar en la base de datos sin ninguna modificación explícita del tipo o del formato; los cambios de formato necesarios se realizan de manera transparente.
2. Los programadores que usan lenguajes de consultas incorporados son responsables de la escritura de código explícito para la búsqueda en la memoria de los datos de la base de datos. Si se realizan actualizaciones, los programadores deben escribir explícitamente código para volver a guardar los datos actualizados en la base de datos.

   Por el contrario, en los lenguajes de programación persistentes, los programadores pueden trabajar con datos persistentes sin tener que escribir explícitamente código para buscarlos en la memoria o volver a guardarlos en el disco.

En este apartado se describe la manera en que se pueden extender los lenguajes de programación orientados a objetos, como C++ y Java, para hacerlos lenguajes de programación persistentes. Las características de estos lenguajes permiten que los programadores trabajen con los datos directamente desde el lenguaje de programación, sin tener que recurrir a lenguajes de tratamiento de datos como SQL. Por tanto, ofrecen una integración más estrecha de los lenguajes de programación con las bases de datos que, por ejemplo, SQL incorporado.

Sin embargo, los lenguajes de programación persistentes presentan ciertos inconvenientes que hay que tener presentes al decidir si conviene usarlos. Dado que los lenguajes de programación suelen ser potentes, resulta relativamente sencillo cometer errores de programación que dañen las bases de datos. La complejidad de los lenguajes hace que la optimización automática de alto nivel, como la reducción de E/S de disco, resulte más difícil. En muchas aplicaciones el soporte de las consultas declarativas resulta de gran importancia, pero los lenguajes de programación persistentes no soportan bien actualmente las consultas declarativas.

En este capítulo se describen varios problemas teóricos que hay que abordar a la hora de añadir la persistencia a los lenguajes de programación ya existentes. En primer lugar se abordan los problemas independientes de los lenguajes y, en apartados posteriores, se tratan problemas que son específicos de los lenguajes C++ y Java. No obstante, no se tratan los detalles de las extensiones de los lenguajes; aunque se han propuesto varias normas, ninguna ha tenido una aceptación universal. Véanse las referencias de las notas bibliográficas para saber más sobre extensiones de lenguajes concretas y más detalles de sus implementaciones.

9.8.1 Persistencia de los objetos

Los lenguajes de programación orientados a objetos ya poseen un concepto de objeto, un sistema de tipos para definir los tipos de los objetos y constructores para crearlos. Sin embargo, esos objetos son *transitorios*; desaparecen en cuanto finaliza el programa, igual que ocurre con las variables de los programas en Pascal o en C. Si se desea transformar uno de estos lenguajes en un lenguaje para la programación de bases de datos, el primer paso consiste en proporcionar una manera de hacer persistentes a los objetos. Se han propuesto varios enfoques.

- **Persistencia por clases**. El enfoque más sencillo, pero el menos conveniente, consiste en declarar que una clase es persistente. Todos los objetos de la clase son, por tanto, persistentes de manera predeterminada. Todos los objetos de las clases no persistentes son transitorios.

  Este enfoque no es flexible, dado que suele resultar útil disponer en una misma clase tanto de objetos transitorios como de objetos persistentes. Muchos sistemas de bases de datos orientados a objetos interpretan la declaración de que una clase es persistente como si se afirmara que los objetos de la clase pueden hacerse persistentes, en vez de que todos los objetos de la clase son persistentes. Estas clases se pueden denominar con más propiedad clases "que pueden ser persistentes".

- **Persistencia por creación**. En este enfoque se introduce una sintaxis nueva para crear los objetos persistentes mediante la extensión de la sintaxis para la creación de los objetos transitorios. Por tanto, los objetos son persistentes o transitorios en función de la forma de crearlos. Varios sistemas de bases de datos orientados a objetos siguen este enfoque.

- **Persistencia por marcas**. Una variante del enfoque anterior es marcar los objetos como persistentes después de haberlos creado. Todos los objetos se crean como transitorios pero, si un objeto tiene que persistir más allá de la ejecución del programa, hay que marcarlo como persistente de manera explícita antes de que éste concluya. Este enfoque, a diferencia del anterior, pospone la decisión sobre la persistencia o la transitoriedad hasta después de la creación del objeto.

- **Persistencia por alcance**. Uno o varios objetos se declaran objetos persistentes (objetos raíz) de manera explícita. Todos los demás objetos serán persistentes si (y sólo si) se pueden alcanzar desde algún objeto raíz mediante una secuencia de una o varias referencias.

  Por tanto, todos los objetos a los que se haga referencia desde (es decir, cuyos identificadores de objetos se guarden en) los objetos persistentes raíz serán persistentes. Pero también lo serán todos los objetos a los que se haga referencia desde ellos, y los objetos a los que éstos últimos hagan referencia serán también persistentes, etc.

  Una ventaja de este esquema es que resulta sencillo hacer que sean persistentes estructuras de datos completas con sólo declarar como persistente su raíz. Sin embargo, el sistema de bases de datos sufre la carga de tener que seguir las cadenas de referencias para detectar los objetos que son persistentes, y eso puede resultar costoso.

9.8.2 Identidad de los objetos y punteros

En los lenguajes de programación orientados a objetos que no se han extendido para tratar la persistencia, cuando se crea un objeto el sistema devuelve un identificador del objeto transitorio. Los identificadores de objetos transitorios sólo son válidos mientras se ejecuta el programa que los ha creado; después de que concluya ese programa, el objeto se borra y el identificador pierde su sentido. Cuando se crea un objeto persistente se le asigna un identificador de objeto persistente.

El concepto de identidad de los objetos tiene una relación interesante con los punteros de los lenguajes de programación. Una manera sencilla de conseguir una identidad intrínseca es usar los punteros a las ubicaciones físicas de almacenamiento. En concreto, en muchos lenguajes orientados a objetos como C++, los identificadores de los objetos son en realidad punteros internos de la memoria.

Sin embargo, la asociación de los objetos con ubicaciones físicas de almacenamiento puede variar con el tiempo. Hay varios grados de permanencia de las identidades:

- **Dentro de los procedimientos**. La identidad sólo persiste durante la ejecución de un único procedimiento. Un ejemplo de identidad dentro de los programas son las variables locales de los procedimientos.
- **Dentro de los programas**. La identidad sólo persiste durante la ejecución de un único programa o de una única consulta. Un ejemplo de identidad dentro de los programas son las variables globales de los lenguajes de programación. Los punteros de la memoria principal o de la memoria virtual sólo ofrecen identidad dentro de los programas.
- **Entre programas**. La identidad persiste de una ejecución del programa a otra. Los punteros a los datos del sistema de archivos del disco ofrecen identidad entre los programas, pero pueden cambiar si se modifica la manera en que los datos se guardan en el sistema de archivos.
- **Persistente**. La identidad no sólo persiste entre las ejecuciones del programa sino también entre reorganizaciones estructurales de los datos. Es la forma persistente de identidad necesaria para los sistemas orientados a objetos.

En las extensiones persistentes de los lenguajes como C++, los identificadores de objetos de los objetos persistentes se implementan como "punteros persistentes". Un *puntero persistente* es un tipo de puntero que, a diferencia de los punteros internos de la memoria, sigue siendo válido después del final de la ejecución del programa y después de algunas modalidades de reorganización de los datos. Los programadores pueden usar los punteros persistentes del mismo modo que usan los punteros internos de la memoria en los lenguajes de programación. Conceptualmente los punteros persistentes se pueden considerar como punteros a objetos de la base de datos.

9.8.3 Almacenamiento y acceso a los objetos persistentes

¿Qué significa guardar un objeto en una base de datos? Evidentemente, hay que guardar por separado la parte de datos de cada objeto. Lógicamente, el código que implementa los métodos de las clases debe guardarse en la base de datos como parte de su esquema, junto con las definiciones de tipos de las clases. Sin embargo, muchas implementaciones se limitan a guardar el código en archivos externos a la base de datos para evitar tener que integrar el software del sistema, como los compiladores, con el sistema de bases de datos.

Hay varias maneras de hallar los objetos de la base de datos. Una manera es dar nombres a los objetos, igual que se hace con los archivos. Este enfoque funciona con un número de objetos relativamente pequeño, pero no resulta práctico para millones de objetos. Una segunda manera es exponer los identificadores de los objetos o los punteros persistentes a los objetos, que pueden guardarse en el exterior. A diferencia de los nombres, los punteros no tienen por qué ser fáciles de recordar y pueden ser, incluso, punteros físicos internos de la base de datos.

Una tercera manera es guardar conjuntos de objetos y permitir que los programas iteren sobre ellos para buscar los objetos deseados. Los conjuntos de objetos pueden a su vez modelarse como objetos de un *tipo conjunto*. Entre los tipos de conjuntos están los conjuntos, los multiconjuntos (es decir, conjuntos con varias apariciones posibles de un mismo valor), las listas, etc. Un caso especial de conjunto son

las *extensiones de clases*, que son el conjunto de todos los objetos pertenecientes a una clase. Si hay una extensión de clase para una clase dada, siempre que se crea un objeto de la clase ese objeto se inserta en la extensión de clase de manera automática; y, siempre que se borra un objeto, éste se elimina de la extensión de clase. Las extensiones de clases permiten que las clases se traten como relaciones en el sentido de que es posible examinar todos los objetos de una clase, igual que se pueden examinar todas las tuplas de una relación.

La mayor parte de los sistemas de bases de datos orientados a objetos soportan las tres maneras de acceso a los objetos persistentes. Dan identificadores a todos los objetos. Generalmente sólo dan nombre a las extensiones de las clases y a otros objetos de tipo conjunto y, quizás, a otros objetos seleccionados, pero no a la mayor parte de los objetos. Las extensiones de las clases suelen conservarse para todas las clases que puedan tener objetos persistentes pero, en muchas de las implementaciones, las extensiones de las clases sólo contienen los objetos persistentes de cada clase.

9.8.4 Sistemas persistentes de C++

En los últimos años han aparecido varias bases de datos orientadas a objetos basadas en las extensiones persistentes de C++ (véanse las notas bibliográficas). Hay diferencias entre ellas en términos de la arquitectura de los sistemas pero tienen muchas características comunes en términos del lenguaje de programación.

Varias de las características orientadas a objetos del lenguaje C++ ayudan a proporcionar un buen soporte para la persistencia sin modificar el propio lenguaje. Por ejemplo, se puede declarar una clase denominada Persistent_Object (objeto persistente) con los atributos y los métodos para dar soporte la persistencia; cualquier otra clase que deba ser persistente puede hacerse subclase de esta clase y heredará, por tanto, el soporte de la persistencia. El lenguaje C++ (igual que otros lenguajes modernos de programación) permite también redefinir los nombres de las funciones y los operadores estándar—como $+$, $-$, el operador de desvinculación de los punteros $->$, etc.—en función del tipo de operandos a los que se aplican. Esta posibilidad se denomina *sobrecarga*; se usa para redefinir los operadores para que se comporten de la manera deseada cuando operan con objetos persistentes.

Proporcionar apoyo a la persistencia mediante las bibliotecas de clases presenta la ventaja de que sólo se realizan cambios mínimos en C++; además, resulta relativamente fácil de implementar. Sin embargo, presenta el inconveniente de que los programadores tienen que usar mucho más tiempo para escribir los programas que trabajan con objetos persistentes y de que no les resulta sencillo especificar las restricciones de integridad del esquema ni ofrecer soporte para las consultas declarativas. Algunas implementaciones persistentes de C++ soportan extensiones de la sintaxis de C++ para facilitar estas tareas.

Es necesario abordar los siguientes problemas a la hora de añadir soporte a la persistencia a C++ (y a otros lenguajes):

- **Punteros persistentes**. Se debe definir un nuevo tipo de datos para que represente los punteros persistentes. Por ejemplo, la norma ODMG de C++ define la clase de plantillas d_Ref$< T >$ para que represente los punteros persistentes a la clase T. El operador de desvinculación para esta clase se vuelve a definir para que capture el objeto del disco (si no se halla ya presente en la memoria) y devuelva un puntero de la memoria al búfer en el que se ha capturado el objeto. Por tanto, si p es un puntero persistente a la clase T, se puede usar la sintaxis estándar como p$->$A o p$->$f(v) para tener acceso al atributo A de la clase T o para invocar al método f de la clase T.

  El sistema de bases de datos ObjectStore usa un enfoque diferente de los punteros persistentes. Utiliza los tipos normales de punteros para almacenar los punteros persistentes. Esto plantea dos problemas: (1) el tamaño de los punteros de la memoria sólo puede ser de 4 bytes, demasiado pequeño para las bases de datos mayores de 4 gigabytes, y (2) cuando se pasa algún objeto al disco los punteros de la memoria que señalan a su antigua ubicación física carecen de significado. ObjectStore usa una técnica denominada "rescate hardware" para abordar ambos problemas; precaptura los objetos de la base de datos en la memoria y sustituye los punteros persistentes por punteros de memoria y, cuando se vuelven a almacenar los datos en el disco, los punteros de memoria se sustituyen por punteros persistentes. Cuando se hallan en el disco, el valor almacenado

en el campo de los punteros de memoria no es el puntero persistente real; en vez de eso, se busca el valor en una tabla para averiguar el valor completo del puntero persistente.

- **Creación de objetos persistentes**. El operador new de C++ se usa para crear objetos persistentes mediante la definición de una versión "sobrecargada" del operador que usa argumentos adicionales especificando que deben crearse en la base de datos. Por tanto, en lugar de new T(), hay que llamar a new (bd) T() para crear un objeto persistente, donde bd identifica a la base de datos.
- **Extensiones de las clases**. Las extensiones de las clases se crean y se mantienen de manera automática para cada clase. La norma ODMG de C++ exige que el nombre de la clase se pase como parámetro adicional a la operación new. Esto también permite mantener varias extensiones para cada clase, pasando diferentes nombres.
- **Relaciones**. Las relaciones entre las clases se suelen representar almacenando punteros de cada objeto a los objetos con los que está relacionado. Los objetos relacionados con varios objetos de una clase dada almacenan un conjunto de punteros. Por tanto, si un par de objetos se halla en una relación, cada uno de ellos debe almacenar un puntero al otro. Los sistemas persistentes de C++ ofrecen una manera de especificar esas restricciones de integridad y de hacer que se cumplan mediante la creación y borrado automático de los punteros. Por ejemplo, si se crea un puntero de un objeto a a un objeto b, se añade de manera automática un puntero a a al objeto b.
- **Interfaz iteradora**. Dado que los programas tienen que iterar sobre los miembros de las clases, hace falta una interfaz para iterar sobre los miembros de las extensiones de las clases. La interfaz iteradora también permite especificar selecciones, de modo que sólo hay que capturar los objetos que satisfagan el predicado de selección.
- **Transacciones**. Los sistemas persistentes de C++ ofrecen soporte para comenzar las transacciones y para comprometerlas o provocar su retroceso.
- **Actualizaciones**. Uno de los objetivos de ofrecer soporte a la persistencia a los lenguajes de programación es permitir la persistencia transparente. Es decir, una función que opere sobre un objeto no debe necesitar saber que el objeto es persistente; por tanto, se pueden usar las mismas funciones sobre los objetos independientemente de que sean persistentes o no.

  Sin embargo, surge el problema de que resulta difícil detectar cuándo se ha actualizado un objeto. Algunas extensiones persistentes de C++ exigían que el programador especificara de manera explícita que se ha modificado el objeto llamando a la función mark_modified(). Además de incrementar el esfuerzo del programador, este enfoque aumenta la posibilidad de que se produzcan errores de programación que den lugar a una base de datos corrupta. Si un programador omite la llamada a mark_modified(), es posible que nunca se propague a la base de datos la actualización llevada a cabo por una transacción, mientras que otra actualización realizada por la misma transacción sí se propaga, lo que viola la atomicidad de las transacciones.

  Otros sistemas, como ObjectStore, usan soporte a la protección de la memoria proporcionada por el sistema operativo o por el hardware para detectar las operaciones de escritura en los bloques de memoria y marcar como sucios los bloques que se deban escribir posteriormente en el disco.
- **Lenguaje de consultas**. Los iteradores ofrecen soporte para consultas de selección sencillas. Para soportar consultas más complejas los sistemas persistentes de C++ definen un lenguaje de consultas.

Gran número de sistemas de bases de datos orientados a objetos basados en C++ se desarrollaron a finales de los años ochenta y principios de los noventa del siglo veinte. Sin embargo, el mercado para esas bases de datos resultó mucho más pequeño de lo esperado, ya que la mayor parte de los requisitos de las aplicaciones se cumplen de sobra usando SQL mediante interfaces como ODBC o JDBC. En consecuencia, la mayor parte de los sistemas de bases de datos orientados a objetos desarrollados en ese periodo ya no existen. En los años noventa el Grupo de Gestión de Datos de Objetos (Object Data Management Group, ODMG) definió las normas para agregar persistencia a C++ y a Java. No obstante,

el grupo concluyó sus actividades alrededor de 2002. ObjectStore y Versant son de los pocos sistemas de bases de datos orientados a objetos originales que siguen existiendo.

Aunque los sistemas de bases de datos orientados a objetos no encontraron el éxito comercial que esperaban, la razón de añadir persistencia a los lenguajes de programación sigue siendo válida. Hay varias aplicaciones con grandes exigencias de rendimiento que se ejecutan en sistemas de bases de datos orientados a objetos; el uso de SQL impondría una sobrecarga de rendimiento excesiva para muchos de esos sistemas. Con los sistemas de bases de datos relacionales orientados a objetos que proporcionan soporte para los tipos de datos complejos, incluidas las referencias, resulta más sencillo almacenar los objetos de los lenguajes de programación en bases de datos de SQL. Todavía puede emerger una nueva generación de sistemas de bases de datos orientadas a objetos que utilicen bases de datos relacionales orientadas a objetos como sustrato.

9.8.5 Sistemas Java persistentes

En años recientes el lenguaje Java ha visto un enorme crecimiento en su uso. La demanda de soporte de la persistencia de los datos en los programas de Java se ha incrementado de manera acorde. Los primeros intentos de creación de una norma para la persistencia en Java fueron liderados por el consorcio ODMG; posteriormente, el consorcio concluyó sus esfuerzos, pero transfirió su diseño al proyecto **Objetos de bases de datos de Java** (Java Database Objects, JDO), que coordina Sun Microsystems.

El modelo JDO para la persistencia de los objetos en los programas de Java es diferente del modelo de soporte de la persistencia en los programas de C++. Entre sus características se hallan:

- **Persistencia por alcance**. Los objetos no se crean explícitamente en la base de datos. El registro explícito de un objeto como persistente (usando el método makePersistent() de la clase PersistenceManager) hace que el objeto sea persistente. Además, cualquier objeto alcanzable desde un objeto persistente pasa a ser persistente.
- **Mejora del código de bytes**. En lugar de declarar en el código de Java que una clase es persistente, se especifican en un archivo de configuración (con la extensión .jdo) las clases cuyos objetos se pueden hacer persistentes. Se ejecuta un programa *mejorador* específico de la implementación que lee el archivo de configuración y lleva a cabo dos tareas. En primer lugar, puede crear estructuras en la base de datos para almacenar objetos de esa clase. En segundo lugar, modifica el código de bytes (generado al compilar el programa de Java) para que maneje tareas relacionadas con la persistencia. A continuación se ofrecen algunos ejemplos de modificaciones de este tipo.
  - Se puede modificar cualquier código que tenga acceso a un objeto para que compruebe primero si el objeto se halla en la memoria y, si no está, dé los pasos necesarios para ponerlo en la memoria.
  - Cualquier código que modifique un objeto se modifica para que también registre el objeto como modificado y, quizás, para que guarde un valor previo a la actualización que se usa en caso de que haga falta deshacerla (es decir, si se retrocede la transacción).

  También se pueden llevar a cabo otras modificaciones del código de bytes. Esas modificaciones del código de bytes son posibles porque el código de bytes es estándar en todas las plataformas e incluye mucha más información que el código objeto compilado.
- **Asignación de bases de datos**. JDO no define la manera en que se almacenan los datos en la base de datos subyacente. Por ejemplo, una situación frecuente es que los objetos se almacenen en una base de datos relacional. El programa mejorador puede crear en la base de datos un esquema adecuado para almacenar los objetos de las clases. La manera exacta en que lo hace depende de la implementación y no está definida por JDO. Se pueden asignar algunos atributos a los atributos relacionales, mientras que otros se pueden almacenar de forma serializada, que la base de datos trata como si fuera un objeto binario. Las implementaciones de JDO pueden permitir que los datos relacionales existentes se vean como objetos mediante la definición de la asignación correspondiente.
- **Extensiones de clase**. Las extensiones de clase se crean y se conservan de manera automática para cada clase declarada como persistente. Todos los objetos que se hacen persistentes se añaden

de manera automática a la extensión de clase correspondiente a a su clase. Los programas de JDO pueden tener acceso a las extensiones de clase e iterar sobre los miembros seleccionados. La interfaz Iteradora ofrecida por Java se puede usar para crear iteradores sobre las extensiones de clase y avanzar por los miembros de cada extensión de clase. JDO también permite que se especifiquen selecciones cuando se crea una extensión de clase y que sólo se capturen los objetos que satisfagan la selección.

- **Tipo de referencia único**. No hay diferencia de tipos entre las referencias a los objetos transitorios y las referencias a los objetos persistentes.

  Un enfoque para conseguir esta unificación de los tipos de puntero es cargar toda la base de datos en la memoria, lo que sustituye todos los punteros persistentes por punteros de memoria. Una vez llevadas a cabo las actualizaciones, el proceso se invierte y se vuelven a almacenar en el disco los objetos actualizados. Este enfoque es muy ineficiente para bases de datos de gran tamaño.

  A continuación se describirá un enfoque alternativo, que permite que los objetos persistentes se capturen en memoria de manera automática cuando hace falta, mientras que permite que todas las referencias contenidas en objetos de la memoria sean referencias de la memoria. Cuando se captura el objeto A, se crea un **objeto hueco** para cada objeto B_i al que haga referencia, y la copia de la memoria de A tiene referencias al objeto hueco correspondiente para cada B_i. Por supuesto, el sistema tiene que asegurar que, si ya se ha capturado el objeto B_i, la referencia apunta al objeto ya capturado en lugar de crear un nuevo objeto hueco. De manera parecida, si no se ha capturado el objeto B_i, pero otro objeto ya capturado hace referencia a él, ya tiene un objeto hueco creado para él; la referencia al objeto hueco ya existente se vuelve a usar, en lugar de crear un objeto hueco nuevo.

  Por tanto, para cada objeto O_i que se haya capturado, cada referencia de O_i es a un objeto ya capturado o a un objeto hueco. Los objetos huecos forman un *borde* que rodea los objetos capturados.

  Siempre que el programa tiene acceso real al objeto hueco O, el código de bytes mejorado lo detecta y captura el objeto en la base de datos. Cuando se captura este objeto, se lleva a cabo el mismo proceso de creación de objetos huecos para todos los objetos a los que O hace referencia. Tras esto, se permite que se lleve a cabo el acceso al objeto[3].

  Para implementar este esquema hace falta una estructura de índices en la memoria que asigne los punteros persistentes a las referencias de la memoria. Cuando se vuelven a escribir los objetos en el disco se usa ese índice para sustituir las referencias de la memoria por punteros persistentes en la copia que se escribe en el disco.

La norma JDO se halla todavía en una etapa preliminar y sometida a revisión. Varias empresas ofrecen implementaciones de JDO. No obstante, queda por ver si JDO se usará mucho, a diferencia de C++ de ODMG.

9.9 Sistemas orientados a objetos y sistemas relacionales orientados a objetos

Ya se han estudiado las bases de datos relacionales orientadas a objetos, que son bases de datos orientadas a objetos construidas sobre el modelo relacional, así como las bases de datos orientadas a objetos, que se crean alrededor de los lenguajes de programación persistentes.

3. La técnica que usa objetos huecos descrita anteriormente se halla estrechamente relacionada con la técnica de rescate hardware (ya mencionada en el Apartado 9.8.4). El rescate hardware la usan algunas implementaciones persistentes de C++ para ofrecer un solo tipo de puntero para los punteros persistentes y los de memoria. El rescate hardware utiliza técnicas de protección de la memoria virtual proporcionadas por el sistema operativo para detectar el acceso a las páginas y captura las páginas de la base de datos cuando es necesario. Por el contrario, la versión de Java modifica el código de bytes para que compruebe la existencia de objetos huecos en lugar de usar la protección de la memoria y captura los objetos cuando hace falta, en vez de capturar páginas enteras de la base de datos.

Las extensiones persistentes de loslenguajes de programación y los sistemas relacionales orientados a objetos se dirigen a mercados diferentes. La naturaleza declarativa y la limitada potencia (comparada con la de los lenguajes de programación) del lenguaje SQL proporcionan una buena protección de los datos respecto de los errores de programación y hacen que las optimizaciones de alto nivel, como la reducción de E/S, resulten relativamente sencillas (la optimización de las expresiones relacionales se trata en el Capítulo 14). Los sistemas relacionales orientados a objetos se dirigen a simplificar la realización de los modelos de datos y de las consultas mediante el uso de tipos de datos complejos. Entre las aplicaciones habituales están el almacenamiento y la consulta de datos complejos, incluidos los datos multimedia.

Los lenguajes declarativos como SQL, sin embargo, imponen una reducción significativa del rendimiento a ciertos tipos de aplicaciones que se ejecutan principalmente en la memoria principal y realizan gran número de accesos a la base de datos. Los lenguajes de programación persistentes se dirigen a las aplicaciones de este tipo que tienen necesidad de un rendimiento elevado. Proporcionan acceso a los datos persistentes con poca sobrecarga y eliminan la necesidad de traducir los datos si hay que tratarlos con un lenguaje de programación. Sin embargo, son más susceptibles de deteriorar los datos debido a los errores de programación y no suelen disponer de gran capacidad de consulta. Entre las aplicaciones habituales están las bases de datos de CAD.

Los puntos fuertes de los diversos tipos de sistemas de bases de datos pueden resumirse de la manera siguiente:

- **Sistemas relacionales**: tipos de datos sencillos, lenguajes de consultas potentes, protección elevada.
- **Bases de datos orientadas a objetos basadas en lenguajes de programación persistentes**: tipos de datos complejos, integración con los lenguajes de programación, elevado rendimiento.
- **Sistemas relacionales orientados a objetos**: tipos de datos complejos, lenguajes de consultas potentes, protección elevada.

Estas descripciones son válidas en general, pero hay que tener en cuenta que algunos sistemas de bases de datos no respetan estas fronteras. Por ejemplo, algunos sistemas de bases de datos orientados a objetos construidos alrededor de lenguajes de programación persistentes se pueden implementar sobre sistemas de bases de datos relacionales o sobre sistemas de bases de datos relacionales orientados a objetos. Puede que estos sistemas proporcionen menor rendimiento que los sistemas de bases de datos orientados a objetos construidos directamente sobre los sistemas de almacenamiento, pero proporcionan parte de las garantías de protección más estrictas propias de los sistemas relacionales.

9.10 Resumen

- El modelo de datos relacional orientado a objetos extiende el modelo de datos relacional al proporcionar un sistema de tipos enriquecido que incluye los tipos de conjuntos y la orientación a objetos.
- Los tipos de conjuntos incluyen las relaciones anidadas, los conjuntos, los multiconjuntos y los arrays, y el modelo relacional orientado a objetos permite que los atributos de las tablas sean conjuntos.
- La orientación a objetos proporciona herencia con subtipos y subtablas, así como referencias a objetos (tuplas).
- La norma SQL:1999 extiende el lenguaje de definición de datos y de consultas con los nuevos tipos de datos y la orientación a objetos.
- Se han estudiado varias características del lenguaje de definición de datos extendido, así como del lenguaje de consultas y, en especial, da soporte a los atributos valorados como conjuntos, a la herencia y a las referencias a las tuplas. Estas extensiones intentan conservar los fundamentos relacionales—en particular, el acceso declarativo a los datos—a la vez que se extiende la potencia de modelado.

- Los sistemas de bases de datos relacionales orientados a objetos (es decir, los sistemas de bases de datos basados en el modelo relacional orientado a objetos) ofrecen un camino de migración cómodo para los usuarios de las bases de datos relacionales que desean usar las características orientadas a objetos.
- Las extensiones persistentes de C++ y Javan integran la persistencia de forma elegante y ortogonalmente a sus elementos de programación previos, por lo que resulta fácil de usar.
- La norma ODMG define las clases y otros constructores para la creación y acceso a los objetos persistentes desde C++, mientras que la norma JDO ofrece una funcionalidad equivalente para Java.
- Se han estudiado las diferencias entre los lenguajes de programación persistentes y los sistemas relacionales orientados a objetos, y se han mencionado criterios para escoger entre ellos.

Términos de repaso

- Relaciones anidadas.
- Modelo relacional anidado.
- Tipos complejos.
- Tipos de conjuntos.
- Tipos de objetos grandes.
- Conjuntos.
- Arrays.
- Multiconjuntos.
- Tipos estructurados.
- Métodos.
- Tipos fila.
- Funciones constructoras.
- Herencia:
  - ☐ Simple.
  - ☐ Múltiple.
- Herencia de tipos.
- Tipo más concreto.
- Herencia de tablas.
- Subtabla.
- Solapamiento de subtablas.
- Tipos de referencia.
- Ámbito de una referencia.
- Atributo autorreferencial.
- Expresiones de camino.
- Anidamiento y desanidamiento.
- Funciones y procedimientos en SQL.
- Lenguajes de programación persistentes.
- Persistencia por:
  - ☐ Clase.
  - ☐ Creación.
  - ☐ Marcado.
  - ☐ Alcance.
- Enlace C++ de ODMG.
- ObjectStore.
- JDO
  - ☐ Persistencia por alcance.
  - ☐ Raíces.
  - ☐ Objetos huecos.
  - ☐ Asignación relacional de objetos.

Ejercicios prácticos

9.1 Una compañía de alquiler de coches tiene una base de datos con todos los vehículos de su flota actual. Para todos los vehículos incluye el número de bastidor, el número de matrícula, el fabricante, el modelo, la fecha de adquisición y el color. Se incluyen datos especiales para algunos tipos de vehículos:

- Camiones: capacidad de carga.
- Coches deportivos: potencia, edad mínima del arrendatario.
- Furgonetas: número de plazas.
- Vehículos todoterreno: altura de los bajos, eje motor (tracción a dos ruedas o a las cuatro).

Constrúyase una definición del esquema de esta base de datos de acuerdo con SQL:1999. Utilícese la herencia donde resulte conveniente.

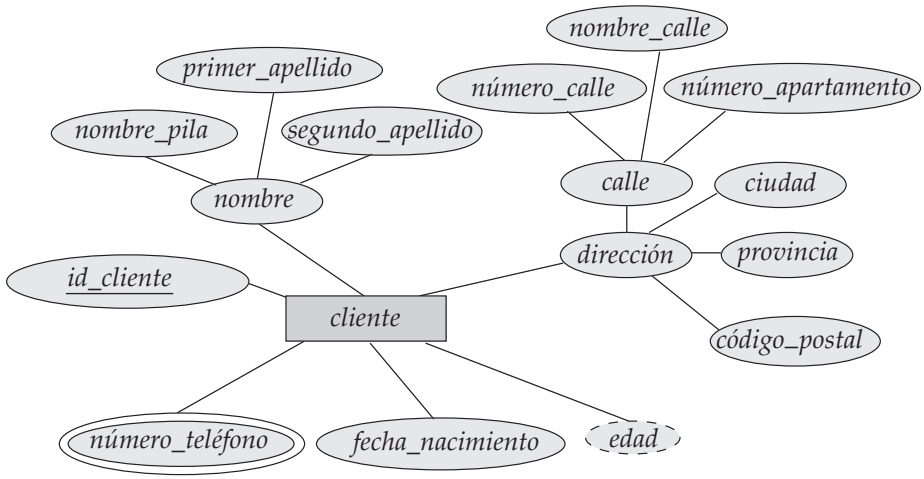


Figura 9.5 Diagrama E-R con atributos compuestos, multivalorados y derivados.

9.2 Considérese el esquema de la base de datos con la relación *Emp* cuyos atributos se muestran a continuación, con los tipos especificados para los atributos multivalorados.

Emp = (nombree, ConjuntoHijos **multiset***(HijosC), ConjuntoConocimientos* **multiset***(Conocimientos))*
Hijos = (nombre, cumpleaños)
Conocimientos = (mecanografía, ConjuntoExámenes **setof***(Exámenes))*
Exámenes = (año, ciudad)

a. Definir el esquema anterior en SQL:2003, con los tipos correspondientes para cada atributo.
b. Usando el esquema anterior, escribir las consultas siguientes en SQL:2003.
   I. Averiguar el nombre de todos los empleados que tienen un hijo nacido el 1 de enero de 2000 o en fechas posteriores.
   II. Averiguar los empleados que han hecho un examen del tipo de conocimiento "mecanografía" en la ciudad "San Rafael".
   III. Hacer una relación de los tipos de conocimiento de la relación *Emp*.

9.3 Considérese el diagrama E-R de la Figura 9.5, que contiene atributos compuestos, multivalorados y derivados.

a. Dese una definición de esquema en SQL:2003 correspondiente al diagrama E-R.
b. Dense constructores para cada uno de los tipos estructurados definidos.

9.4 Considérese el esquema relacional de la Figura 9.6.

a. Dese una definición de esquema en SQL:2003 correspondiente al esquema relacional, pero usando referencias para expresar las relaciones de clave externa.
b. Escríbanse cada una de las consultas del Ejercicio 2.9 sobre el esquema anterior usando SQL:2003.

9.5 Supóngase que se trabaja como asesor para escoger un sistema de bases de datos para la aplicación del cliente. Para cada una de las aplicaciones siguientes indíquese el tipo de sistema de bases de datos (relacional, base de datos orientada a objetos basada en un lenguaje de programación per-

empleado (*nombre_empleado*, *calle*, *ciudad*)
trabaja (*nombre_empleado*, *nombre_empresa*, sueldo)
empresa (*nombre_empresa*, *ciudad*)
supervisa (*nombre_empleado*, *nombre_jefe*)

Figura 9.6 Base de datos relacional para el Ejercicio práctico 9.4.

sistente, relacional orientada a objetos; no se debe especificar ningún producto comercial) que se recomendaría. Justifíquese la recomendación.

a. Sistema de diseño asistido por computadora para un fabricante de aviones.
b. Sistema para realizar el seguimiento de los donativos hechos a los candidatos a un cargo público
c. Sistema de información de ayuda para la realización de películas.

9.6 ¿En qué se diferencia el concepto de objeto del modelo orientado a objetos del concepto de entidad del modelo entidad-relación?

Ejercicios

9.7 Vuélvase a diseñar la base de datos del Ejercicio práctico 9.2 en la primera y en la cuarta formas normales. Indíquense las dependencias funcionales o multivaloradas que se den por supuestas. Indíquense también todas las restricciones de integridad referencial que deban incluirse en los esquemas de la primera y de la cuarta formas normales.

9.8 Considérese el esquema del Ejercicio práctico 9.2.

a. Dense instrucciones del LDD de SQL:2003 para crear la relación *EmpA*, que tiene la misma información que *Emp*, pero en la que los atributos valorados como multiconjuntos *ConjuntoNiños*, *ConjuntoConocimientos* y *ConjuntoExámenes* se sustituyen por los atributos valorados como arrays *ArrayHijos*, *ArrayConocimientos* y *ArrayExámenes*.
b. Escríbase una consulta para transformar los datos del esquema de *Emp* al de *EmpA*, con el array de los hijos ordenado por fecha de cumpleaños, el de conocimientos por el tipo de conocimientos y el de exámenes por el año de realización.
c. Escríbase una instrucción de SQL para actualizar la relación *Emp* añadiendo el hijo Jorge, con fecha de nacimiento de 5 de febrero de 2001 al empleado llamado Gabriel.
d. Escríbase una instrucción de SQL para llevar a cabo la misma actualización que antes, pero sobre la relación *EmpA*. Asegúrese de que el array de los hijos sigue ordenado por años.

9.9 Considérense los esquemas de la tabla *personas* y las tablas *estudiantes* y *profesores* que se crearon bajo *personas* en el Apartado 9.4. Dese un esquema relacional en la tercera forma normal que represente la misma información. Recuérdense las restricciones de las subtablas y dense todas las restricciones que deban imponerse en el esquema relacional para que cada ejemplar de la base de datos del esquema relacional pueda representarse también mediante un ejemplar del esquema con herencia.

9.10 Explíquese la diferencia entre el tipo x y el tipo de referencia **ref**(x). ¿En qué circunstancias se debe escoger usar el tipo de referencia?

9.11
a. Dese una definición de esquema en SQL:1999 del diagrama E-R de la Figura 9.7, que contiene especializaciones, usando subtipos y subtablas.
b. Dese una consulta de SQL:1999 para averiguar el nombre de todas las personas que no son secretarias.
c. Dese una consulta de SQL:1999 para imprimir el nombre de las personas que no sean empleados ni clientes.
d. ¿Se puede crear una persona que sea a la vez empleado y cliente con el esquema que se acaba de crear? Explíquese la manera de hacerlo o el motivo de que no sea posible.

9.12 Supóngase que una base de datos de JDO tiene un objeto A, que hace referencia al objeto B, que, a su vez, hace referencia al objeto C. Supóngase que todos los objetos se hallan inicialmente en el disco. Supóngase que un programa desvincula en primer lugar a A, luego a B siguiendo la referencia de A y, finalmente, desvincula a C. Muéstrense los objetos que están en representados en memoria tras cada desvinculación, junto con su estado (hueco o lleno, y los valores de los campos de referencia).

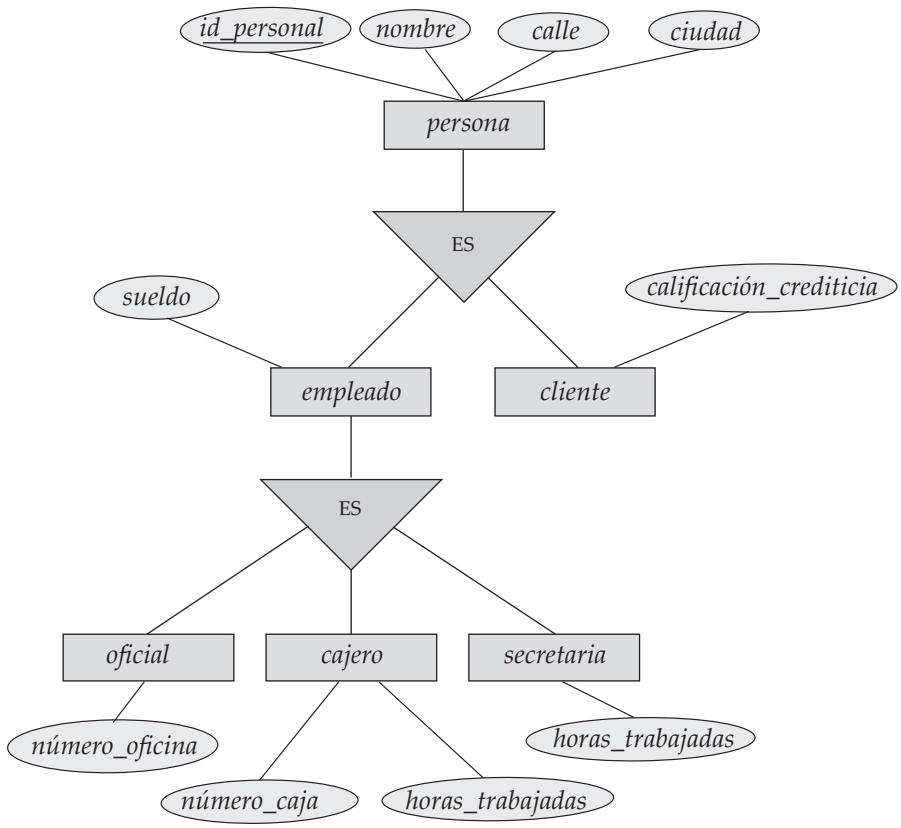


Figura 9.7 Especialización y generalización.

Notas bibliográficas

Se han propuesto varias extensiones de SQL orientadas a objetos. POSTGRES (Stonebraker y Rowe [1986] y Stonebraker [1986]) fue una de las primeras implementaciones de un sistema relacional orientado a objetos. Otros sistemas relacionales orientados a objetos de la primera época son las extensiones de SQL de O_2 (Bancilhon et al. [1989]) y UniSQL (UniSQL [1991]). SQL:1999 fue producto de un amplio (y muy tardío) esfuerzo de normalización, que se inició originalmente mediante el añadido a SQL de características orientadas a objetos y acabó añadiendo muchas más características, como las estructuras procedimentales que se han visto anteriormente. El soporte de los tipos multiconjuntos se añadió como parte de SQL:2003.

Entre los libros de texto sobre SQL:1999 están Melton y Simon [2001] y Melton [2002]; el último se centra en las características relacionales orientadas a objetos de SQL:1999. Eisenberg et al. [2004] ofrece una visión general de SQL:2003, incluido el soporte de los multiconjuntos. Se deben consultar los manuales (en línea) del sistema de bases de datos que se utilice para averiguar las características de SQL:1999 y de SQL:2003 que soporta.

A finales de los años ochenta y principios de los años noventa del siglo veinte se desarrollaron varios sistemas de bases de datos orientadas a objetos. Entre los más notables de los comerciales figuran ObjectStore (Lamb et al. [1991]), O_2 (Lecluse et al. [1988]) y Versant. La norma para bases de datos orientadas a objetos ODMG se describe con detalle en Cattell [2000]. JDO se describe en Roos [2002], Tyagi et al. [2003] y Jordan y Russell [2003].

Herramientas

Hay diferencias considerables entre los diversos productos de bases de datos en cuanto al soporte de las características relacionales orientadas a objetos. Probablemente Oracle tenga el soporte más amplio entre los principales fabricantes de bases de datos. El sistema de bases de datos de Informix ofrece so-

porte para muchas características relacionales orientadas a objetos. Tanto Oracle como Informix ofrecían características relacionales orientadas a objetos antes de la finalización de la norma SQL:1999 y presentan algunas características que no forman parte de SQL:1999.

La información sobre ObjectStore y sobre Versant, incluida la descarga de versiones de prueba, se puede obtener de sus respectivos sitios Web (objectstore.com y versant.com). El proyecto Apache DB (db.apache.org) ofrece una herramienta de asignación relacional orientada a objetos para Java que soporta tanto Java de ODMG como las APIs de JDO. Se puede obtener una implementación de referencia de JDO de sun.com; se puede usar un motor de búsqueda para averiguar el URL completo.

CAPÍTULO 10

XML

A diferencia de la mayor parte de las tecnologías presentadas en los capítulos anteriores, el **lenguaje de marcas extensible** (Extensible Markup Language, **XML**) no se concibió como una tecnología para bases de datos. En realidad, al igual que el *lenguaje de marcas de hipertexto* (*Hyper-Text Markup Language*, HTML) en el que está basado la World Wide Web, XML tiene sus raíces en la gestión de documentos y está derivado de un lenguaje para estructurar documentos grandes conocido como *lenguaje estándar generalizado de marcas* (*Standard Generalized Markup Language*, SGML). Sin embargo, a diferencia de SGML y de HTML, XML puede representar datos de bases de datos, así como muchas clases de datos estructurados. Es particularmente útil como formato de datos cuando las aplicaciones se deben comunicar con otra aplicación o integrar información de varias aplicaciones. Cuando XML se usa en estos contextos, se generan muchas dudas sobre las bases de datos, incluyendo cómo organizar, manipular y consultar los datos XML. En este capítulo se introduce XML y se estudia la gestión de los datos XML con las técnicas de bases de datos, así como el intercambio de datos con formato como documentos XML.

10.1 Motivación

Para comprender XML es importante entender sus raíces como un lenguaje de marcas de documentos. El término **marca** se refiere a cualquier elemento en un documento del que no se tiene intención que sea parte de la salida impresa. Por ejemplo, un escritor que crea un texto que finalmente se compone en una revista puede desear realizar notas sobre cómo se ha de realizar la composición. Sería importante introducir estas notas de tal forma que se pudieran distinguir del contenido real; una nota como "usar un tipo mayor y poner en negrita" o "no romper este párrafo" no acabaría impresa en la revista. Estas notas comunican información adicional sobre el texto. En un procesamiento electrónico de documentos un **lenguaje de marcas** es una descripción formal de la parte del documento que es contenido, la parte que es marca y lo que significa la marca.

Así como los sistemas de bases de datos evolucionaron desde el procesamiento físico de archivos para proporcionar una vista lógica aislada, los lenguajes de marcas evolucionaron desde la especificación de instrucciones que indicaban cómo imprimir partes del documento para la *función* del contenido. Por ejemplo, con marcas funcionales, el texto que representa los encabezamientos de sección (para este apartado, la palabra "Motivación") se marcaría como un encabezamiento de apartado en lugar de marcarse como un texto con el fin de imprimirse en tamaño grande y negrita. Desde el punto de vista del editor, dicha marca funcional permite que el documento tenga distintos formatos en contextos diferentes. También ayuda a que distintas partes de un documento largo, o distintas páginas en un sitio Web grande, tengan un formato uniforme. Más importante, la marca funcional también ayuda a registrar lo que representa semánticamente cada parte del texto y ayuda a la extracción automática de partes claves de los documentos.

Para la familia de lenguajes de marcado, en los que se incluyen HTML, SGMLy XML, las marcas adoptan la forma de **etiquetas** encerradas entre corchetes angulares, <>. Las etiquetas se usan en pares, con

```
<banco>
    <cuenta>
        <número_cuenta> C-101 </número_cuenta>
        <nombre_sucursal> Centro </nombre_sucursal>
        <saldo> 500 </saldo>
    </cuenta>
    <cuenta>
        <número_cuenta> C-102 </número_cuenta>
        <nombre_sucursal> Navacerrada </nombre_sucursal>
        <saldo> 400 </saldo>
    </cuenta>
    <cuenta>
        <número_cuenta> C-201 </número_cuenta>
        <nombre_sucursal> Galapagar </nombre_sucursal>
        <saldo> 900 </saldo>
    </cuenta>
    <cliente>
        <nombre_cliente> González </nombre_cliente>
        <calle_cliente> Arenal </calle_cliente>
        <ciudad_cliente> La Granja </ciudad_cliente>
    </cliente>
    <cliente>
        <nombre_cliente> López </nombre_cliente>
        <calle_cliente> Mayor </calle_cliente>
        <ciudad_cliente> Peguerinos </ciudad_cliente>
    </cliente>
    <impositor>
        <número_cuenta> C-101 </número_cuenta>
        <nombre_cliente> González </nombre_cliente>
    </impositor>
    <impositor>
        <número_cuenta> C-201 </número_cuenta>
        <nombre_cliente> González </nombre_cliente>
    </impositor>
    <impositor>
        <número_cuenta> C-102 </número_cuenta>
        <nombre_cliente> López </nombre_cliente>
    </impositor>
</banco>
```

Figura 10.1 Representación XML de información bancaria.

<etiqueta> y </etiqueta> delimitando el comienzo y final de la porción de documento a la cual se refiere la etiqueta. Por ejemplo, el título de un documento podría estar marcado de la siguiente forma:

```
<title>Fundamentos de bases de datos</title>
```

A diferencia de HTML, XML no impone las etiquetas permitidas, y se pueden elegir como sea necesario para cada aplicación. Esta característica es la clave de la función principal de XML en la representación e intercambio de datos, mientras que HTML se usa principalmente para el formato de documentos.

Por ejemplo, en nuestra aplicación del banco, la información de la cuenta y del cliente se puede representar como parte de un documento XML, como se muestra en la Figura 10.1. Obsérvese el uso de etiquetas tales como cuenta y número_cuenta. Estas etiquetas proporcionan el contexto de cada valor y permiten identificar la semántica del valor. Para este ejemplo, la representación de datos XML no

proporciona ninguna ventaja significativa con respecto a la representación relacional; sin embargo, se usa este ejemplo por su simplicidad.

La Figura 10.2, que muestra la forma en que se puede representar un pedido de compra en XML, ilustra un uso más realista de XML. Los pedidos de compra se generan habitualmente por una empresa y se envían a otra. Tradicionalmente el comprador los imprimía en papel y los enviaba al proveedor; éste reintroducía manualmente los datos en el sistema informático. Este lento proceso se puede acelerar en gran medida enviando electrónicamente la información entre el comprador y el proveedor. La representación anidada permite que toda la información de un pedido de compra se represente de forma natural en un único documento (los pedidos de compra reales tienen considerablemente más información que la que se muestra en este ejemplo simple). XML proporciona una forma estándar de etiquetar los datos; obviamente, las dos empresas deben estar de acuerdo en las etiquetas que aparecen en el pedido de compra y lo que significan.

Comparado con el almacenamiento de los datos en una base de datos relacional, la representación XML puede parecer poco eficiente, puesto que los nombres de las etiquetas se repiten por todo el documento. Sin embargo, a pesar de esta desventaja, una representación XML presenta ventajas significativas cuando se usa para el intercambio de datos entre empresas y para almacenar información estructurada en archivos.

- En primer lugar la presencia de las etiquetas hace que el mensaje sea **autodocumentado**, es decir, no se tiene que consultar un esquema para comprender el significado del texto. Por ejemplo, se puede leer fácilmente el fragmento anterior.

- En segundo lugar, el formato del documento no es rígido. Por ejemplo, si algún remitente agrega información adicional tal como una etiqueta último_acceso que informa de la última fecha en la que se ha accedido a la cuenta, el recipiente de los datos XML puede sencillamente ignorar la etiqueta. Como ejemplo adicional, en la Figura 10.2, el elemento con el identificador SG2 tiene una etiqueta denominada unidad_de_medida, que el primer elemento no tiene. La etiqueta se requiere en los elementos que se ordenan por peso o volumen, y se puede omitir en elementos que simplemente se ordenan por número.

  La capacidad de reconocer e ignorar las etiquetas inesperadas permite al formato de los datos evolucionar con el tiempo sin invalidar las aplicaciones existentes. De forma similar, la posibilidad de que la misma etiqueta aparezca varias veces hace que sea fácil representar atributos multivalorados.

- En tercer lugar, XML permite estructuras anidadas. El pedido de compra mostrado en la Figura 10.2 ilustra las ventajas de tener estas estructuras. Cada pedido de compra tiene un comprador y una lista de elementos como dos de sus estructuras anidadas. Cada elemento tiene a su vez un identificador, una descripción y un precio anidados, mientras que el comprador tiene un nombre y dirección anidados.

  Esta información se podría haber dividido en varias relaciones en el modelo relacional. La información de los elementos se habría almacenado en una relación, la información del comprador en una segunda relación, los pedidos de compra en una tercera y la relación entre los pedidos de compra, compradores y elementos en una cuarta.

  La representación relacional ayuda a evitar la redundancia; por ejemplo, las descripciones de los elementos se almacenarían sólo una vez por cada identificador de elemento en un esquema relacional normalizado. Sin embargo, en el pedido de compra de XML, las descripciones se pueden repetir en varios pedidos con el mismo elemento. No obstante, es atractivo recoger toda la información relacionada con un pedido en una única estructura anidada, incluso añadiendo redundancia, cuando la información se tiene que intercambiar con terceros.

- Finalmente, puesto que el formato XML está ampliamente aceptado hay una gran variedad de herramientas disponibles para ayudar a su procesamiento, incluyendo APIs para lenguajes de programación para crear y leer datos XML, software de exploración y herramientas de bases de datos.

```
<pedido_compra>
    <identificador> P-101 </identificador>
    <comprador>
        <nombre> Coyote Loco </nombre>
        <dirección> Mesa Flat, Route 66, Arizona 12345, EEUU</dirección>
    </comprador>
    <proveedor>
        <nombre> Proveedores Acme SA</nombre>
        <dirección> 1, Broadway, Nueva York, NY, EEUU</dirección>
    </proveedor>
    <lista_elementos>
        <elemento>
            <identificador>EA1</identificador>
            <descripción> Trineo propulsado por cohetes atómicos</descripción>
            <cantidad> 2 </cantidad>
            <precio> 199.95 </precio>
        </elemento>
        <elemento>
            <identificador>PF2</identificador>
            <descripción> Pegamento fuerte</descripción>
            <cantidad> 1 </cantidad>
            <unidad_de_medida> litro </unidad_de_medida>
            <precio> 29.95 </precio>
        </elemento>
    </lista_elementos>
    <coste_total> 429.85 </coste_total>
    <forma_de_pago> Reembolso </forma_de_pago>
    <forma_de_envío> Avión </forma_de_envío>
</pedido_compra>
```

Figura 10.2 Representación XML de un pedido de compra.

Más adelante se describen varias aplicaciones de XML en la sección 10.7. Al igual que SQL es el *lenguaje* dominante para consultar los datos relacionales, XML se está convirtiendo en el *formato* dominante para el intercambio de datos.

10.2 Estructura de los datos XML

El constructor fundamental en un documento XML es el **elemento**. Un elemento es sencillamente un par de etiquetas de inicio y finalización coincidentes y todo el texto que aparece entre ellas. Los documentos XML deben tener un único elemento **raíz** que abarque al resto de elementos en el documento. En el ejemplo de la Figura 10.1 el elemento <banco> forma el elemento raíz. Además, los elementos en un documento XML se deben **anidar** adecuadamente. Por ejemplo:

```
...
    <cuenta>
        Esta cuenta se usa muy rara vez, por no decir nunca.
        <número_cuenta> C-102 </número_cuenta>
        <nombre_sucursal> Navacerrada </nombre_sucursal>
        <saldo> 400 </saldo>
    </cuenta>
...
```

Figura 10.3 Mezcla de texto con subelementos.

<cuenta> . . . <saldo> . . . </saldo> . . . </cuenta>

está anidado adecuadamente, mientras que

<cuenta> . . . <saldo> . . . </cuenta> . . . </saldo>

no está adecuadamente anidado.

Aunque el anidamiento adecuado es una propiedad intuitiva, es necesario definirla más formalmente. Se dice que el texto aparece **en el contexto de** un elemento si aparece entre la etiqueta de inicio y la etiqueta de finalización de dicho elemento. Las etiquetas están anidadas adecuadamente si toda etiqueta de inicio tiene una única etiqueta de finalización coincidente que está en el contexto del mismo elemento padre.

Obsérvese que el texto puede estar mezclado con los subelementos de otro elemento, como en la Figura 10.3. Como con otras características de XML, esta libertad tiene más sentido en un contexto de procesamiento de documentos que en el contexto de procesamiento de datos y no es particularmente útil para representar en XML datos más estructurados, como es el contenido de las bases de datos.

La capacidad de anidar elementos con otros elementos proporciona una forma alternativa de representar información. La Figura 10.4 muestra una representación de la información bancaria de la Figura 10.1, pero con los elementos cuenta anidados con los elementos cliente, aunque almacenaría elementos cuenta de una forma redundante si pertenecen a varios clientes.

Las representaciones anidadas se usan ampliamente en las aplicaciones de intercambio de datos XML para evitar las reuniones. Por ejemplo, un pedido de compra almacenaría la dirección completa del emisor y receptor de una forma redundante en varios pedidos de compra, mientras que una representación normalizada puede requerir una reunión de registros de pedidos de compras con una relación *dirección_empresa* para obtener la información de la dirección.

```
<banco-1>
    <cliente>
        <nombre_cliente> González </nombre_cliente>
        <calle_cliente> Arenal </calle_cliente>
        <ciudad_cliente> La Granja </ciudad_cliente>
        <cuenta>
            <número_cuenta> C-101 </número_cuenta>
            <nombre_sucursal> Centro </nombre_sucursal>
            <saldo> 500 </saldo>
        </cuenta>
        <cuenta>
            <número_cuenta> C-201 </número_cuenta>
            <nombre_sucursal> Galapagar </nombre_sucursal>
            <saldo> 900 </saldo>
        </cuenta>
    </cliente>
    <cliente>
        <nombre_cliente> López </nombre_cliente>
        <calle_cliente> Mayor </calle_cliente>
        <ciudad_cliente> Peguerinos </ciudad_cliente>
        <cuenta>
            <número_cuenta> C-102 </número_cuenta>
            <nombre_sucursal> Navacerrada </nombre_sucursal>
            <saldo> 400 </saldo>
        </cuenta>
    </cliente>
</banco-1>
```

Figura 10.4 Representación XML anidada de información bancaria.

```
...
    <cuenta tipo_cuenta= "corriente">
        <número_cuenta> C-102 </número_cuenta>
        <nombre_sucursal> Navacerrada </nombre_sucursal>
        <saldo> 400 </saldo>
    </cuenta>
...
```

Figura 10.5 Uso de atributos.

Además de los elementos, XML especifica la noción de **atributo**. Por ejemplo, el tipo de una cuenta se puede representar como un atributo, como en la Figura 10.5. Los atributos de un elemento aparecen como pares *nombre=valor* antes del cierre ">" de una etiqueta. Los atributos son cadenas y no contienen marcas. Además, los atributos pueden aparecer solamente una vez en una etiqueta dada, al contrario que los subelementos, que pueden estar repetidos.

Obsérvese que en un contexto de construcción de un documento es importante la distinción entre subelemento y atributo (un atributo es implícitamente texto que no aparece en el documento impreso o visualizado). Sin embargo, en las aplicaciones de bases de datos y de intercambio de datos de XML esta distinción es menos relevante y la elección de representar los datos como un atributo o un subelemento es frecuentemente arbitraria.

Una nota sintáctica final es que un elemento de la forma <elemento></elemento>, que no contiene subelementos o texto, se puede abreviar como <elemento/>; los elementos abreviados pueden, no obstante, contener atributos.

Puesto que los documentos XML se diseñan para su intercambio entre aplicaciones se tiene que introducir un mecanismo de **espacio de nombres** para permitir a las organizaciones especificar nombre únicos globalmente para que se usen como marcas de elementos en los documentos. La idea de un espacio de nombres es anteponer cada etiqueta o atributo con un identificador de recursos universal (por ejemplo, una dirección Web). Por ello, por ejemplo, si Banco Principal deseara asegurar que los documentos XML creados no duplican las etiquetas usadas por los documentos de otros socios del negocio, se puede anteponer un identificador único con dos puntos a cada nombre de etiqueta. El banco puede usar un URL Web como el siguiente

http://www.BancoPrincipal.com

como un identificador único. El uso de identificadores únicos largos en cada etiqueta puede ser poco conveniente por lo que el espacio de nombres estándar proporciona una forma de definir una abreviatura para los identificadores.

En la Figura 10.6 el elemento raíz (banco) tiene un atributo xmlns:BP, que declara que BP está definido como abreviatura del URL dado anteriormente. Se puede usar entonces la abreviatura en varias marcas de elementos como se ilustra en la figura.

Un documento puede tener más de un espacio de nombres, declarado como parte del elemento raíz. Se puede entonces asociar elementos diferentes con espacios de nombres distintos. Se puede definir un *espacio de nombres predeterminado* mediante el uso del atributo xmlns en lugar de xmlns:BP en el elemento raíz. Los elementos sin un prefijo de espacio de nombres explícito pertenecen entonces al espacio de nombres predeterminado.

```
<banco xmlns:BP="http://www.BancoPrincipal.com">
    ...
    <BP:sucursal>
        <BP:nombre_sucursal> Centro </BP:nombre_sucursal>
        <BP:ciudad_sucursal> Arganzuela </BP:ciudad_sucursal>
    </BP:sucursal>
    ...
</banco>
```

Figura 10.6 Nombres únicos de etiqueta mediante el uso de espacios de nombres.

Algunas veces es necesario almacenar valores que contienen etiquetas sin que se interpreten como etiquetas XML. Para ello, XML permite esta construcción:

<![CDATA[<cuenta> ···</cuenta>]]>

Debido a que el texto <cuenta> está encerrado en CDATA, se trata como datos de texto normal, no como una etiqueta. El término CDATA viene de datos de tipo carácter ("character data" en inglés).

10.3 Esquema de los documentos XML

Las bases de datos contienen esquemas que se usan para restringir qué información se puede almacenar en la base de datos y para restringir los tipos de datos de la información almacenada. En cambio, los documentos XML se pueden crear de forma predeterminada sin un esquema asociado. Un elemento puede tener entonces cualquier subelemento o atributo. Aunque dicha libertad puede ser aceptable algunas veces dada la naturaleza autodescriptiva del formato de datos, no es útil generalmente cuando los documentos XML se deben procesar automáticamente como parte de una aplicación o incluso cuando se van a dar formato en XML a grandes cantidades de datos relacionados. Aquí se describirá el primer lenguaje de definición de esquemas incluido como parte de la norma XML, la definición de tipos de documentos (*Document Type Definition*), así como su sustituto *XML Schema*, definido más recientemente. También se usa otro lenguaje de definición de esquemas denominado Relax NG, pero no se estudia aquí; para obtener más información sobre Relax NG, consúltense las referencias en el apartado de notas bibliográficas.

10.3.1 Definición de tipos de documentos

La **definición de tipos de documentos** (Document Type Definition, **DTD**) es una parte opcional de un documento XML. El propósito principal de DTD es similar al de un esquema: restringir el tipo de información presente en el documento. Sin embargo, DTD no restringe en realidad los tipos en el sentido de tipos básicos como entero o cadena. En su lugar solamente restringe el aspecto de subelementos y atributos en un elemento. DTD es principalmente una lista de reglas que indican el patrón de subelementos que aparecen en un elemento. La Figura 10.7 muestra una parte de una DTD de ejemplo de un documento de información bancaria; el documento XML de la Figura 10.1 se ajusta a esa DTD.

Cada declaración está en la forma de una expresión normal para los subelementos de un elemento. Así, en la DTD de la Figura 10.7 un elemento bancario consiste en uno o más elementos cuenta, cliente o impositor; el operador | especifica "o" mientras que el operador + especifica "uno o más". Aunque no se muestra aquí, el operador $*$ se usa para especificar "cero o más" mientras que el operador ? se usa para especificar un elemento opcional (es decir, "cero o uno").

El elemento cuenta se define para contener los subelementos número_cuenta, nombre_sucursal y saldo (en ese orden). De forma similar, cliente y impositor tienen los atributos en su esquema definidos como subelementos.

```
<!DOCTYPE banco [
    <!ELEMENT banco ( (cuenta|cliente|impositor)+)>
    <!ELEMENT cuenta ( número_cuenta nombre_sucursal saldo )>
    <!ELEMENT cliente ( nombre_cliente calle_cliente ciudad_cliente )>
    <!ELEMENT impositor ( nombre_cliente número_cuenta )>
    <!ELEMENT número_cuenta ( #PCDATA )>
    <!ELEMENT nombre_sucursal ( #PCDATA )>
    <!ELEMENT saldo( #PCDATA )>
    <!ELEMENT nombre_cliente( #PCDATA )>
    <!ELEMENT calle_cliente( #PCDATA )>
    <!ELEMENT ciudad_cliente( #PCDATA )>
] >
```

Figura 10.7 Ejemplo de DTD.

Finalmente, se declara a los elementos número_cuenta, nombre_sucursal, saldo, nombre_cliente, calle_cliente y ciudad_cliente del tipo #PCDATA. La palabra clave #PCDATA indica dato de texto; su nombre deriva históricamente de "parsed character data" (datos analizados de tipo carácter). Otros dos tipos especiales de declaraciones son empty (vacío) que indica que el elemento no tiene ningún contenido y any (cualquiera) que indica que no hay restricción sobre los subelementos del elemento; es decir, cualquier elemento, incluso los no mencionados en la DTD, puede ser subelemento del elemento. La ausencia de una declaración para un subelemento es equivalente a declarar explícitamente el tipo como any.

Los atributos permitidos para cada elemento también se declaran en la DTD. Al contrario que los subelementos no se impone ningún orden a los atributos. Los atributos se pueden especificar del tipo CDATA, ID, IDREFo IDREFS; el tipo CDATA simplemente dice que el atributo contiene datos de caracteres mientras que los otros tres no son tan sencillos; se explicarán detalladamente en breve. Por ejemplo, la siguiente línea de una DTD especifica que el elemento cuenta tiene un atributo del tipo tipo_cuenta con valor predeterminado corriente.

```
<!ATTLIST cuenta tipo_cuenta CDATA "corriente" >
```

Los atributos deben tener una declaración de tipo y una declaración predeterminada. La declaración predeterminada puede consistir en un valor predeterminado para el atributo o #REQUIRED, lo que quiere decir que se debe especificar un valor para el atributo en cada elemento, #IMPLIED, lo que significa que no se ha proporcionado ningún valor predeterminado y se puede omitir este atributo en el documento. Si un atributo tiene un valor predeterminado, para cada elemento que no tenga especificado un valor para el atributo el valor se rellena automáticamente cuando se lee el documento XML.

Un atributo de tipo ID proporciona un identificador único para el elemento; un valor de un atributo ID de un elemento no debe aparecer en ningún otro elemento del mismo documento. Sólo se permite que un único atributo de un elemento sea de tipo ID.

Un atributo del tipo IDREF es una referencia a un elemento; el atributo debe contener un valor que aparezca en el atributo ID de algún elemento en el documento. El tipo IDREFS permite una lista de referencias, separadas por espacios.

La Figura 10.8 muestra una DTD de ejemplo en la que las relaciones de la cuenta de un cliente se representan mediante los atributos ID e IDREFS, en lugar de hacerlo mediante los registros impositor. Los elementos cuenta usan número_cuenta como atributo identificador; para realizar esto se ha hecho que número_cuenta sea atributo de cuenta en lugar de subelemento. Los elementos cliente tienen un nuevo atributo identificador denominado id_cliente. Además, cada elemento cliente contiene un atributo cuentas del tipo IDREFS, que es una lista de identificadores de las cuentas que tiene abiertas el cliente. Cada elemento cuenta tiene un atributo titulares del tipo IDREFS, que es una lista de titulares de la cuenta.

La Figura 10.9 muestra un ejemplo de documento XML basado en la DTD de la Figura 10.8. Obsérvese que se usa un conjunto distinto de cuentas y clientes del ejemplo anterior con el fin de ilustrar mejor la característica IDREFS.

```
<!DOCTYPE banco-2 [
    <!ELEMENT cuenta ( sucursal, saldo )>
    <!ATTLIST cuenta
        número_cuenta ID #REQUIRED
        titulares IDREFS #REQUIRED >
    <!ELEMENT cliente ( nombre_cliente, calle_cliente, ciudad_cliente )>
    <!ATTLIST cliente
        id_cliente ID #REQUIRED
        cuentas IDREFS #REQUIRED >
    ··· declaraciones para sucursal, saldo, nombre_cliente,
        calle_cliente y ciudad_cliente ···
] >
```

Figura 10.8 DTD con los tipos de atributo ID e IDREFS.

```
<banco-2>
    <cuenta número_cuenta="C-401" titulares="C100 C102">
        <nombre_sucursal> Centro </nombre_sucursal>
        <saldo> 500 </saldo>
    </cuenta>
    <cuenta número_cuenta="C-402" titulares="C102 C101">
        <nombre_sucursal> Navacerrada </nombre_sucursal>
        <saldo> 900 </saldo>
    </cuenta>
    <cliente id_cliente="C100" cuentas="C-401">
        <nombre_cliente>Juncal</nombre_cliente>
        <calle_cliente> Mártires </calle_cliente>
        <ciudad_cliente> Melilla </ciudad_cliente>
    </cliente>
    <cliente id_cliente="C101" cuentas="C-402">
        <nombre_cliente>Loreto</nombre_cliente>
        <calle_cliente> Montaña </calle_cliente>
        <ciudad_cliente> Cáceres </ciudad_cliente>
    </cliente>
    <cliente id_cliente="C102" cuentas="C-401 C-402">
        <nombre_cliente>María</nombre_cliente>
        <calle_cliente> Eneas </calle_cliente>
        <ciudad_cliente> Alicante </ciudad_cliente>
    </cliente>
</banco-2>
```

Figura 10.9 Datos XML con los atributos ID y IDREF.

Los atributos ID e IDREF desempeñan la misma función que los mecanismos de referencia en las bases de datos orientadas a objetos y las bases de datos relacionales orientadas a objetos permitiendo la construcción de relaciones de datos complejas.

Las definiciones de tipos de documentos están fuertemente relacionadas con la herencia del formato del documento XML. Debido a esto no son adecuadas por varios motivos para servir como estructura de tipos de XML para aplicaciones de procesamiento de datos. No obstante, un tremendo número de formatos de intercambio de datos se están definiendo en términos de DTD, puesto que fueron parte original de la norma. Veamos algunas limitaciones de las DTDcomo mecanismo de esquema.

- No se puede declarar el tipo de cada elemento y de cada atributo de texto. Por ejemplo, el elemento saldo no se puede restringir para que sea un número positivo. La falta de tal restricción es problemática para las aplicaciones de procesamiento e intercambio de datos, las cuales deben contener el código para verificar los tipos de los elementos y atributos.

- Es difícil usar el mecanismo DTD para especificar conjuntos desordenados de subelementos. El orden es rara vez importante para el intercambio de datos (al contrario que en el diseño de documentos, donde es crucial). Aunque la combinación de la alternativa (la operación |) y las operaciones $*$ y $+$ como en la Figura 10.7 permite la especificación de colecciones desordenadas de marcas, es mucho más complicado especificar que cada marca pueda aparecer solamente una vez.

- Hay una falta de tipos en IDe IDREFS. Por ello no hay forma de especificar el tipo de elemento al cual se debería referir un atributo IDREF o IDREFS. En consecuencia, la DTD de la Figura 10.8 no evita que el atributo "titulares" de un elemento cuenta se refiera a otras cuentas, aunque esto no tenga sentido.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element nombre="banco" type="TipoBanco" />
<xs:element nombre="cuenta">
    <xs:complexType>
        <xs:sequence>
            <xs:element nombre="número_cuenta" type="xs:string"/>
            <xs:element nombre="nombre_sucursal" type="xs:string"/>
            <xs:element nombre="saldo" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element nombre="cliente">
            <xs:element nombre="número_cliente" type="xs:string"/>
            <xs:element nombre="calle_cliente" type="xs:string"/>
            <xs:element nombre="ciudad_cliente" type="xs:string"/>
</xs:element>
<xs:element nombre="impositor">
    <xs:complexType>
        <xs:sequence>
            <xs:element nombre="nombre_cliente" type="xs:string"/>
            <xs:element nombre="número_cuenta" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:complexType nombre="TipoBanco">
    <xs:sequence>
        <xs:element ref="cuenta" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="cliente" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="impositor" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
</xs:schema>
```

Figura 10.10 Versión XML Schema de la DTD de la Figura 10.7.

10.3.2 XML Schema

Un intento de reparar las deficiencias del mecanismo DTD produjo el desarrollo de un lenguaje de esquema más sofisticado, **XML Schema**. Se presenta aquí una visión general de XML Schema y se mencionan algunas áreas en las cuales mejora a las DTDs.

XML Schema define varios tipos predefinidos como string, integer, decimal date y boolean. Además permite tipos definidos por el usuario, que pueden ser tipos más simples con restricciones añadidas, o tipos complejos construidos con constructores como complexType y sequence.

La Figura 10.10 muestra cómo la DTD de la Figura 10.7 se puede representar mediante XML Schema. A continuación se describen las características de XML Schema que aparecen en esa figura.

Lo primero que es preciso resaltar es que las propias definiciones en XML Schema se especifican en sintaxis XML, usando varias etiquetas definidas en XML Schema. Para evitar conflictos con las etiquetas definidas por los usuarios, se antepone a la etiqueta XML Schema con la etiqueta de espacio de nombres "xs:"; este prefijo se asocia con el espacio de nombres de XML Schema mediante la especificación xmlns:xs en el elemento raíz:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

Obsérvese que se podría usar cualquier prefijo de espacio de nombres en lugar de xs; por tanto se podrían reemplazar todas las apariciones de "xs:" por "xsd:" sin cambiar el significado de la definición

del esquema. A todos los tipos definidos en XML Schema se les debe anteponer este prefijo de espacio de nombres.

El primer elemento es el elemento raíz banco, cuyo tipo es TipoBanco, que se declara posteriormente. En el ejemplo se definen después los tipos de los elementos cuenta, cliente e impositor. Obsérvese que cada uno de ellos se especifica con un elemento con la etiqueta xs:element, cuyo cuerpo contiene la definición de tipo.

El tipo de cuenta es compuesto, y se precisa como una secuencia de elementos número_cuenta, nombre_sucursal y saldo. Cualquier tipo que tenga atributos o subelementos anidados se puede especificar como tipo complejo.

Alternativamente se puede especificar que el tipo de un elemento sea un tipo predefinido con el atributo type; obsérvese el uso de los tipos de XML Schema xs:string y xs:decimal para restringir los tipos de los elementos de datos tales como número_cuenta.

Finalmente, el ejemplo define el tipo TipoBanco para contener ninguna o más apariciones tanto de cuenta como de cliente e impositor. Obsérvese el uso de ref para especificar la aparición de un elemento definido anteriormente. XML Schema puede definir el número mínimo y máximo de apariciones de subelementos mediante minOccurs y maxOccurs. El valor predeterminado para las apariciones máxima y mínima es 1, por lo que se tiene que especificar explícitamente para permitir cero o más cuentas, impositores y clientes.

Los atributos se especifican usando la etiqueta xs:attribute. Por ejemplo, se podría haber definido número_cuenta como atributo añadiendo

```
<xs:attribute nombre = "número_cuenta"/>
```

dentro de la declaración del elemento cuenta. Al añadir el atributo use = "required" a la especificación anterior se declara que el atributo se debe especificar, mientras que el valor predeterminado de use es optional. Las especificaciones de atributos pueden aparecer directamente dentro de la especificación complexType, aunque los elementos se aniden dentro de una especificación de secuencia.

Se puede usar el elemento xs:complexType para crear typos complejos con nombre; la sintaxis es la misma que la usada en el elemento xs:complexType de la Figura 10.10, salvo que se añade el atributo nombre = nombreTipo al elemento xs:complexType, donde nombreTipo es el nombre que se desea para el tipo. Se puede usar entonces el tipo nombrado para especificar el tipo de un elemento usando el atributo type, al igual que se han usado xs:decimal y xs:string en el ejemplo.

Además de definir tipos, un esquema relacional también permite la especificación de restricciones. XML Schema permite la especificación de claves y referencias a claves, que se corresponden con las definiciones de clave primaria y clave externa de SQL. En SQL una restricción de clave primaria o de unicidad asegura que los valores del atributo no se dupliquen en la relación. En el contexto de XML es necesario definir un ámbito en el que los valores sean únicos y formen una clave. selector es una expresión de ruta que define el ámbito de la restricción, y las declaraciones field especifican los elementos o atributos que forman la clave. Para especificar que los números de cuenta forman una clave de los elementos cuenta en el elemento raíz banco se podría añadir la siguiente especificación de restricción a la definición del esquema:

```
<xs:key nombre = "claveCuenta">
    <xs:selector xpath = "/banco/cuenta"/>
    <xs:field xpath = "número_cuenta"/>
</xs:key>
```

Se puede definir también la restricción de clave externa desde impositor hasta cuenta como sigue:

```
<xs:keyref nombre = "claveExtImpositorCuenta" refer="claveCuenta">
    <xs:selector xpath = "/banco/impositor"/>
    <xs:field xpath = "número_cuenta"/>
</xs:key>
```

Obsérvese que el atributo refer especifica el nombre de la declaración de clave a la que se hace referencia, mientras que la especificación field identifica los atributos que hacen la referencia.

XML Schema ofrece varias ventajas con respecto a las DTDs, y actualmente se usa mucho. Entre las ventajas que se han visto en los ejemplos anteriores se encuentran las siguientes:

- Permite que el texto que aparece en los elementos esté restringido a tipos específicos tales como tipos numéricos en formatos específicos o tipos más complejos como secuencias de elementos de otros tipos.
- Permite crear tipos definidos por el usuario.
- Permite restricciones de unicidad y de clave externa.
- Está integrado con espacios de nombres para permitir a diferentes partes de un documento adaptarse a esquemas diferentes.

Además de las características estudiadas, XML Schema soporta otras características que las DTDs no soportan, tales como:

- Permite restringir tipos para crear tipos especializados, por ejemplo especificando valores máximos y mínimos.
- Permite extender los tipos complejos mediante el uso de una forma de herencia.

Esta descripción de XML Schema es sólo una visión general; para obtener más información, consúltense las notas bibliográficas.

10.4 Consulta y transformación

Dado el creciente número de aplicaciones que usan XML para intercambiar, transmitir y almacenar datos, las herramientas para una gestión efectiva de datos XML están siendo cada vez más importantes. En particular las herramientas para consultar y transformar los datos XML son esenciales para extraer información de grandes cuerpos de datos XML y para convertir los datos entre distintas representaciones (esquemas) en XML. Al igual que la salida de una consulta relacional es una relación, la salida de una consulta XML puede ser un documento XML. Como resultado, la consulta y la transformación se pueden combinar en una única herramienta.

Los siguientes lenguajes proporcionan más capacidades de consulta y transformación:

- XPath es un lenguaje para expresiones de rutas de acceso y es realmente el fundamento de los otros dos lenguajes de consultas.
- XQuery es el lenguaje normalizado para la consulta de datos XML. Se ha modelado a partir de SQL, pero es significativamente diferente, ya que tiene que manejar datos XML anidados. También incorpora expresiones XPath.
- XSLT fue diseñado como lenguaje de transformación como parte del sistema de hojas de estilo XSL, que se usa para controlar el formato de los datos XML en HTML u otro lenguaje de impresión o visualización. Aunque diseñado para el formato, XSLT puede generar XML y expresar muchas consultas interesantes. Actualmente es el lenguaje más usado para la manipulación de datos XML, aunque XQuery es más adecuado para el tratamiento de bases de datos.

Se usa en todos estos lenguajes un **modelo de árbol** de datos XML. Cada documento XML se modela como un **árbol** con **nodos** correspondientes a los elementos y a los atributos. Los nodos de los elementos pueden tener nodos hijos, que pueden ser subelementos o atributos del elemento. De igual forma, cada nodo (ya sea de atributo o de elemento) distinto del elemento raíz tiene un nodo padre, que es un elemento. El orden de los elementos y de los atributos en el documento XML se modela ordenando los nodos hijos del árbol. Los términos padre, hijo, ascendiente, descendiente y hermano se interpretan en el modelo de árbol de datos XML.

El contenido textual de un elemento se puede modelar como un nodo de texto hijo del elemento. Los elementos que contienen texto descompuesto por subelementos intervinientes pueden tener varios nodos de texto hijo. Por ejemplo, un elemento que contenga "Éste es un <bold> gran </bold> libro" contiene un subelemento hijo correspondiente al elemento bold y dos nodos de texto hijos correspondientes a "Éste es un" y a "libro". Puesto que dichas estructuras no se usan habitualmente en los datos de las bases de datos, se asumirá que los elementos no contienen a la vez texto y subelementos.

10.4.1 XPath

XPath trata partes de los documentos XML mediante expresiones de rutas de acceso. Se puede ver el lenguaje como una extensión a las expresiones de rutas de acceso sencillas en las bases de datos orientadas a objetos y relacionales orientadas a objetos (véase el Apartado 9.6). La versión actual de XPath es la 2.0 y nuestra descripción se basa en ella.

Cada **expresión de ruta** de XPath es una secuencia de pasos de ubicación separados por "/" (en lugar de por el operador "." que separa los pasos en SQL:1999). El resultado de la expresión de ruta es un conjunto de nodos. Por ejemplo, en el documento de la Figura 10.9 la expresión XPath

```
/banco-2/cliente/nombre_cliente
```

devolverá estos elementos:

```
<nombre_cliente>Juncal</nombre_cliente>
<nombre_cliente>Loreto</nombre_cliente>
<nombre_cliente>María</nombre_cliente>
```

La expresión

```
/banco-2/cliente/nombre_cliente/text()
```

devolverá los mismos nombres, pero sin las etiquetas que los rodean.

Las expresiones de ruta se evalúan de izquierda a derecha. Al igual que en una jerarquía de directorios, la barra '/' inicial indica la raíz del documento (obsérvese que esto es una raíz abstracta "por encima de" <banco-2>, que es la etiqueta de documento).

Al evaluar una expresión de ruta, el resultado de la ruta en cualquier punto consiste en un conjunto ordenado de nodos del documento. Inicialmente el conjunto de elementos "actual" contiene un solo nodo, la raíz abstracta. Cuando el próximo paso de una expresión de ruta es un nombre de elemento, como cliente, el resultado del paso consiste en los nodos correspondientes a elementos del nombre especificado que son los hijos de los elementos en el conjunto actual de elementos. Estos nodos serán el conjunto actual de elementos para el siguiente paso de la evaluación de la expresión de ruta. Los nodos devueltos en cada paso aparecen en el mismo orden que en el documento.

Ya que varios hijos pueden tener el mismo nombre, el número de nodos en el conjunto puede aumentar o disminuir en cada paso. También se puede acceder a los valores de los atributos usando el símbolo "@". Por ejemplo, /banco-2/cuenta/@número_cuenta devuelve un conjunto con todos los valores de los atributos número_cuenta de los elementos cuenta. De forma predeterminada no se siguen los vínculos IDREF; posteriormente veremos cómo tratar con IDREF.

XPath soporta otras características:

- Los predicados de selección pueden seguir cualquier paso en una ruta y se escriben entre corchetes. Por ejemplo,

  ```
  /banco-2/cuenta[saldo > 400]
  ```

  devuelve los elementos cuenta con un valor saldo mayor que 400, mientras que

  ```
  /banco-2/cuenta[saldo > 400]/@número_cuenta
  ```

  devuelve los números de cuenta de dichas cuentas.

Se puede comprobar la existencia de un subelemento listándolo sin ningún operador de comparación; por ejemplo, si se quita simplemente "> 400", la expresión anterior devolvería los números de cuenta de todas las cuentas que tengan un subelemento saldo, independientemente de su valor.

- XPath proporciona varias funciones que se pueden usar como parte de predicados, incluidas la comprobación de la posición del nodo actual en el orden de los hermanos y la función de agregación count(), que cuenta el número de nodos coincidentes con la expresión a la que se aplica. Por ejemplo, la expresión de ruta

  /banco-2/cuenta[count(./cliente)>2]

  devuelve las cuentas con más de dos clientes. Se pueden usar las conectivas lógicas and y or en los predicados y la función not(. . .) para la negación.

- La función id("fu") devuelve el nodo (si existe) con un atributo del tipo ID cuyo valor sea "fu". La función id se puede incluso aplicar a conjuntos de referencias o incluso a cadenas que contengan referencias múltiples separadas por espacios vacíos, tales como IDREFS. Por ejemplo, la ruta

  /banco-2/cuenta/id(@titular)

  devuelve todos los clientes referenciados desde el atributo titular de los elementos cuenta.

- El operador | permite unir resultados de expresiones. Por ejemplo, si la DTD de banco-2 también contiene elementos para préstamos con atributo prestatario del tipo IDREFS para identificar el tomador de un préstamo, la expresión

  /banco-2/cuenta/id(@titular) | /banco-2/préstamo/id(@prestatario)

  proporciona los clientes con cuentas o préstamos. Sin embargo, el operador | no se puede anidar dentro de otros operadores. También merece la pena destacar que los nodos de la unión se devuelven en el orden en el que aparecen en el documento.

- Una expresión XPath puede saltar varios niveles de nodos mediante el uso de "//". Por ejemplo la expresión /banco-2//nombre_cliente encuentra cualquier elemento nombre_cliente en *cualquier lugar* por debajo del elemento /banco-2, independientemente del elemento en el que esté contenido y del número de niveles de inclusión presentes entre los elementos banco-2 y nombre_cliente. Este ejemplo ilustra la posibilidad de buscar los datos necesarios sin un conocimiento completo del esquema.

- Cada paso en la ruta no selecciona los hijos de los nodos del conjunto actual de nodos. En realidad esto es solamente una de las distintas direcciones que puede tomar un paso en la ruta tales como padres, hermanos, ascendientes y descendientes. Aquí se omiten los detalles, pero obsérvese que "//", descrito anteriormente, es una forma abreviada de especificar "todos los descendientes", mientras que ".." especifica el padre.

- La función predeterminada doc(nombre) devuelve la raíz de un documento con nombre; el nombre puede ser un nombre de archivo o un URL. La raíz devuelta pro la función se pueden usar en una expresión de ruta para acceder a los contenidos del documento. Por tanto, una expresión de ruta se puede aplicar en un documento especificado, en lugar de aplicarse al documento actual predeterminado.

  Por ejemplo, si los datos bancarios de nuestro ejemplo se contuviesen en el archivo "banco.xml", la siguiente expresión de ruta devolvería todas las cuentas del banco

  doc("banco.xml")/banco/cuenta

  La función collection(nombre) es similar a doc, pero devuelve una colección de documentos identificada por nombre.

10.4.2 XQuery

El consorcio W3C (World Wide Web Consortium) ha desarrollado XQuerycomo lenguaje de consulta normalizado de XML. Nuestra discusión aquí está basada en el último borrador de la norma del lenguaje disponible a principios de enero de 2005: aunque la norma final puede variar, se espera que las principales características que aquí se tratan no se modifiquen. El lenguaje XQuery procede de un lenguaje de consulta XML denominado Quilt; el propio Quilt incluía características de lenguajes anteriores, tales como XPath, estudiado en el Apartado 10.4, y otros dos lenguajes de consultas XML: XQL y XML-QL.

10.4.2.1 Expresiones FLWOR

Las consultas XQuery se basan en SQL, pero difieren significativamente de él. Se organizan en cinco secciones: **for**, **let**, **where**, **order by** y **return**. Se conocen como expresiones "FLWOR" (pronunciado "flauer", como "flor" en inglés), cuyas letras denotan las cinco secciones.

Una expresión FLWOR sencilla que devuelve los números de cuentas de las cuentas corrientes está basada en el documento XML de la Figura 10.9, que usa ID and IDREFS:

```
for $x in /banco-2/cuenta
let $numcuenta := $x/@número_cuenta
where $x/saldo > 400
return <número_cuenta> { $numcuenta } </número_cuenta>
```

La cláusula **for** es como la cláusula **from** de SQL y proporciona una serie de variables cuyos valores son los resultados de expresiones XPath. Cuando se especifica más de una variable, los resultados incluyen el producto cartesiano de los valores posibles que las variables pueden tomar, al igual que la cláusula **from** de SQL.

La cláusula **let** simplemente permite que se asigne el resultado de expresiones XPath al nombre de las variables por simplicidad de representación. La cláusula **where**, como la cláusula **where** de SQL, ejecuta comprobaciones adicionales sobre las tuplas reunidas de la cláusula **for**. La cláusula **order by**, al igual que la cláusula **order by** de SQL, permite la ordenación de la salida. Finalmente, la cláusula **return** permite la construcción de resultados en XML.

Una consulta FLWOR no necesita tener todas las cláusulas; por ejemplo, una consulta puede contener simplemente las cláusulas **for** y **return** y omitir **let**, **where** y **order by**. La consulta XQuery anterior no contiene ninguna cláusula **order by**. De hecho, dado que esta consulta es sencilla, se puede prescindir fácilmente de la cláusula **let**, y la variable $numcuenta de la cláusula **return** se podría remplazar por $x/@número_cuenta. Obsérvese además que, puesto que la cláusula **for** usa expresiones XPath, se pueden producir selecciones en la expresión XPath. Por ello, una consulta equivalente puede tener solamente cláusulas **for** y **return**:

```
for $x in /banco-2/cuenta[saldo > 400]
return <número_cuenta> { $x/@número_cuenta } </número_cuenta>
```

Sin embargo, la cláusula **let** simplifica las consultas complejas. Obsérvese también que las variables asignadas por la cláusula **let** puden contener secuencias con varios elementos o valores, si la expresión de ruta de la parte derecha los devuelve.

Obsérvese el uso de las llaves ("{}") en la cláusula **return**. Cuando XQuery encuentra un elemento como <número_cuenta> que inicia una expresión, trata su contenido como texto XML normal, excepto las partes encerradas entre llaves, que se evalúan como expresiones. Así, si se hubiesen omitido las llaves en la anterior cláusula **return**, el resultado contendría varias copias de la cadena "$x/@número_cuenta", cada una de ellas encerrada en una etiqueta número_cuenta. Los contenidos dentro de las llaves son, no obstante, tratados como expresiones a evaluar. Este convenio se aplica incluso con las llaves entre comillas. Así, se podría modificar la consulta anterior para devolver un elemento con una etiqueta cuenta, con el número de cuenta como atributo, reemplazando la cláusula **return** por:

```
return <cuenta número_cuenta="{$x/@número_cuenta}" />
```

XQuery proporciona otra forma de construir elementos usando los constructores **element** y **attribute**. Por ejemplo, si la cláusula **return** de la consulta anterior se reemplaza por la siguiente cláusula **return**, la consulta devolvería elementos cuenta con número_cuenta and nombre_sucursal como atributos y saldo como subelemento.

```
return element cuenta {
        attribute número_cuenta {$x/@número_cuenta},
        attribute nombre_sucursal {$x/nombre_sucursal},
        element saldo {$x/saldo}
}
```

Obsérvese que, al igual que sucedía antes, las llaves son necesarias para tratar una cadena como una expresión a evaluar.

10.4.2.2 Reuniones

Las reuniones se especifican en XQuery de forma parecida a SQL. La reunión de los elementos impositor, cuenta y cliente de la Figura 10.1, que se escribe en XSLT en el Apartado 10.4.3, se puede escribir en XQuery de la siguiente forma:

```
for $a in /banco/cuenta,
        $c in /banco/cliente,
        $i in /banco/impositor
where $a/número_cuenta = $i/número_cuenta
        and $c/nombre_cliente = $i/nombre_cliente
return <cuenta_cliente> { $c $a } </cuenta_cliente>
```

La misma consulta se puede expresar con las selecciones especificadas como selecciones XPath:

```
for $a in /banco/cuenta,
        $c in /banco/cliente,
        $i in /banco/impositor[número_cuenta = $a/número_cuenta
                and nombre_cliente = $c/nombre_cliente]
return <cuenta_cliente> { $c $a } </cuenta_cliente>
```

Las expresiones de ruta en XQuery son las misma expresiones de ruta en XPath2.0. Las expresiones de ruta pueden devolver un único valor o elemento, o una secuencia de ellos. A falta de información de esquema puede no ser posible inferir si una expresión de ruta devuelve un único valor o una secuencia de valores. Tales expresiones pueden participar en operaciones de comparación tales como $=, <$ y $>=$.

XQuery tiene un definición interesante de las operaciones de comparación sobre secuencias. Por ejemplo, la expresión $x/saldo > 400$ tendría la interpretación usual si el resultado de $x/saldo fuese un único valor, pero si el resultado es una secuencia que contiene varios valores, la expresión se evalúa a cierto si al menos uno de los valores es mayor que 400. Análogamente, la expresión $x/saldo = $y/saldo es cierta si alguno de los valores devueltos por la primera expresión es igual a cualquier otro valor de la segunda. Si este comportamiento no es apropiado, se pueden usar las operaciones eq, ne, lt, gt, le, ge en su lugar, las cuales generan un error si cualquiera de sus entradas es una secuencia de varios valores.

10.4.2.3 Consultas anidadas

Las expresiones FLWOR de XQuery se pueden anidar en la cláusula **return** con el fin de generar anidamientos de elementos que no aparecen en el documento origen. Esta característica es similar a las subconsultas anidadas en la cláusula **from** de las consultas SQL del Apartado 9.5.3. Por ejemplo, la estructura XML mostrada en la Figura 10.4, con los elementos de la cuenta anidados en elementos cliente, se puede generar a partir de la estructura en la Figura 10.1 mediante esta consulta:

```
<banco-1> {
    for $c in /banco/cliente
    return
        <cliente>
            { $c/* }
            { for $i in /banco/impositor[nombre_cliente = $c/nombre_cliente],
                  $a in /banco/cuenta[número_cuenta=$i/número_cuenta]
              return $a }
        </cliente>
} </banco-1>
```

La consulta también introduce la sintaxis $c/*, que se refiere a todos los hijos del nodo (o secuencia de nodos), ligados a la variable $c. De forma similar, $c/text() proporciona el contenido textual de un elemento, sin las etiquetas.

XQuery proporciona funciones de agregado tales como sum() y count() que se pueden aplicar a secuencias de elementos o valores. La función distinct-values() aplicada sobre una secuencia devuelve una secuencia sin duplicados. La secuencia (colección) de valores devueltos por una expresión de ruta puede tener algunos valores repetidos porque se repiten en el documento, aunque un resultado de una expresión XPath pueda contener a lo suma una aparición de cada nodo en el documento. XQuery soporta muchas otras funciones que son comunes a XPath 2.0 y XQuery, y se pueden usar en cualquier expresión de ruta de XPath.

Para evitar conflictos, las funciones se asocian con un espacio de nombres

http://www.w3.org/2004/10/xpath-functions

que tiene el prefijo predeterminado fn para el espacio de nombres. Así, estas funciones se pueden reconocer sin ambigüedades como fn:sum o fn:count.

Aunque XQuery no proporciona un constructor **group by**, las consultas de agregación se pueden escribir usando funciones de agregado sobre expresiones de ruta o FLWOR anidadas dentro de la cláusula **return**. Por ejemplo, la siguiente consulta sobre el esquema XML banco calcula el saldo total de todas las cuentas de cada cliente.

```
for $c in /banco/cliente
return
    <saldo-total-cliente>
        <nombre_cliente> { $c/nombre_cliente } </nombre_cliente>
        <saldo_total> { fn:sum(
            for$i in /banco/impositor[nombre_cliente = $c/nombre_cliente],
                $a in /banco/cuenta[número_cuenta = $d/número_cuenta]
            return $a/saldo
        ) } </saldo_total>
    </saldo-total-cliente>
```

10.4.2.4 Ordenación de resultados

En XQuery los resultados se pueden ordenar si se incluye una cláusula **order by**. Por ejemplo, esta consulta tiene como salida todos los elementos cliente ordenados por el subelemento nombre_cliente:

```
for $c in /banco/cliente,
order by $c/nombre_cliente
return <cliente> { $c/* } </cliente>
```

Para ordenar de forma decreciente podemos usar **order by** nombre_cliente **descending**.

La ordenación se puede realizar en varios niveles de anidamiento. Por ejemplo, se puede obtener una representación anidada de la información bancaria ordenada según el nombre del cliente, con las cuentas de cada cliente ordenadas según el número de cuenta, como sigue:

```
<banco-1> {
    for $c in /banco/cliente
    order by $c/nombre_cliente
    return
        <cliente>
            { $c/* }
            { for $i in /banco/impositor[nombre_cliente = $c/nombre_cliente],
                  $a in /banco/cuenta[número_cuenta = $d/número_cuenta]
            order by $a/número_cuenta
            return <cuenta> { $a/* } </cuenta> }
        </cliente>
} </banco-1>
```

10.4.2.5 Funciones y tipos

XQuery proporciona una serie de funciones predefinidas, como funciones numéricas y de comparación de cadenas y funciones de manipulación. Además, XQuery soporta funciones definidas por el usuario. La siguiente función definida por el usuario devuelve una lista de todos los saldos de un cliente con un nombre especificado:

```
define function saldos(xs:string $c) as xs:decimal* {
    for $i in /banco/impositor[nombre_cliente = $c],
        $a in /banco/cuenta[número_cuenta = $d/número_cuenta]
    return $a/saldo
}
```

La especificación de tipo de los argumentos de la función y de los valores devueltos es opcional. XQuery usa el sistema de tipos de XML Schema. El prefijo xs: de espacio de nombres usado en el ejemplo anterior está asociado con el espacio de nombres de XML Schema de forma predefinida en XQuery.

Se puede añadir a los tipos * como sufijo para indicar una secuencia de valores de ese tipo; por ejemplo, la definición de la función saldos especifica su valor devuelto como una secuencia de valores numéricos. Los tipos se pueden especificar parcialmente; por ejemplo, el tipo element permite elementos con cualquier etiqueta, mientras que element(cuenta) permite elementos con la etiqueta cuenta.

XQuery realiza convierte los tipos automáticamente siempre que sea necesario. Por ejemplo, si un valor numérico representado por una cadena se compara con un tipo numérico, la conversión de tipo de cadena a número se hace automáticamente. Cuando se pasa un elemento a una función que espera una cadena, la conversión a cadena se hace concatenando todos los valores de texto contenidos (anidados) dentro del elemento. Así, la función contains(a,b), que comprueba si la cadena a contiene a la cadena b, se puede usar con un elemento en su primer argumento, en cuyo caso comprueba si el elemento a contiene la cadena b anidada en cualquier lugar dentro de él. XQuery también proporciona funciones para convertir tipos. Por ejemplo, number(x) convierte una cadena en un número.

10.4.2.6 Otras características

XQuery ofrece una gran variedad de características adicionales tales como constructores if-then-else, que se pueden usar con cláusulas **return** y la cuantificación existencial y universal, que se pueden usar en predicados en cláusulas **where**. Por ejemplo, la cuantificación existencial se puede expresar en la cláusula **where** usando

```
some $e in ruta satisfies P
```

donde ruta es una expresión de ruta y P es un predicado que puede usar $e. La cuantificación universal se puede expresar usando **every** en lugar de **some**.

Como se puede ver en la descripción anterior, XQuery con XPath es un lenguaje bastante complejo, y debe manejar datos de estructura compleja. Aunque hace varios años que se definió (en un borrador),

```
<xsl:template match="/banco-2/cliente">
    <cliente>
    <xsl:value-of select="nombre_cliente"/>
    </cliente>
</xsl:template>
<xsl:template match="*"/>
```

Figura 10.11 Uso de XSLT para convertir los resultados en nuevos elementos XML.

muchas implementaciones o bien implementan un subconjunto de XQuery o bien son ineficientes en grandes conjuntos de datos.

La norma **XQJ** proporciona una interfaz de programación de aplicaciones (API) para ejecutar consultas XQuery en un sistema de bases de datos XML y para devolver los resultados XML. Su funcionalidad es similar a la API JDBC.

10.4.3 XSLT**

Una **hoja de estilo** es una representación de las opciones de formato para un documento, normalmente almacenado fuera del documento mismo, por lo que el formato está separado del contenido. Por ejemplo, una hoja de estilo para HTML puede especificar la fuente a usar en todas la cabeceras y, por ello, reemplaza un gran número de declaraciones de fuente en la página HTML. **XML XSL (Stylesheet Language)**, el lenguaje de hojas de estilo XML, estaba originalmente diseñado para generar HTML a partir de XML y es por ello una extensión lógica de hojas de estilo HTML. El lenguaje incluye un mecanismo de transformación de propósito general, denominado **XSLT** (**XSL Transformations**, transformaciones XSL), que se puede usar para transformar un documento XML en otro documento XML, o a otros formatos como HTML[1]. Las transformaciones XSLT son bastante potentes y en realidad XSLT puede incluso actuar como lenguaje de consulta.

Las transformaciones XSLT se expresan como una serie de reglas recursivas, denominadas **plantillas**. En su forma básica las plantillas permiten la selección de nodos en un árbol XML mediante una expresión XPath. Sin embargo, las plantillas también pueden generar contenido XML nuevo de forma que esa selección y generación de contenido se pueda mezclar de formas naturales y potentes. Aunque XSLT se puede usar como lenguaje de consulta, su sintaxis y semántica son bastante distintas de las de SQL.

Una plantilla sencilla para XSLT consiste en una parte de **coincidencia** (match) y una parte de **selección** (select). Consideremos el siguiente código XSLT:

```
<xsl:template match="/banco-2/cliente">
    <xsl:value-of select="nombre_cliente"/>
</xsl:template>
<xsl:template match="*"/>
```

La instrucción xsl:template match contiene una expresión XPath que selecciona uno o más nodos. La primera plantilla busca coincidencias de elementos cliente que aparecen como hijos del elemento raíz banco-2. La instrucción xsl:value-of encerrada en la instrucción de coincidencia devuelve valores de los nodos en el resultado de la expresión XPath. La primera plantilla devuelve el valor del subelemento nombre_cliente; obsérvese que el valor no contiene la etiqueta de elemento.

Obsérvese que la segunda plantilla coincide con todos los nodos. Esto es necesario debido a que el comportamiento predeterminado de XSLT sobre los elementos del documento de entrada que no coinciden con ninguna plantilla es copiar sus atributos y contenidos textuales al documento de salida, y aplicar recursivamente plantillas a sus subelementos.

Cualquier texto o etiqueta de la hoja de estilo XSLT que no esté en el espacio de nombres xsl se copia sin cambiar en la salida. La Figura 10.11 muestra cómo usar esta característica para hacer que cada nombre de cliente del ejemplo aparezca como un subelemento del elemento "<cliente>", colocando la

1. La norma XSL consiste ahora en XSLT más una norma para especificar las características de formato tales como fuentes, márgenes de página y tablas. El formato no es relevante desde el punto de vista de las bases de datos, por lo que no se tratará aquí.

```
<xsl:template match="/banco">
    <clientes>
    <xsl:apply-templates/>
    </clientes>
</xsl:template>
<xsl:template match="/cliente">
    <cliente>
    <xsl:value-of select="nombre_cliente"/>
    </cliente>
</xsl:template>
<xsl:template match="*"/>
```

Figura 10.12 Aplicación recursiva de reglas.

instrucción xsl:value-of entre <cliente> y </cliente>. La creación de un atributo como id_cliente en el elemento cliente generado es difícil y requiere el uso de xsl:attribute; véase un manual de XSLT para más detalles.

La **recursividad estructural** es una parte clave de XSLT. Hay que recordar que los elementos y subelementos naturalmente forman una estructura en árbol. La idea de la recursividad estructural es la siguiente: cuando una plantilla coincide con un elemento en la estructura de árbol XSLT puede usar la recursividad estructural para aplicar las reglas de la plantilla recursivamente a los subárboles en lugar de simplemente devolver un valor. Aplica las reglas recursivamente mediante la directiva xsl:apply-templates, que aparece dentro de otras plantillas.

Por ejemplo, los resultados de nuestra consulta anterior se pueden ubicar en un elemento <clientes> mediante la adición de una regla usando xsl:apply-templates, como en la Figura 10.12. La nueva regla coincide con la etiqueta externa "banco" y construye un documento resultado mediante la aplicación de otras plantillas a los subárboles que aparecen en el elemento banco, pero envolviendo los resultados en el elemento <clientes> </clientes> dado. Sin la recursividad forzada por la cláusula <xsl:apply-templates/> la plantilla devolvería <clientes> </clientes> y entonces aplicaría otras plantillas a los subelementos.

De hecho, la recursividad estructural es crítica para construir documentos XML bien formados, puesto que los documentos XML deben tener un único elemento de nivel superior que contenga el resto de elementos del documento.

XSLT proporciona una característica denominada **keys** (claves) que permite la búsqueda de elementos mediante el uso de valores de subelementos o atributos; los objetivos son similares a los de la función id() de XPath, pero las claves de XSLT permiten usar atributos distintos a los atributos ID. Las claves se definen mediante una directiva xsl:key, la cual tiene tres partes, por ejemplo:

```
<xsl:key name="numcuenta" match="cuenta" use="número_cuenta"/>
```

El atributo nombre se usa para distinguir claves distintas. El atributo match especifica los nodos a los que se aplica la clave. Finalmente, el atributo use especifica la expresión a usar como el valor de la clave. Obsérvese que la expresión no tiene que ser única para un elemento; esto es, más de un elemento puede tener el mismo valor de expresión. En el ejemplo la clave denominada numcuenta especifica que el subelemento número_cuenta de cuenta se debería usar como una clave para esa cuenta.

Las claves se pueden usar en plantillas como parte de cualquier patrón mediante la función key. Esta función toma el nombre de la clave y un valor y devuelve el conjunto de nodos que coinciden con ese valor. Por ello, el nodo XML para la cuenta "C-401" se puede referenciar como key("numcuenta", "C-401").

Las claves se pueden usar para implementar algunos tipos de reuniones, como en la Figura 10.13. El código de la figura se puede aplicar a datos XML en el formato de la Figura 10.1. Aquí la función key reúne los elementos impositor con los elementos coincidentes cliente y cuenta. El resultado de la consulta consiste en pares de elementos cliente y cuenta encerrados entre elementos cuenta_cliente.

XSLT permite ordenar los nodos. Un ejemplo sencillo muestra cómo se usa xsl:sort en la hoja de estilo para devolver los elementos cliente ordenados por nombre:

```
<xsl:key name="numcuenta" match="cuenta"use="número_cuenta"/>
<xsl:key name="numcliente" match="cliente" use="nombre_cliente"/>
<xsl:template match="impositor">
    <cuenta_cliente>
    <xsl:value-of select=key("numcliente", "nombre_cliente")/>
    <xsl:value-of select=key("numcuenta", "número_cuenta")/>
    </cuenta_cliente>
</xsl:template>
<xsl:template match="*"/>
```

Figura 10.13 Reuniones en XSLT.

```
<xsl:template match="/banco">
    <xsl:apply-templates select="cliente">
    <xsl:sort select="nombre_cliente"/>
    </xsl:apply-templates>
</xsl:template>
<xsl:template match="cliente">
    <cliente>
        <xsl:value-of select="nombre_cliente"/>
        <xsl:value-of select="calle_cliente"/>
        <xsl:value-of select="ciudad_cliente"/>
    </cliente>
</xsl:template>
<xsl:template match="*"/>
```

Aquí xsl:apply-template tiene un atributo select que lo restringe para que sólo se aplique a los subelementos cliente. La directiva xsl:sort en el elemento xsl:apply-template hace que los nodos se ordenen *antes* de ser procesados por el siguiente conjunto de plantillas. Existen opciones para permitir la ordenación sobre varios subelementos/attributes, por valor numérico y en orden descendente.

10.5 La interfaz de programación de aplicaciones de XML

Debido a la gran aceptación de XML como una representación de datos y formato de intercambio hay gran cantidad de herramientas de software disponibles para la manipulación de datos XML. Hay dos modelos para la manipulación mediante programación de XML, cada uno disponible para su uso con una varios lenguajes de programación populares. Ambas APIs se pueden usar para analizar documentos XML y crear su representación en memoria. Se emplean para aplicaciones que manejan documentos XML individuales. Obsérvese, sin embargo, que no son adecuadas para consultar grandes colecciones de datos XML; los mecanismos de consulta declarativos tales como XPath y XQuery son mucho más adecuados para esta tarea.

Una de las APIs estándar para la manipulación de XML se basa en el *modelo de objetos documento* (Document Object Model, DOM), que trata el contenido XML como un árbol, con cada elemento representado por un nodo, denominado DOMNode. Los programas pueden acceder a partes del documento mediante navegación comenzando con la raíz.

Se encuentran disponibles bibliotecas DOM para la mayor parte de los lenguajes de programación más comunes y están incluso presentes en los exploradores Web, donde se pueden usar para manipular el documento mostrado al usuario. Aquí se explican algunas de las interfaces y métodos de la API DOM de Java, para mostrar cómo puede ser un DOM.

- La API DOM de Java proporciona una interfaz denominada Node e interfaces Element y Attribute, las cuales reciben la herencia de la interfaz Node.
- La interfaz Node proporciona métodos tales como getParentNode(), getFirstChild() y getNextSibling() para navegar por el árbol DOM comenzando por el nodo raíz.

- Se puede acceder a los subelementos de un elemento mediante el nombre getElementsByTagNombre(nombre), que devuelve una lista de todos los elementos hijo con un nombre de etiqueta especificado; se puede acceder a cada miembro de la lista mediante el método item(i), que devuelve el i-ésimo elemento de la lista.
- Se puede acceder a los valores de atributo de una elemento mediante el nombre, usando el método getAttribute(nombre).
- El valor de texto de un elemento se modela como nodo Text, que es un hijo del nodo elemento; un nodo elemento sin subelemento tiene solamente un nodo hijo. El método getData() del nodo Text devuelve el contenido de texto.

DOM también proporciona una serie de funciones para actualizar el documento mediante la adición y el borrado de hijos elemento y atributo, el establecimiento de valores de nodos, etc.

Se necesitan muchos más detalles para escribir un programa DOM real; véanse las notas bibliográficas para obtener referencias con más información.

DOM se puede utilizar para acceder a los datos XML almacenados en las bases de datos y se puede construir una base de datos XML con DOM como interfaz principal para acceder y modificar los datos. Sin embargo, la interfaz DOM no soporta ninguna forma de consulta declarativa.

La segunda interfaz de programación empleada más habitualmente, *API simple para XML* (Simple API for XML, SAX) es un modelo de *eventos* diseñado para proporcionar una interfaz común entre analizadores y aplicaciones. Esta API está construida bajo la noción de *manejadores de eventos*, que consisten en funciones especificadas por el usuario asociadas con eventos de análisis. Los eventos de análisis corresponden con el reconocimiento de partes de un documento; por ejemplo, cuando se encuentra la etiqueta de inicio de un elemento se genera un evento y cuando se encuentra su etiqueta de finalización se genera otro. Las piezas de un documento siempre se encuentran en orden desde el inicio al final.

El desarrollador de aplicaciones SAX crea funciones controladoras para cada evento y las registra. Cuando el analizador SAX lee un documento, cuando ocurre cada evento, se llama a la función controladora con parámetros que describen el evento (como la etiqueta del elemento o los contenidos textuales). Las funciones controladoras realizan entonces su función. Por ejemplo, para construir un árbol que represente los datos XML, las funciones controladoras para un evento de inicio de un atributo o elemento establecerían el nuevo elemento como el nodo donde otros nodos hijos deben adjuntarse. El elemento y evento correspondientes establecerían el padre del nodo como el nodo actual donde se pueden adjuntar más nodos hijos.

SAX requiere generalmente más esfuerzo de programación que DOM, pero ayuda a evitar la sobrecarga de la creación de un árbol DOM en situaciones donde la aplicación necesita su propia representación de datos. Si se usase DOM en tales aplicaciones, habría unas sobrecargas innecesarias de espacio y tiempo para la construcción del árbol DOM.

10.6 Almacenamiento de datos XML

Muchas aplicaciones requieren el almacenamiento de datos XML. Una forma de almacenar datos XML es como documentos en un sistema de archivos y otra es construir una base de datos de propósito especial para almacenar datos XML. Otro enfoque es convertir los datos XML a una representación relacional y almacenarla en la base de datos relacional. Hay varias alternativas para almacenar datos XML que se muestran brevemente en este apartado.

10.6.1 Almacenamiento de datos no relacionales

Existen varias alternativas para almacenar datos XML en sistemas de almacenamiento de datos no relacionales:

- **Almacenamiento en archivos planos.** Puesto que XML es principalmente un formato de archivo, un mecanismo de almacenamiento natural es simplemente un archivo plano. Este enfoque tiene muchos de los inconvenientes mostrados en el Capítulo 1 sobre el uso de sistemas de archivos como base para las aplicaciones de bases de datos. En particular hay una carencia de aislamiento

de datos, comprobaciones de integridad, atomicidad, acceso concurrente y seguridad. Sin embargo, la amplia disponibilidad de herramientas XML que funcionan sobre archivos de datos hace relativamente sencillo el acceso y consulta de datos XML almacenados en archivos. Por ello, este formato de almacenamiento puede ser suficiente para algunas aplicaciones.

- **Creación de bases de datos XML.** Las bases de datos XML son bases de datos que usan XML como modelo de datos básico. Las primeras bases de datos XML implementaban el modelo de objetos documento sobre bases de datos orientadas a objetos basadas en C++. Esto permite reusar gran parte de la infraestructura de bases de datos orientada a objetos mientras se usa una interfaz XML estándar. La adición de un XQuery o de otros lenguajes de consultas XML permite consultas declarativas. Otras implementaciones han construido toda la infraestructura de almacenamiento y de consulta XML sobre un gestor de almacenamiento que proporciona soporte transaccional.

Aunque se han diseñado varios sistemas de bases de datos específicamente para almacenar datos XML, el desarrollo de un sistema gestor de bases de datos desde cero es una tarea muy compleja. Se debería dar soporte no sólo al almacenamiento y consulta de datos XML, sino también a otras características tales como transacciones, seguridad, soporte de acceso a datos desde clientes y capacidades de administración. Parece entonces razonable usar un sistema de bases de datos existente para proporcionar estas características e implementar el almacenamiento y la consulta XML, bien sobre la abstracción relacional, bien como una capa paralela a ella. Estas alternativas se estudian en el Apartado 10.6.2.

10.6.2 Bases de datos relacionales

Puesto que las bases de datos relacionales se usan ampliamente en aplicaciones existentes, es una gran ventaja almacenar datos XML en bases de datos relacionales de forma que se pueda acceder a los datos desde aplicaciones existentes.

La conversión de datos XML a una forma relacional es normalmente muy sencilla si los datos se han generado en un principio desde un esquema relacional y XML se usó simplemente como un formato de intercambio de datos para datos relacionales. Sin embargo, hay muchas aplicaciones donde los datos XML no se han generado desde un esquema relacional y la traducción de éstos a una forma relacional de almacenamiento puede no ser tan sencilla. En particular los elementos anidados y los elementos que se repiten (correspondientes a atributos con valores de conjunto) complican el almacenamiento de los datos XML en un formato relacional. Se encuentran disponibles diversas alternativas que se describen a continuación.

10.6.2.1 Almacenamiento con cadenas

Los pequeños documentos XML se pueden almacenar como cadenas (**clob**) en tuplas de una base de datos relacional. Los documentos XML grandes cuyo elemento de nivel superior tenga muchos hijos se puede tratar almacenando cada elemento hijo como una cadena en una tupla separada de la base de datos. Por ejemplo, los datos XML de la Figura 10.1 se podrían almacenar como un conjunto de tuplas en una relación *elementos*(*datos*), donde el atributo *datos* de cada tupla almacena un elemento XML (cuenta, cliente o impositor) en forma de cadena.

Aunque esta representación es fácil de usar, el sistema de la base de datos no conoce el esquema de los elementos almacenados. Como resultado no es posible consultar los datos directamente. En realidad no es siquiera posible implementar selecciones sencillas tales como buscar todos los elementos cuenta o encontrar el elemento cuenta cuyo número de cuenta sea C-401, sin explorar todas las tuplas de la relación y examinar los contenidos de la cadena.

Una solución parcial a este problema es almacenar distintos tipos de elementos en relaciones diferentes y también almacenar los valores de algunos elementos críticos como atributos de la relación que permite la indexación. Así, en nuestro ejemplo, las relaciones serían *elementos_cuenta*, *elementos_cliente* y *elementos_impositor*, cada una con un atributo *datos*. Cada relación puede tener atributos extra para almacenar los valores de algunos subelementos tales como *número_cuenta* o *nombre_cliente*. Por ello se puede responder eficientemente con esta representación a una consulta que requiera los elementos cuenta con un número de cuenta especificado. Tal enfoque depende del tipo de información sobre los datos XML, tales como la DTD de los datos.

Algunos sistemas de bases de datos, tales como Oracle, soportan **índices de función**, que pueden ayudar a evitar la duplicación de atributos entre la cadena XML y los atributos de la relación. A diferencia de los índices normales, que se construyen sobre los valores de atributos, los índices de función se pueden construir sobre el resultado de aplicar funciones definidas por el usuario a las tuplas. Por ejemplo, se puede construir un índice de función sobre una función definida por el usuario que devuelve el valor del subelemento número_cuenta de la cadena XML en una tupla. El índice se puede entonces usar de la misma forma que un índice sobre un atributo *número_cuenta*.

Estos enfoques tienen el inconveniente de que una gran parte de la información XML se almacena en cadenas. Es posible almacenar toda la información en relaciones en alguna de las formas que se examinan a continuación.

10.6.2.2 Representación con árboles

Cualquier dato XML se puede modelar como árbol y almacenar mediante el uso de un par de relaciones:

nodos(*id*, *tipo*, *etiqueta*, *valor*)
hijo(*id_hijo*, *id_padre*)

A cada elemento y atributo de los datos XML se le proporciona un identificador único. Una tupla insertada en la relación *nodos* para cada elemento y atributo con su identificador (*id*), su tipo (atributo o elemento), el nombre del elemento o atributo (*etiqueta*) y el valor textual del elemento o atributo (*valor*). La relación *hijo* se usa para guardar el elemento padre de cada elemento y atributo. Si la información de orden de los elementos y atributos se debe preservar, se puede agregar un atributo adicional, *posición*, a la relación *hijo* para indicar su posición relativa entre los hijos del padre. Como ejercicio se pueden representar los datos XML de la Figura 10.1 mediante el uso de esta técnica.

Esta representación tiene la ventaja de que toda la información XML se puede representar directamente de forma relacional y se pueden trasladar muchas consultas XML a consultas relacionales y ejecutar dentro del sistema de la base de datos. Sin embargo, tiene el inconveniente de que cada elemento se divide en muchas piezas y se necesita gran número de combinaciones para volver a ensamblar los subelementos en un solo elemento.

10.6.2.3 Representación con relaciones

Según este enfoque, los elementos XML cuyo esquema es conocido se representan mediante relaciones y atributos. Los elementos cuyo esquema es desconocido se almacenan como cadenas o como una representación en árbol.

Se crea una relación para cada tipo de elemento (incluyendo subelementos) cuyo esquema sea conocido y cuyo tipo sea complejo (es decir, que contenga atributos o subelementos). Los atributos de la relación se definen como sigue:

- Todos los atributos de estos elementos se almacenan como atributos de tipo cadena de la relación.
- Si un subelemento del elemento es de tipo simple (es decir, no contiene atributos o subelementos), se añade un atributo a la relación para representarlo. El tipo predeterminado del atributo es cadena, pero si el subelemento tuviese el tipo XML Schema se podría usar el tipo SQL correspondiente.

  Por ejemplo, el subelemento *número_cuenta* del elemento *cuenta* se convierte en atributo de la relación *cuenta*.
- En otro caso, se crea una relación correspondiente al subelemento (usando recursivamente las mismas reglas en sus subelementos). Además:
  - Se añade un atributo identificador a las relaciones que representen al elemento (el atributo identificador se añade sólo una vez aunque el elemento tenga varios subelementos).
  - Se añade el atributo *id_padre* a la relación que representa al subelemento, almacenando el identificador de su elemento padre.
  - Si es necesario conservar el orden, se añade el atributo *posición* a la relación que representa al subelemento.

Obsérvese que cuando se aplica este enfoque a los elementos bajo el elemento raíz de la DTD de los datos de la Figura 10.1 se vuelve al esquema relacional original que se ha usado en capítulos anteriores.

Este enfoque admite diferentes variantes. Por ejemplo, las relaciones correspondientes a subelementos que pueden aparecer una vez a lo sumo se pueden "aplanar" en la relación padre trasladando todos los atributos a ella. Las notas bibliográficas proporcionan referencias a enfoques diferentes para representar datos XML como relaciones.

10.6.2.4 Publicación y fragmentación de datos XML

Cuando se usa XML para intercambiar datos entre aplicaciones de negocio, los datos se originan muy frecuentemente en bases de datos relacionales. Los datos en las bases de datos relacionales deben ser *publicados*, esto es, convertidos a formato XML para su exportación a otras aplicaciones. Los datos de entrada deben ser *fragmentados*, es decir, convertidos de XML a un formato normalizado de relación y almacenado en una base de datos relacional. Aunque el código de la aplicación pueda ejecutar las operaciones de publicación y fragmentación, las operaciones son tan comunes que la conversión se debería realizar de forma automática, sin escribir ningún código en la aplicación, siempre que sea posible. Por ello, los fabricantes de bases de datos están trabajando para dar *capacidades XML* a sus productos de bases de datos.

Una base de datos con capacidades XML permite una correspondencia automática de su modelo relacional interno con XML. Esta correspondencia puede ser sencilla o compleja. Una correspondencia sencilla podría asignar un elemento XML a cada fila de una tabla y hacer de cada columna un subelemento del elemento XML. El esquema XML de la Figura 10.1 se puede crear de una representación relacional de información bancaria usando dicha correspondencia. Esta correspondencia es sencilla de generar automáticamente. Esta vista XML de los datos relacionales se pueden tratar como un documento XML *virtual*, y las consultas XML se pueden ejecutar en el documento XML *virtual*. Una correspondencia más complicada permitiría que se crearan estructuras anidadas. Las extensiones de SQL con consultas anidadas en la cláusula **select** se han desarrollado para permitir una creación fácil de salida XML anidado. Estas extensiones se describen en el Apartado 10.6.3.

También se han definido correspondencias para fragmentar datos XML en una representación relacional. Para los datos XML creados de una representación relacional la correspondencia requerida para fragmentar los datos es sencilla e inversa a la correspondencia usada para publicar los datos. Para el caso general se puede generar la correspondencia como se describe en el Apartado 10.6.2.3.

10.6.2.5 Almacenamiento nativo en bases de datos relacionales

Recientemente, las bases de datos han comenzado a dar soporte al **almacenamiento nativo** de XML. Estos sistemas almacenan datos XML como cadenas o en representaciones binarias más eficientes, sin convertir los datos a la forma relacional. Se introduce el nuevo tipo de datos **xml** para representar datos XML, aunque los tipos CLOB y BLOB pueden proporcionar el mecanismo subyacente de almacenamiento. Se incluyen lenguajes de consultas XML tales como XPath y XQuery para las consultas de datos XML.

Se puede utilizar una relación con un atributo de tipo **xml** para almacenar una colección de documentos XML; cada documento se almacena como un valor de tipo **xml** en una tupla diferente. Se crean índices ad hoc para indexar los datos XML.

Varios sistemas de bases de datos proporcionan soporte nativo para datos XML. Proporcionan un tipo de datos **xml** y permiten que las consultas XQuery se incorporen dentro de las consultas SQL. Cada consulta XQuery se puede ejecutar en un solo documento XML y se puede incorporar dentro de una consulta SQL para permitir que se ejecute en cada colección de documentos, con cada documento almacenado en una tupla diferente. Por ejemplo, véase el Apartado 29.11 para más detalles sobre el soporte nativo de XML en SQL Server 2005 de Microsoft.

10.6.3 SQL/XML

La norma SQL/XML recientemente publicada define una extensión normalizada de SQL, que permite la creación de respuesta XML anidada. La norma contiene varias partes, incluyendo una manera estándar

de identificar los tipos SQL con los tipos de XML Schema, y una manera estándar de identificar esquemas relacionales con esquemas XML, así como extensiones del lenguaje de consultas SQL.

Por ejemplo, la representación SQL/XML de la relación *cuenta* tendría un esquema XML con *cuenta* como elemento más externo, con cada tupla asociada a un elemento *fila* de XML, y cada atributo de la relación asociada a un elemento de XML del mismo nombre (con algunos convenios para resolver incompatibilidades con los caracteres especiales de los nombres). También se puede asociar un esquema SQL completo, con varias relaciones, a XML de manera parecida. La Figura 10.14 muestra la representación SQL/XML del esquema *banco* que contiene las relaciones *cuenta*, *cliente* e *impositor*.

SQL/XML añade varios operadores y operaciones de agregación a SQL para permitir la construcción de la salida XML directamente a partir de SQL extendido. La función **xmlelement** se puede utilizar para crear elementos de XML, mientras que se puede usar **xmlattributes** para crear atributos, según se ilustra en la siguiente consulta.

```
select xmlelement( name "cuenta",
                   xmlattributes( número_cuenta as número_cuenta),
                   xmlelement( name "nombre_sucursal", nombre_sucursal),
                   xmlelement( name "saldo", saldo))
       from cuenta
```

Esta consulta crea un elemento XML para cada cuenta, con el número de cuenta representado como atributo, y nombre de la sucursal y saldo como subelementos. El resultado sería como los elementos cuenta mostrados en la Figura 10.9, sin el atributo tenedores. El operador **xmlattributes** crea el nombre de atributo XML usando el nombre del atributo SQL, que se puede cambiar usando la cláusula **as** según se ha visto.

El operador **xmlforest** simplifica la construcción de las estructuras XML. Su sintaxis y comportamiento son similares a los de **xmlattributes**, excepto que crea un bosque (colección) de subelementos, en vez de una lista de atributos. Toma varios argumentos, creando un elemento para cada argumento con el nombre SQL del atributo usado como nombre del elemento XML. El operador del **xmlconcat** se puede utilizar para concatenar los elementos creados por subexpresiones en un bosque.

Cuando el valor SQL usado para construir un atributo es nulo, se omite. Los valores nulos se omiten cuando se construye el cuerpo de un elemento.

SQL/XML también proporciona la nueva función de agregación **xmlagg** que crea un bosque (colección) de elementos XML a partir de la colección de los valores a los que se aplica. La consulta siguiente crea un elemento para cada sucursal, conteniendo como subelementos todos los números de cuenta en esa sucursal. Puesto que la pregunta tiene una cláusula **group by** *nombre_sucursal*, la función de agregación se aplica a todas las cuentas de cada sucursal, lo que crea una secuencia de elementos número de cuenta.

```
select xmlelement( name "sucursal",
                   nombre_sucursal,
                   xmlagg ( xmlforest(número_cuenta)
                            order by número_cuenta))
from cuenta
group by nombre_sucursal
```

SQL/XML permite que la secuencia creada por **xmlagg** esté ordenada, según se ilustra en la consulta anterior. Véanse las notas bibliográficas para consultar referencias sobre más información de SQL/XML.

10.7 Aplicaciones XML

A continuación se describen varias aplicaciones de XML para el almacenamiento y comunicación (intercambio) de datos para el acceso a servicios Web (recursos de información).

```
<banco>
<cuenta>
        <fila>
            <número_cuenta> C-101 </número_cuenta>
            <nombre_sucursal> Centro </nombre_sucursal>
            <saldo> 500 </saldo>
        </fila>
        <fila>
            <número_cuenta> C-102 </número_cuenta>
            <nombre_sucursal> Navacerrada </nombre_sucursal>
            <saldo> 400 </saldo>
        </fila>
        <fila>
            <número_cuenta> C-201 </número_cuenta>
            <nombre_sucursal> Galapagar </nombre_sucursal>
            <saldo> 900 </saldo>
        </fila>
    </cuenta>
    <cliente>
        <fila>
            <nombre_cliente> González </nombre_cliente>
            <calle_cliente> Arenal </calle_cliente>
            <ciudad_cliente> La Granja </ciudad_cliente>
        </fila>
        <fila>
            <nombre_cliente> López </nombre_cliente>
            <calle_cliente> Mayor </calle_cliente>
            <ciudad_cliente> Peguerinos </ciudad_cliente>
        </fila>
    </cliente>
    <impositor>
        <fila>
            <número_cuenta> C-101 </número_cuenta>
            <nombre_cliente> González </nombre_cliente>
        </fila>
        <fila>
            <número_cuenta> C-201 </número_cuenta>
            <nombre_cliente> González </nombre_cliente>
        </fila>
        <fila>
            <número_cuenta> C-102 </número_cuenta>
            <nombre_cliente> López </nombre_cliente>
        </fila>
    </impositor>
</banco>
```

Figura 10.14 Representación SQL/XML de información bancaria.

10.7.1 Almacenamiento de datos con estructura compleja

Muchas aplicaciones necesitan almacenar datos que están estructurados, pero no se modelan fácilmente como relaciones. Considérense, por ejemplo, las preferencias de usuario que una aplicación como un navegador debe almacenar. Normalmente existe un gran número de campos, como la página de inicio, ajustes de seguridad, de idioma y de la representación que se deben registrar. Algunos de los campos son multivalorados, por ejemplo, una lista de sitios de confianza o quizás listas ordenadas, por ejem-

plo, una lista de marcadores. Las aplicaciones han usado tradicionalmente algún tipo de representación textual para almacenar estos datos. Hoy, un gran número de estas aplicaciones prefieren almacenar esta información de configuración en formato XML. Las representaciones textuales ad hoc usadas anteriormente requieren el esfuerzo de diseñar y crear los programas de análisis para leer el archivo y convertir los datos en una forma que el programa pueda utilizar. La representación XML evita estos dos pasos.

Las representaciones basadas en XML se han propuesto incluso como normas para almacenar los documentos, los datos de hojas de cálculo y otros datos que son parte de paquetes de aplicaciones de oficina.

XML también se usa para representar datos de estructura compleja que se deben intercambiar entre diversas partes de una aplicación. Por ejemplo, un sistema de bases de datos puede representar planes de ejecución de consultas (una expresión del álgebra relacional con información adicional sobre la forma en que se deben ejecutar las operaciones) usando XML. Esto permite que una parte del sistema genere el plan de ejecución de la consulta y otra parte para mostrarla sin usar una estructura de datos compartida. Por ejemplo, los datos se pueden generar en un sistema servidor y enviarse a un sistema cliente donde se muestren.

10.7.2 Formatos normalizados para el intercambio de datos

Se han desarrollando normas basadas en XML para la representación de datos para una gran variedad de aplicaciones especializadas que van desde aplicaciones de negocios tales como banca y transportes a aplicaciones científicas tales como química y biología molecular. Veamos algunos ejemplos:

- La industria química necesita información sobre los compuestos químicos tales como su estructura molecular y una serie de propiedades importantes tales como los puntos de fusión y ebullición, valores caloríficos y solubilidad en distintos disolventes. *ChemML* es una norma para representar dicha información.
- En el transporte, los transportes de mercancías y los agentes de aduana e inspectores necesitan los registros de los envíos que contengan información detallada sobre los bienes que están siendo transportados, por quién y desde dónde se han enviado, a quién y a dónde se envían, el valor monetario de los bienes, etc.
- Un mercado en línea en que los negocios pueden vender y comprar bienes (el llamado mercado B2B—business-to-business, entre negocios) requiere información tal como los catálogos de producto, incluyendo descripciones detalladas de los productos e información de los precios, inventarios de los productos, cuotas para una venta propuesta y pedidos de compras. Por ejemplo las normas *RosettaNet* para aplicaciones de negocios electrónicos definen los esquemas XML y la semántica para la representación de datos, así como normas para el intercambio de mensajes.

El uso de esquemas relacionales normalizados para modelar requisitos de datos tan complejos resultarían en un gran número de relaciones que no se corresponden directamente con los objetos que se modelan. Las relaciones frecuentemente tienen un gran número de atributos; la representación explícita de nombres de atributos y elementos con sus valores en XML ayuda a evitar confusiones entre los atributos. Las representaciones de elementos anidados ayudan a reducir el número de relaciones que se deben representar así como el número de combinaciones requeridas para obtener la información, con el posible coste de redundancia. Por ejemplo, en nuestro ejemplo bancario el listado de clientes con elementos cuenta anidados en elementos cuenta, como en la Figura 10.4, resulta en un formato más adecuado para algunas aplicaciones—en particular para su legibilidad—que la representación normalizada de la Figura 10.1.

10.7.3 Servicios Web

Las aplicaciones requieren a menudo datos externos o de otro departamento de la empresa en una base de datos diferente. En estas situaciones, la empresa externa o el departamento no están dispuestos a permitir acceso directo a su base de datos usando SQL, sino a proporcionar información limitada a través de interfaces predefinidas.

Cuando son las personas quienes deben usar directamente la información, las empresas proporcionan formularios Web donde los usuarios pueden introducir valores y conseguir la información deseada en HTML. Sin embargo, hay muchas aplicaciones en las que son programas los que acceden a esta información, en lugar de personas. Proporcionar los resultados de una consulta en XML es un requisito claro. Además, tiene sentido especificar los valores de la entrada a la consulta también en formato de XML.

En efecto, el proveedor de información define los procedimientos cuya entrada y salida están en formato XML. El protocolo HTTP se utiliza para comunicar la información de entrada y salida, puesto que se usa en gran medida y puede pasar a través de los cortafuegos que las instituciones emplean para mantener aislado el tráfico indeseado de Internet.

El **protocolo simple de acceso a objetos** (**SOAP**, Simple Object Access Protocol) define una norma para invocar procedimientos usando XML para representar la entrada y salida de los procedimientos. SOAP define un esquema XML estándar para representar el nombre del procedimiento y de los indicadores de estado del resultado, como fallo y error. Los parámetros y resultados de los procedimientos son datos XML dependientes de las aplicaciones incorporados en las cabeceras XML de SOAP.

HTTP se usa normalmente como el protocolo de transporte para SOAP, pero también se puede emplear un protocolo basado en mensajes (como correo electrónico sobre el protocolo SMTP). Actualmente la norma SOAP se utiliza mucho. Por ejemplo, Amazon y Google proporcionan procedimientos basados en SOAP para realizar búsquedas y otras actividades. Otras aplicaciones que proporcionen servicios de alto nivel a los usuarios pueden invocar a estos procedimientos. La norma SOAP es independiente del lenguaje de programación subyacente, y es posible que un sitio que funciona con un lenguaje, como C#, invoque un servicio que funcione en otro lenguaje, como Java.

Un sitio que proporcione tal colección de procedimientos SOAP se denomina **servicio Web**. Se han definido varias normas para dar soporte a los servicios Web. El **lenguaje de descripción de servicios Web** (WSDL, Web Services Descripción Language) es un lenguaje usado para describir las capacidades de los servicios Web. WSDL proporciona las capacidades que la definición de la interfaz (o las definiciones de las funciones) proporciona en un lenguaje de programación tradicional, especificando las funciones disponibles y sus tipos de entrada y de salida. Además, WSDL permite la especificación del URL y del número de puerto de red que se utilizarán para invocar el servicio Web. Hay también una norma denominada **descripción, descubrimiento e integración universales** (**UDDI**, Universal Descripción, Discovery and Integration), que define la forma en que se puede crear un directorio de los servicios Web disponibles y cómo un programa puede buscar en el directorio para encontrar un servicio Web que satisfaga sus necesidades.

El siguiente ejemplo ilustra el valor de los servicios Web. Una compañía de líneas aéreas puede definir un servicio Web que proporcione el conjunto de procedimientos que un sitio Web de una agencia de viajes puede invocar; pueden incluir procedimientos para encontrar horarios de vuelo y la información de tasas, así como para hacer reservas. El sitio Web de la agencia puede interactuar con varios servicios Web proporcionados por diversas líneas aéreas, hoteles y otras compañías, proporcionar la información del viaje a los clientes y hacer reservas. Al dar soporte a los servicios Web, las empresas permiten que se construya un servicio útil que integre servicios individuales. Los usuarios pueden interactuar con un solo sitio Web para hacer sus reservas sin tener que entrar en contacto con varios sitios Web diferentes.

Para llamar a un servicio Web los clientes deben preparar un mensaje XML apropiado bajo SOAP y enviarlo al servicio; cuando consigue el resultado en XML, el cliente debe extraer entonces la información del resultado XML. Existen APIs estándar en lenguajes como Java y C# para crear y extraer la información de los mensajes SOAP.

Véanse las notas bibliográficas para referencias a más información sobre los servicios Web.

10.7.4 Mediación de datos

La compra comparada es un ejemplo de aplicación de mediación de datos en la que los datos sobre elementos, inventario, precio y costes de envío se extraen de una serie de sitios Web que ofrecen un elemento concreto de venta. La información agregada resultante es significativamente más valiosa que la información individual ofrecida por un único sitio.

Un gestor financiero personal es una aplicación similar en el contexto de la banca. Consideremos un consumidor con una gran cantidad de cuentas a gestionar, tales como cuentas bancarias, cuentas de aho-

rro y cuentas de jubilación. Supóngase que estas cuentas pueden estar en distintas instituciones. Es un reto importante proporcionar una gestión centralizada de todas las cuentas de un cliente. La mediación basada en XML soluciona el problema extrayendo una representación XML de la información de la cuenta desde los sitios Web respectivos de las instituciones financieras donde está cada cuenta. Esta información se puede extraer fácilmente si la institución la exporta a un formato XML estándar, por ejemplo, como servicio Web. Para aquellas que no lo hacen se usa un software *envolvente* para generar datos XML a partir de las páginas Web HTML devueltas por el sitio Web. Las aplicaciones envolventes necesitan un mantenimiento constante puesto que dependen de los detalles de formato de las páginas Web, que cambian constantemente. No obstante, el valor proporcionado por la mediación frecuentemente justifica el esfuerzo requerido para desarrollar y mantener las aplicaciones envolventes.

Una vez que las herramientas básicas están disponibles para extraer la información de cada fuente, se usa una aplicación *mediadora* para combinar la información extraída bajo un único esquema. Esto puede requerir más transformación de los datos XML de cada sitio, puesto que los distintos sitios pueden estructurar la misma información de una forma diferente. Por ejemplo, uno de los bancos puede exportar información en el formato de la Figura 10.1 aunque otros pueden usar la formato anidado de la Figura 10.4. También pueden usar nombres diferentes para la misma información (por ejemplo num_cuenta e id_cuenta), o pueden incluso usar el mismo nombre para información distinta. El mediador debe decidir sobre un único esquema que representa toda la información requerida, y debe proporcionar código para transformar los datos entre diferentes representaciones. Dichos temas se estudiarán con más detalle en el Apartado 22.8 en el contexto de bases de datos distribuidas. Los lenguajes de consultas XML tales como XSLT y XQuery desempeñan un papel importante en la tarea de transformación entre distintas representaciones XML.

10.8 Resumen

- Al igual que el lenguaje de marcas de hipertexto HTML (Hyper-Text Markup Language), en que está basado la Web, el lenguaje de marcas extensible, XML (Extensible Markup Language), es descendiente del lenguaje estándar generalizado de marcas (SGML, Standard Generalized Markup Language). XML estaba pensado inicialmente para proporcionar marcas funcionales para documentos Web, pero se ha convertido de facto en el formato de datos estándar para el intercambio de datos entre aplicaciones.

- Los documentos XML contienen elementos con etiquetas de inicio y finalización correspondientes que indican el comienzo y finalización de un elemento. Los elementos puede tener subelementos anidados a ellos, a cualquier nivel de anidamiento. Los elementos pueden también tener atributos. La elección entre representar información como atributos o como subelementos suele ser arbitraria en el contexto de la representación de datos.

- Los elementos pueden tener un atributo del tipo ID que almacene un identificador único para el elemento. Los elementos también pueden almacenar referencias a otros elementos mediante el uso de atributos del tipo IDREF. Los atributos del tipo IDREFS pueden almacenar una lista de referencias.

- Los documentos pueden opcionalmente tener su esquema especificado mediante una definición de tipos de documentos (DTD, Document Type Declaration). La DTD de un documento especifica los elementos que pueden aparecer, cómo se pueden anidar y los atributos que puede tener cada elemento. Aunque las DTDs se usan mucho, tienen graves limitaciones. Por ejemplo, no proporcionan un sistema de tipos.

- XML Schema es ahora el mecanismo estándar para especificar el esquema de los documentos XML. Proporciona un gran conjunto de tipos básicos, así como constructores para crear tipos complejos y para especificar restricciones de integridad, incluidas las claves y las claves externas (keyref).

- Los datos XML se pueden representar como estructuras en árbol, como nodos correspondientes a los elementos y atributos. El anidamiento de elementos se refleja mediante la estructura padre-hijo de la representación en árbol.
- Las expresiones de ruta se pueden usar para recorrer la estructura de árbol XML y así localizar los datos requeridos. XPath es un lenguaje normalizado para las expresiones de rutas de acceso y permite especificar los elementos requeridos mediante una ruta parecida a un sistema de archivos y además permite la selección y otras características. XPath también forma parte de otros lenguajes de consultas XML.
- El lenguaje XQuery es el estándar para la consulta de datos XML. Tiene una estructura similar a la de SQL, con cláusulas **for**, **let**, **where**, **order by** y **return**. Sin embargo, soporta muchas extensiones para tratar con la naturaleza en árbol de XML y para permitir la transformación de documentos XML en otros documentos con una estructura significativamente diferente. Las expresiones de ruta XPath forman parte de XQuery. XQuery soporta consultas anidadas y funciones definidas por el usuario.
- El lenguaje XSLT se diseñó originalmente como el lenguaje de transformación para una aplicación de hojas de estilo, en otras palabras, para aplicar información de formato a documentos XML. Sin embargo, XSLT ofrece características bastante potentes de consulta y transformación y está ampliamente disponible, por lo que se usa para consultar datos XML.
- Las APIs DOM y SAX se usan mucho para el acceso mediante programación a datos XML. Estas APIs están disponibles para varios lenguajes de programación.
- Los datos XML se pueden almacenar de varias formas distintas. Los datos XML se pueden almacenar en sistemas de archivos, o en bases de datos XML, que emplean XML como representación interna.

  XML se puede almacenar como cadenas en una base de datos relacional. Alternativamente, las relaciones pueden representar datos XML como árboles. Otra alternativa es que los datos XML se pueden hacer corresponder con relaciones de la misma forma que se hacen corresponder los esquemas E-R con esquemas relacionales. El almacenamiento nativo de XML en las bases de datos relacionales se facilita añadiendo el tipo de datos **xml** a SQL.
- XML se utiliza en gran variedad de aplicaciones, como el almacenamiento de datos complejos, el intercambio de datos entre empresas de forma estándar, la mediación de datos y los servicios Web. Los servicios Web proporcionan una interfaz de llamada a procedimientos remotos, con XML como mecanismo para codificar los parámetros y los resultados.

Términos de repaso

- Lenguaje de marcas extensible (Extensible Markup Language, XML).
- Lenguaje de marcas de hipertexto (HyperText Markup Language, HTML).
- Lenguaje estándar generalizado de marcas (Standard Generalized Markup Language, SGML).
- Lenguaje de marcas.
- Marcas.
- Autodocumentado.
- Elemento.
- Elemento raíz.
- Elementos anidados.
- Atributos.
- Espacio de nombres.
- Espacio de nombres predeterminado.
- Definición del esquema.
- Definición de tipos de documentos (Document Type Definition, DTD).
  - ☐ ID.
  - ☐ IDREF e IDREFS.
- XML Schema.
  - ☐ Tipos simples y complejos.
  - ☐ Tipo secuencia.
  - ☐ Clave y **keyref**.
  - ☐ Restricciones de aparición.
- Modelo de árbol de datos XML.
- Nodos.

- Consulta y transformación.
- Expresiones de ruta.
- XPath.
- XQuery.
  - ☐ Expresiones FLWOR:
    - — **for**.
    - — **let**.
    - — **order by**.
    - — **where**.
    - — **return**.
  - ☐ Reuniones.
  - ☐ Expresión FLWOR anidada.
  - ☐ Ordenación.
- XSL (XML Stylesheet Language), lenguaje de hojas de estilo XML.
- Transformaciones XSL (XSL Transformations, XSLT).
  - ☐ Plantillas:
    - — **match** (coincidencia).
    - — **select** (selección).
  - ☐ Recursividad estructural.
  - ☐ Claves.
  - ☐ Ordenación.
- API XML.
- Modelo de objetos documento (Document Object Model, DOM).
- API simple para XML (Simple API for XML, SAX).
- Almacenamiento de datos XML.
  - ☐ En almacenamientos de datos no relacionales.
  - ☐ En bases de datos relacionales.
    - — Almacenamiento como cadena.
    - — Representación en árbol.
    - — Representación con relaciones.
    - — Publicación y fragmentación.
    - — Base de datos con capacidades XML.
    - — Almacenamiento nativo.
    - — SQL/XML.
- Aplicaciones XML.
  - ☐ Almacenamiento de datos complejos.
  - ☐ Intercambio de datos.
  - ☐ Mediación de datos.
  - ☐ SOAP.
  - ☐ Servicios Web.

Ejercicios prácticos

10.1 Dese una representación alternativa de la información bancaria que contenga los mismos datos que en la Figura 10.1, pero usando atributos en lugar de subelementos. Dese también la DTD para esta representación.

10.2 Dese la DTD para una representación XML del siguiente esquema relacional anidado:

Emp = (nombree, ConjuntoHijos **setof***(Hijos), ConjuntoMaterias* **setof***(Materias))*
Hijos = (nombre, Cumpleaños)
Cumpleaños = (día, mes, año)
Materias = (tipo, ConjuntoExámenes **setof***(Exámenes))*
Exámenes = (año, ciudad)

10.3 Escríbase una consulta en XPath sobre la DTD del Ejercicio 10.2 para listar todos los tipos de materia de *Emp*.

10.4 Escríbase una consulta en XQuery en la representación XML de la Figura 10.1 para encontrar el saldo total de cada sucursal teniendo en cuenta todas las cuentas.

10.5 Escríbase una consulta en XQuery en la representación XML de la Figura 10.1 para calcular la reunión externa por la izquierda de los elementos impositor con los elementos cuenta. *Sugerencia*: se puede usar la cuantificación universal.

10.6 Escríbanse consultas en XQuery y XSLT que devuelvan los elementos cliente con los elementos cuenta asociados anidados en los elementos cliente, dada la representación de la información bancaria usando ID e IDREFS de la Figura 10.9.

10.7 Dese un esquema relacional para representar la información bibliográfica como se especifica en el fragmento DTD de la Figura 10.15. El esquema relacional debe registrar el orden de los elementos autor. Se puede asumir que sólo los libros y los artículos aparecen como elementos de nivel superior en los documentos XML.

```
<!DOCTYPE bibliografía [
    <!ELEMENT libro (título, autor+, año, editor, lugar?)>
    <!ELEMENT artículo (título, autor+, revista, año, número, volumen, páginas?)>
    <!ELEMENT autor ( apellidos, nombre) >
    <!ELEMENT título ( #PCDATA )>
    · · · declaraciones PCDATA similares para el año, el editor, el lugar,
        la revista, el año, el número, el volumen, las páginas, los apellidos y el nombre
] >
```

Figura 10.15 DTD para los datos bibliográficos.

10.8 Mostrar la representación en árbol de los datos XML de la Figura 10.1 y la representación del árbol usando las relaciones *nodos* e *hijo* descritas en el Apartado 10.6.2.

10.9 Considérese la siguiente DTD recursiva:

```
<!DOCTYPE parts [
    <!ELEMENT producto (nombre, infocomponente*)>
    <!ELEMENT infocomponente (producto, cantidad)>
    <!ELEMENT nombre ( #PCDATA )>
    <!ELEMENT cantidad ( #PCDATA )>
] >
```

a. Dese un pequeño ejemplo de datos correspondientes a esta DTD.
b. Muéstrese cómo hacer corresponder este DTD con un esquema relacional. Se puede suponer que los nombres de producto son únicos, esto es, cada vez que aparezca un producto, la estructura de sus componentes será la misma.
c. Créese un esquema en XML Schema correspondiente a esta DTD.

Ejercicios

10.10 Demuéstrese, proporcionando una DTD, cómo representar la relación *libros*, que no está en primera forma normal, del Apartado 9.2 mediante el uso de XML.

10.11 Escríbanse las siguientes consultas en XQuery, asumiendo la DTD del Ejercicio práctico 10.2.

a. Encontrar los nombres de todos los empleados que tienen un hijo cuyo cumpleaños cae en marzo.
b. Encontrar aquellos empleados que se examinaron del tipo de materia "mecanografía" en la ciudad "Madrid".
c. Listar todos los tipos de materias de *Emp*.

10.12 Considérense los datos XML de la Figura 10.2. Supóngase que se desea encontrar los pedidos con dos o más compras del producto con identificador 123. Considérese esta posible solución al problema:

```
for $p in pedidocompra
where $p/id/producto = 123 and $p/cantidad/producto >= 2
return $p
```

Explíquese por qué la pregunta puede devolver algunos pedidos con menos de dos compras del producto 123. Dese una versión correcta de esta consulta.

10.13 Dese una consulta en XQuery para invertir el anidamiento de los datos del Ejercicio 10.10. Esto es, el nivel más externo del anidamiento de la salida debe tener los elementos correspondientes a los autores, y cada uno de estos elementos debe tener anidados los elementos correspondientes a todos los libros escritos por el autor.

10.14 Dese la DTD de una representación XML de la información de la Figura 2.29. Créese un tipo de elemento diferente para representar cada relación, pero úsense ID e IDREF para implementar las claves primarias y externas.

10.15 Dese una representación en XML Schema de la DTD del Ejercicio 10.14.

10.16 Escríbanse las consultas en XQuery del fragmento DTD de bibliografía de la Figura 10.15 para realizar lo siguiente:

a. Determinar todos los autores que tienen un libro y un artículo en el mismo año.
b. Mostrar los libros y artículos ordenados por años.
c. Mostrar los libros con más de un autor.
d. Encontrar todos los libros que contengan la palabra "base de datos" en su título y la palabra "Remedios" en el nombre o apellidos del autor.

10.17 Dese la versión relacional del esquema de pedidos en XML ilustrado en la Figura 10.2, usando el enfoque descrito en el Apartado 10.6.2.3. Sugiérase cómo eliminar la redundancia en el esquema relacional cuando los identificadores determinan funcionalmente a su descripción, y los nombres del comprador y del proveedor determinan funcionalmente a las direcciones del comprador y del proveedor respectivamente.

10.18 Escríbanse consultas en SQL/XML para convertir los datos bancarios del esquema relacional que hemos utilizado en capítulos anteriores a los esquemas XML *banco-1* y *banco-2* (para el esquema de *banco-2* se puede asumir que la relación cliente tiene el atributo adicional *id_cliente*).

10.19 Como en el Ejercicio 10.18, escríbanse consultas para convertir los datos bancarios a los esquemas XML *banco-1* y *banco-2*, pero esta vez escribiendo consultas XQuery sobre la base de datos SQL/XML predeterminada a la versión XML.

10.20 Una forma de fragmentar un documento XML es emplear XQuery para convertir el esquema a la versión SQL/XML del esquema relacional correspondiente y después usar esta versión en sentido inverso para rellenar la relación.

Como ilustración, dese una consulta XQuery para convertir datos del esquema XML *banco-1* al esquema SQL/XML de la Figura 10.14.

10.21 Considérese el esquema XML de ejemplo del Apartado 10.3.2 y escríbanse consultas XQuery para realizar las siguientes tareas:

a. Comprobar si se cumple la restricción de clave mostrada en el Apartado 10.3.2.
b. Comprobar si se cumple la restricción keyref mostrada en el Apartado 10.3.2.

10.22 Considérese el Ejercicio práctico 10.7 y supóngase que los autores también pueden aparecer como elementos de nivel superior ¿Qué cambio habría que realizar en el esquema relacional?

Notas bibliográficas

El consorcio W3C (World Wide Web Consortium) actúa como cuerpo normativo para las normas relacionadas con la Web, incluyendo XML básico y todos los lenguajes relacionados con XML tales como XPath, XSLTy XQuery. Se puede obtener en www.w3c.org gran número de informes técnicos que definen las normas relacionadas con XML. Este sitio también contiene tutoriales y punteros a software que implementa las distintas normas. El sitio XML Cover Pages (www.oasis-open.org/cover/) también contiene bastante información sobre XML, incluida la especificación del lenguaje Relax NG para la especificación de esquemas XML.

En el libro de texto Katz et al. [2004] se proporciona un tratamiento detallado de XQuery. Quilt se describe en Chamberlin et al. [2000]. Deutsch et al. [1999] describe el lenguaje XML-QLLa integración de consultas con palabras clave en XML se describe en Florescu et al. [2000] y Amer-Yahia et al. [2004].

Funderburk et al. [2002a], Florescu y Kossmann [1999], Kanne y Moerkotte [2000] y Shanmugasundaram et al. [1999] describen el almacenamiento de datos XML. Eisenberg y Melton [2004a] proporciona una descripción de SQL/XML, mientras que Funderburk et al. [2002b] proporciona descripciones de

SQL/XML y de XQuery. Schning [2001] describe una base de datos diseñada para XML. Véanse los Capítulos 27 a 29 para obtener más información sobre el soporte de XML en bases de datos comerciales. Eisenberg y Melton [2004b] proporcionan una descripción del API XQJ para XQuery.

Herramientas

Se encuentran disponibles una serie de herramientas de uso público para tratar con XML. El sitio www.oasis-open.org/cover/ contiene enlaces a una serie de herramientas software para XML y XSL (incluido XSLT). El sitio Web www.w3c.org de W3C tiene páginas que describen las diferentes normas relacionadas con XML, así como punteros a las herramientas software como implementaciones de lenguajes. Téngase en cuenta que varias de las implementaciones, tales como Galax, son pruebas conceptuales. Aunque pueden servir como herramientas de aprendizaje, no son capaces de manejar bases de datos grandes. Varias bases de datos comerciales, incluidas DB2 de IBM, Oracle y SQL Server de Microsoft, soportan el almacenamiento XML, la publicación empleando varias extensiones de SQL, y las consultas mediante XPath y XQuery.

PARTE 4

Almacenamiento de datos y consultas

Aunque los sistemas de bases de datos proporcionan una visión de alto nivel de los datos, en último término es necesario guardarlos como bits en uno o varios dispositivos de almacenamiento. Una amplia mayoría de sistemas de bases de datos actuales almacenan los datos en discos magnéticos y los extraen a la memoria principal para su procesamiento, o los copian como archivos en cintas u otros dispositivos de copia de seguridad. Las características físicas de los dispositivos de almacenamiento desempeñan un papel importante en el modo en que se almacenan los datos, en especial porque el acceso a un fragmento aleatorio de los datos en el disco resulta mucho más lento que el acceso a la memoria: los accesos al disco invierten decenas de milisegundos, mientras que el acceso a la memoria invierte una décima de microsegundo.

En el Capítulo 11 se comienza con una introducción a los medios físicos de almacenamiento, incluidos los mecanismos para minimizar las posibilidades de pérdida de datos debidas a fallos de los dispositivos. A continuación se describe el modo en que se asignan los registros a los archivos que, a su vez, se asignan a bits del disco. El almacenamiento y la recuperación de los objetos se tratan también en el Capítulo 11.

Muchas consultas sólo hacen referencia a una pequeña parte de los registros de un archivo. Los índices son estructuras que ayudan a localizar rápidamente los registros deseados de una relación, sin tener que examinar todos sus registros. El índice de este libro de texto es un ejemplo de ello aunque está pensado para su empleo por personas, a diferencia de los índices de las bases de datos. En el Capítulo 12 se describen varios tipos de índices utilizados en los sistemas de bases de datos.

Las consultas de los usuarios tienen que ejecutarse sobre el contenido de la base de datos, que reside en los dispositivos de almacenamiento. Suele ser conveniente dividir las consultas en operaciones más pequeñas, que se correspondan aproximadamente con las operaciones del álgebra relacional. En el Capítulo 13 se describe el modo en que se procesan las consultas, presenta los algoritmos para la implementación de las operaciones individuales y esboza el modo en que las operaciones se ejecutan en sincronía para procesar una consulta.

Existen muchas maneras alternativas de procesar cada consulta con costes muy distintos. La optimización de consultas hace referencia al proceso de hallar el método de coste mínimo para evaluar una consulta dada. En el Capítulo 14 se describe este proceso.

CAPÍTULO 11

Almacenamiento y estructura de archivos

En los capítulos anteriores se han estudiado los modelos de bases de datos de alto nivel. Por ejemplo, en el nivel *conceptual* o *lógico* se ha presentado una bases de datos del modelo relacional como un conjunto de tablas. En realidad, el modelo lógico de las bases de datos es el mejor nivel para que se centren los *usuarios*. Esto se debe a que el objetivo de los sistemas de bases de datos es simplificar y facilitar el acceso a los datos; no se debe agobiar innecesariamente a quienes utilizan el sistema con los detalles físicos de su implementación.

En este capítulo, no obstante, así como en los Capítulos 12, 13 y 14, se analizan niveles inferiores y se describen diferentes métodos de implementación de los modelos de datos y de los lenguajes presentados en capítulos anteriores. Se comienza con las características de los medios de almacenamiento subyacentes, como sistemas de disco y de cinta. Más adelante se definen varias estructuras de datos que permiten un acceso rápido a los datos. Se consideran varias arquitecturas alternativas, idóneas para diferentes tipos de acceso a los datos. La elección final de la estructura de datos hay que hacerla en función del uso que se espera dar al sistema y de las características de cada máquina concreta.

11.1 Visión general de los medios físicos de almacenamiento

La mayoría de sistemas informáticos presentan varios tipos de almacenamientos de datos. Estos medios se clasifican según la velocidad con la que se puede tener acceso a los datos, su coste de adquisición por unidad de datos y su fiabilidad. Entre los medios disponibles habitualmente figuran:

- **Caché**. La caché es el mecanismo de almacenamiento más rápido y costoso. La memoria caché es pequeña; su uso lo gestiona el hardware del sistema informático. Los sistemas de bases de datos no la gestionan.
- **Memoria principal**. El medio de almacenamiento utilizado para los datos con los que se opera es la memoria principal. Las instrucciones máquina operan en la memoria principal. Aunque la memoria principal puede contener muchos megabytes de datos (un PC normal viene con 512 megabytes, como mínimo), o incluso cientos de gigabytes de datos en grandes sistemas servidores suele ser demasiado pequeña (o demasiado cara) para guardar toda la base de datos. El contenido de la memoria principal suele perderse en caso de fallo del suministro eléctrico o de caída del sistema.
- **Memoria flash**. La memoria flash se diferencia de la memoria principal en que los datos no se pierden en caso de que se produzca un corte de la alimentación. La lectura de los datos de la memoria flash emplea menos de cien nanosegundos (un nanosegundo es una milésima de microsegundo), más o menos igual de rápida que la lectura de los datos de la memoria principal. Sin embargo, la escritura de los datos en la memoria flash resulta más complicada (los datos pueden escribirse una vez, lo que tarda de cuatro a diez microsegundos, pero no se pueden sobrescribir

de manera directa). Para sobrescribir la memoria es necesario borrar simultáneamente todo un banco de memoria, el cual queda preparado para volver a escribir en él. Un inconveniente de la memoria flash es que sólo permite un número limitado de ciclos de borrado, que varía entre diez mil y un millón. Es un tipo de *memoria sólo de lectura programable y borrable eléctricamente* (*electrically erasable programmable read-only memory*, *EEPROM*); otras variantes de EEPROM permiten el borrado y reescritura de posiciones individuales de la memoria, pero su empleo no está tan difundido.

La memoria flash se ha hecho popular como sustituta de los discos magnéticos para el almacenamiento de pequeños volúmenes de datos (en el año 2005, normalmente menos de un gigabyte, aunque ya hay memorias flash de mayor capacidad que pueden almacenar varios gigabytes) en los sistemas informáticos de coste reducido que se empotran en otros dispositivos, en computadoras de mano y en otros dispositivos electrónicos digitales como las cámaras digitales (a principios del año 2005, 256 megabytes de memoria flash para cámaras tenían un coste menor de 25 €, mientras que un gigabyte tenía un coste menor de 100 €). La memoria flash se utiliza también en las "llaves USB", que se pueden conectar a los puertos USB (**Universal Serial Bus**, bus serie universal) de los dispositivos informáticos. Estas llaves USB son muy populares para el transporte datos entre sistemas informáticos (los "disquetes" desempeñaron el mismo papel en su momento, pero han quedado obsoletos debido a su capacidad limitada).

- **Almacenamiento en discos magnéticos**. El principal medio de almacenamiento persistente en conexión es el disco magnético. Generalmente se guarda en ellos toda la base de datos. Para acceder a los datos es necesario trasladarlos desde el disco a la memoria principal. Después de realizar la operación deseada se deben escribir en el disco los datos que se hayan modificado.

  El tamaño de los discos magnéticos actuales varía entre unos pocos gigabytes y 400 gigabytes. Un disco de 250 gigabytes tiene un coste en 2005 alrededor de 160 €. Los dos extremos de este rango han crecido a un ritmo cercano al cincuenta por ciento anual, y cada año se pueden esperar discos de mucha mayor capacidad. El almacenamiento en disco resiste los fallos del suministro eléctrico y las caídas del sistema. Los propios dispositivos de almacenamiento en disco pueden fallar a veces y, en consecuencia, destruir los datos, pero estos fallos se producen con mucha menos frecuencia que las caídas del sistema.

- **Almacenamiento óptico**. La forma más popular de almacenamiento óptico es el disco compacto (*Compact Disk*, CD), que puede almacenar alrededor de 700 megabytes de datos y un tiempo de reproducción de 80 minutos, y el *disco de vídeo digital* (*Digital Video Disk*, DVD), que puede almacenar 4,7 u 8,5 gigabytes de datos en cada cara del disco (o hasta 17 gigabytes en un disco de doble cara). También se utiliza el término **disco digital versátil** en lugar de **disco de vídeo digital**, ya que los DVD pueden almacenar cualquier tipo de dato digital, no sólo datos de vídeo. Los datos se almacenan ópticamente en el disco y se leen mediante un láser.

  No se puede escribir en los discos ópticos empleados como discos compactos de sólo lectura (CD-ROM) o como discos de vídeo digital de sólo lectura (DVD-ROM), pero se suministran con datos pregrabados. Existen también versiones "para una sola grabación" de los discos compactos (denominados CD-R) y de los discos de vídeo digital (DVD-R y DVD+R) en los que sólo se puede escribir una vez; estos discos también se denominan **de escritura única y lectura múltiple** (**write-once, read-many**, WORM). Existen también versiones "para escribir varias veces" de los discos compactos (llamada CD-RW) y de los discos de vídeo digital (DVD-RW, DVD+RW y DVD-RAM), en los que se puede escribir varias veces.

  Los **cambiadores automáticos** (**jukebox**) de discos ópticos contienen varias unidades y numerosos discos que pueden cargarse de manera automática en las diferentes unidades (mediante un brazo robotizado) a petición del usuario.

- **Almacenamiento en cinta**. El almacenamiento en cinta se utiliza principalmente para copias de seguridad y datos archivados. Aunque la cinta magnética es más barata que los discos, el acceso a los datos resulta mucho más lento, ya que hay que acceder secuencialmente desde el comienzo de la cinta. Por este motivo, se dice que el almacenamiento en cinta es de **acceso secuencial**,

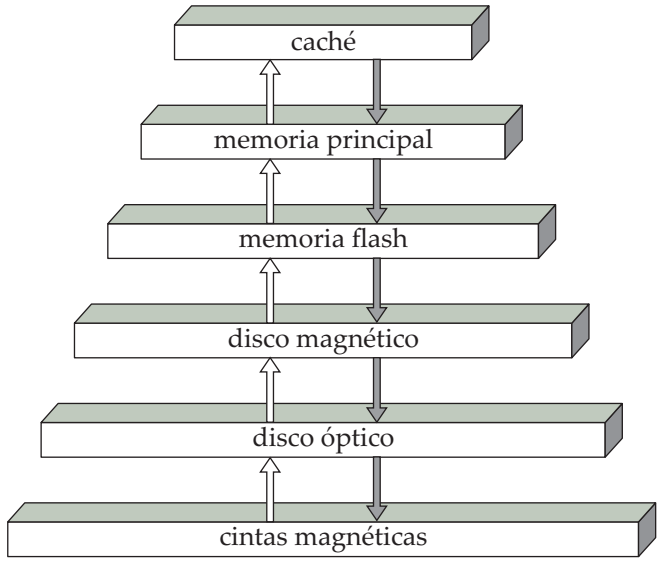


Figura 11.1 Jerarquía de los dispositivos de almacenamiento.

mientras que el almacenamiento en disco es de **acceso directo**, ya que se pueden leer datos de cualquier parte del disco.

Las cintas poseen una capacidad elevada (actualmente se dispone de cintas de 40 a 300 gigabytes) y pueden retirarse de la unidad de lectura, por lo que resultan idóneas para el almacenamiento de archivos con coste reducido. Los cambiadores automáticos de cinta (jukeboxes) se utilizan para guardar conjuntos de datos excepcionalmente grandes, como los datos obtenidos mediante satélite, que pueden ocupar centenares de terabytes (10^{12} bytes), o incluso petabytes (10^{15} bytes) en algunos casos.

Los diferentes medios de almacenamiento se pueden organizar de forma jerárquica (Figura 11.1) de acuerdo con su velocidad y su coste. Los niveles superiores resultan caros, pero son rápidos. A medida que se desciende por la jerarquía disminuye el coste por bit, pero aumenta el tiempo de acceso. Este compromiso es razonable; si un sistema de almacenamiento dado fuera a la vez más rápido y menos costoso que otro (en igualdad del resto de condiciones) no habría ninguna razón para utilizar la memoria más lenta y más cara. De hecho, muchos dispositivos de almacenamiento primitivos, como la cinta de papel y las memorias de núcleos de ferrita, se hallan relegados a los museos ahora que la cinta magnética y la memoria de semiconductores son más rápidas y baratas. Las propias cintas magnéticas se utilizaban para guardar los datos activos cuando los discos resultaban costosos y tenían una capacidad de almacenamiento reducida. Hoy en día casi todos los datos activos se almacenan en disco, excepto en los casos excepcionales en que se guardan en cambiadores automáticos de cinta u ópticos.

Los medios de almacenamiento más rápidos (por ejemplo, la caché y la memoria principal) se denominan **almacenamiento primario**. Los medios del siguiente nivel de la jerarquía (por ejemplo, los discos magnéticos) se conocen como **almacenamiento secundario** o **almacenamiento en conexión**. Los medios del nivel inferior de la jerarquía—por ejemplo, la cinta magnética y los cambiadores automáticos de discos ópticos—se denominan **almacenamiento terciario** o **almacenamiento sin conexión**.

Además de la velocidad y del coste de los diferentes sistemas de almacenamiento hay que tener en cuenta la volatilidad del almacenamiento. El **almacenamiento volátil** pierde su contenido cuando se suprime el suministro eléctrico del dispositivo. En la jerarquía mostrada en la Figura 11.1 los sistemas de almacenamiento desde la memoria principal hacia arriba son volátiles, mientras que los sistemas de almacenamiento por debajo de la memoria principal son no volátiles. Los datos se deben escribir en **almacenamiento no volátil** por motivos de seguridad. Este asunto se volverá a tratar en el Capítulo 17.

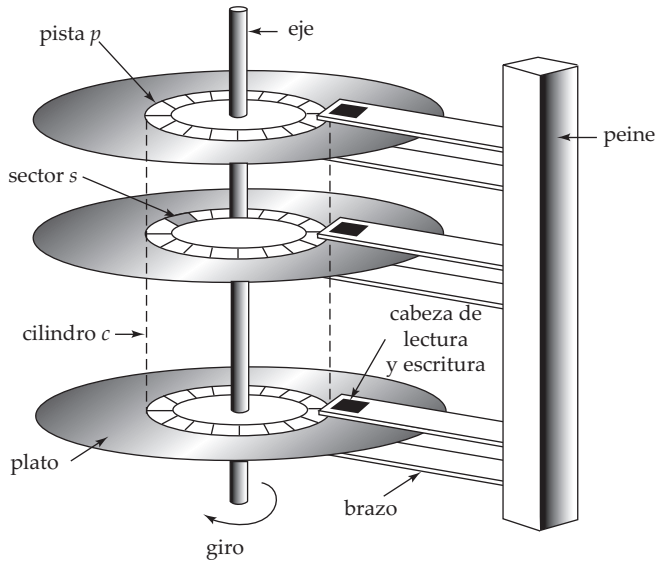


Figura 11.2 Mecanismo de disco de cabezas móviles.

11.2 Discos magnéticos

Los discos magnéticos proporcionan la mayor parte del almacenamiento secundario de los sistemas informáticos modernos. Aunque la capacidad de los discos ha crecido año tras año, los requisitos de almacenamiento de las grandes aplicaciones también han crecido muy rápido; en algunos casos, más que la capacidad de los discos. Una base de datos de gran tamaño puede necesitar centenares de discos.

11.2.1 Características físicas de los discos

Físicamente, los discos son relativamente sencillos (Figura 11.2). Cada **plato** del disco es de forma circular plana. Sus dos superficies están cubiertas por un material magnético en el que se graba la información. Los platos están hechos de metal rígido o de vidrio.

Mientras se utiliza el disco, un motor lo hace girar a una velocidad constante elevada (generalmente sesenta, noventa o ciento veinte revoluciones por segundo, aunque existen discos que giran a doscientas cincuenta revoluciones por segundo). Una cabeza de lectura y escritura está colocada justo encima de la superficie del plato. La superficie del disco se divide a efectos lógicos en **pistas**, que se subdividen en **sectores**. Un **sector** es la unidad mínima de información que se puede leer o escribir en el disco. En los discos actuales, el tamaño de los sectores suele ser de quinientos doce bytes; hay entre cincuenta mil y cien mil pistas en cada plato y de uno a cinco platos por disco. Las pistas internas (las más cercanas al eje) son más cortas, y en los discos actuales, las pistas exteriores contienen más sectores que las internas; suele haber unos quinientos sectores por pista en las pistas internas y alrededor de mil en las externas. Estos números pueden variar de un modelo a otro; los discos de mayor capacidad tienen más sectores por pista y más pistas en cada plato.

La **cabeza de lectura y escritura** guarda la información en los sectores en forma de inversiones de la dirección de magnetización del material magnético.

Cada cara de un plato del disco posee una cabeza de lectura y escritura que se desplaza por el plato para tener acceso a las diferentes pistas. El disco suele contener muchos platos y las cabezas de lectura y escritura de todas las pistas están montadas en un único dispositivo denominado **brazo del disco** y se mueven conjuntamente. El conjunto de los platos del disco montados sobre un mismo eje y de las cabezas montadas en el brazo del disco se denomina **dispositivo cabeza–disco**. Dado que las cabezas de todos los platos se desplazan conjuntamente, cuando la cabeza se halle en la pista i-ésima de un plato, las restantes también se encontrarán en la pista i-ésima de sus platos respectivos. Por tanto, el conjunto de las pistas i-ésimas de todos los platos se denomina **cilindro** i-ésimo.

Actualmente dominan el mercado los discos con un diámetro de plato de tres pulgadas y media. Tienen un menor coste y tiempos de búsqueda más cortos (debido al menor tamaño) que los discos de mayor tamaño (de hasta catorce pulgadas) que eran habituales en el pasado y, aun así, ofrecen gran capacidad de almacenamiento. Se usan discos de diámetro incluso menor en dispositivos móviles como las computadoras portátiles, las de mano y los reproductores de música de bolsillo.

Las cabezas de lectura y escritura se mantienen tan próximas como resulta posible a la superficie de los discos para aumentar la densidad de grabación. Las cabezas suelen flotar o volar tan sólo a micras de la superficie de cada disco; el giro del disco crea una pequeña corriente de aire y el dispositivo de las cabezas se fabrica de manera que ese flujo de aire mantenga las cabezas flotando rasantes sobre la superficie de los discos. Como las cabezas flotan tan cercanas a la superficie, los platos se deben elaborar con esmero para que sean lisos.

Los choques de las cabezas con la superficie de los platos pueden suponer un problema. Si la cabeza entra en contacto con la superficie del disco, puede arrancar el medio de grabación, lo que destruye los datos que allí hubiera. En los modelos antiguos, el contacto de la cabeza con la superficie hacía que el material arrancado flotase en el aire y se interpusiera entre las cabezas y los platos, lo que causaba más choques; por tanto, la caída de las cabezas podía dar lugar a un fallo de todo el disco. Los dispositivos actuales emplean como medio de grabación una fina capa de metal magnético. Son mucho menos susceptibles de fallar a causa del choque de las cabezas con las superficies de los platos que los antiguos discos recubiertos de óxido.

El **controlador de disco** actúa como interfaz entre el sistema informático y el hardware concreto de la unidad de disco; en los sistemas de disco modernos el controlador se implementa dentro de la unidad de disco. El controlador de disco acepta los comandos de alto nivel de lectura o escritura en un sector dado e inicia las acciones correspondientes, como el desplazamiento del brazo del disco a la pista adecuada y la lectura o escritura real de los datos. Los controladores de disco también añaden **sumas de comprobación** a cada sector en el que se escribe; las sumas de comprobación se calculan a partir de los datos escritos en ese sector. Cuando se vuelve a realizar allí una operación de lectura, se vuelve a calcular esa suma a partir de los datos recuperados y se compara con el valor guardado; si los datos se han deteriorado, resulta muy probable que la suma de comprobación recién calculada no coincida con la guardada. Si se produce un error de este tipo, el controlador volverá a intentar la lectura varias veces; si el error persiste, el controlador indica un fallo de lectura.

Otra labor interesante llevada a cabo por los controladores de disco es la **reasignación de los sectores dañados**. Si el controlador detecta que un sector está dañado cuando se da formato al disco por primera vez, o cuando se realiza un intento de escribir allí, puede reasignar lógicamente el sector a una ubicación física diferente (escogida de entre un grupo de sectores adicionales preparados con esta finalidad). La reasignación se anota en disco o en memoria no volátil y la escritura se realiza en la nueva ubicación.

Los discos se conectan a los sistemas informáticos mediante conexiones de alta velocidad. Las interfaces más comunes para la conexión de los discos a las computadoras personales y a las estaciones de trabajo son: (1) la interfaz **ATA** (**AT Attachment**) (que es una versión más rápida que la **interfaz electrónica de dispositivos integrados** (IDE, **Integrated Drive Electronics**) usada antiguamente en los PCs de IBM); (2) la nueva versión de ATA: **SATA** (**ATA serie**, **Serial ATA**; la versión original de ATA se denomina **PATA**, o Parallel ATA- ATA paralela - para distinguirla de SATA); y (3) la **interfaz de conexión para sistemas informáticos pequeños** (SCSI, **Small Computer System Interconnect**, pronunciado "escasi"). Los grandes sistemas (mainframes) y los sistemas servidores suelen disponer de interfaces más rápidas y caras, como las versiones de alta capacidad de la interfaz SCSI y la interfaz Fibre Channel. Los sistemas de discos externos portátiles utilizan generalmente las interfaces USB o FireWire.

Aunque los discos se suelen conectar directamente a la interfaz de disco de la computadora mediante cables, también pueden hallarse en una ubicación remota y conectarse mediante una red de alta velocidad al controlador de disco. En la arquitectura de **red de área de almacenamiento** (**Storage Area Network**, **SAN**) se conecta un gran número de discos a varias computadoras servidoras mediante una red de alta velocidad. Los discos suelen disponerse localmente mediante una técnica de organización de almacenamiento denominada **disposición redundante de discos independientes** (**redundant arrays of independent disks**, RAID) (que se describe en el Apartado 11.3) para proporcionar a los servidores la vista lógica de un solo disco de gran tamaño y muy fiable. La computadora y el subsistema de disco siguen usando las interfaces SCSI y Fibre Channel para comunicarse entre sí, aunque puedan estar sepa-

rados por una red. El acceso remoto a los discos también implica que pueden compartirse entre varias computadoras que pueden ejecutar en paralelo diferentes partes de una misma aplicación. El acceso remoto también supone que los discos que contienen datos importantes se pueden guardar en una sala donde los administradores del sistema pueden supervisarlos y mantenerlos, en lugar de estar dispersos por diferentes locales de la organización.

El **almacenamiento conectado en red** (**Network Attached Storage, NAS**) es una alternativa a SAN. NAS es muy parecido a SAN, salvo que, en lugar de que el almacenamiento en red aparezca como un gran disco, proporciona una interfaz al sistema de archivos que utiliza protocolos de sistemas de archivos en red como NFS o CIFS.

11.2.2 Medidas del rendimiento de los discos

Las principales medidas de la calidad de los discos son su capacidad, su tiempo de acceso, su velocidad de transferencia de datos y su fiabilidad.

El **tiempo de acceso** es el tiempo transcurrido desde que se formula una solicitud de lectura o de escritura hasta que comienza la transferencia de datos. Para tener acceso (es decir, para leer o escribir) a los datos de un sector dado del disco, primero se debe desplazar el brazo para que se ubique sobre la pista correcta y luego hay que esperar a que el sector aparezca bajo él debido a la rotación del disco. El tiempo para volver a ubicar el brazo se denomina **tiempo de búsqueda** y aumenta con la distancia que deba recorrer. Los tiempos de búsqueda típicos varían de dos a treinta milisegundos, en función de la distancia de la pista a la posición inicial del brazo. Los discos de menor tamaño tienden a tener tiempos de búsqueda menores, dado que la cabeza tiene que recorrer una distancia menor.

El **tiempo medio de búsqueda** es el promedio de los tiempos de búsqueda medido en una sucesión de solicitudes aleatorias (uniformemente distribuidas). Si todas las pistas tienen el mismo número de sectores y se desprecia el tiempo necesario para que la cabeza comience a moverse y se detenga, se puede demostrar que el tiempo medio de búsqueda es un tercio del peor de los tiempos de búsqueda posibles. Teniendo en cuenta estos factores, el tiempo medio de búsqueda es aproximadamente la mitad del tiempo máximo de búsqueda. Los tiempos medios de búsqueda varían actualmente entre cuatro y diez milisegundos, dependiendo del modelo de disco.

Una vez que la cabeza ha alcanzado la pista deseada, el tiempo de espera hasta que el sector al que hay que acceder aparece bajo la cabeza se denomina **tiempo de latencia rotacional**. Las velocidades rotacionales de los discos varían actualmente entre cinco mil cuatrocientas rotaciones por minuto (noventa rotaciones por segundo) y quince mil rotaciones por minuto (doscientas cincuenta rotaciones por segundo) o, lo que es lo mismo, de cuatro a 11,1 milisegundos por rotación. En promedio, hace falta media rotación del disco para que aparezca bajo la cabeza el comienzo del sector deseado. Por tanto, el **tiempo de latencia medio** del disco es la mitad del tiempo empleado en una rotación completa del disco.

El tiempo de acceso es la suma del tiempo de búsqueda y del tiempo de latencia y varía de ocho a veinte milisegundos. Una vez situado bajo la cabeza el primer sector de los datos, comienza la transferencia. La **velocidad de transferencia de datos** es la velocidad a la que se pueden recuperar datos del disco o guardarlos en él. Los sistemas de disco actuales soportan velocidades máximas de transferencia de veinticinco a cien megabytes por segundo; la velocidad de transferencia medias son significativamente menores que la velocidad de transferencia máxima para las pistas interiores del disco, dado que éstas tienen menos sectores. Por ejemplo, un disco con una velocidad de transferencia máxima de cien megabytes por segundo puede tener una velocidad de transferencia continuada de alrededor de treinta megabytes por segundo en sus pistas internas.

La última medida que se estudiará de entre las utilizadas habitualmente es el **tiempo medio entre fallos**, que indica una medida de la fiabilidad del disco. El tiempo medio entre fallos de un disco (o de cualquier otro sistema) es la cantidad de tiempo que, en promedio, se puede esperar que el sistema funcione de manera continua sin tener ningún fallo. De acuerdo con las afirmaciones de los fabricantes, el tiempo medio entre fallos de los discos actuales varía entre quinientas mil y un millón doscientas mil horas (de cincuenta y siete a ciento treinta y seis años). En la práctica, el tiempo medio entre fallos anunciado se calcula en términos de la probabilidad de fallo cuando el disco es nuevo (este dato significa que, dados mil discos nuevos, si el tiempo medio entre fallos es de un millón doscientas mil horas, en promedio, uno de ellos fallará cada mil doscientas horas). Un tiempo medio entre fallos de un

millón doscientas mil horas no implica que se pueda esperar que el disco funcione ciento treinta y seis años. La mayoría de los discos tienen una esperanza de vida de unos cinco años, y tienen tasas de fallo significativamente más altas en cuanto alcanzan cierta edad.

Las unidades de disco para las máquinas de sobremesa incorporan normalmente las interfaces PATA (Parallel ATA), las cuales proporcionan velocidades de transferencia de ciento treinta y tres megabytes por segundo, y SATA (Serial ATA), capaces de soportar ciento cincuenta megabytes por segundo. Las antiguas interfaces ATA-4 y ATA-5 soportaban velocidades de transferencia de treinta y tres y sesenta y seis megabytes por segundo, respectivamente. Las unidades de disco diseñadas para los sistemas servidores suelen soportar las interfaces Ultra320 SCSI, que permite hasta trescientos veinte megabytes por segundo, y Fibre Channel FC 2Gb, que proporciona tasas de hasta doscientos cincuenta y seis megabytes por segundo. La velocidad de transferencia de cada interfaz se comparte entre todos los discos conectados a la misma (SATA sólo permite que se conecte un disco a cada interfaz).

11.2.3 Optimización del acceso a los bloques del disco

Las solicitudes de E/S al disco las generan tanto el sistema de archivos como el gestor de la memoria virtual presente en la mayor parte de los sistemas operativos. Cada solicitud especifica la dirección del disco a la que hay que hacer referencia; esa dirección se expresa como un *número de bloque*. Un **bloque** es una unidad lógica que consiste en un número fijo de sectores contiguos. El tamaño de los bloques varía de quinientos doce bytes a varios kilobytes. Entre el disco y la memoria principal los datos se transfieren por bloques.

El acceso a los datos del disco es varios órdenes de magnitud más lento que el acceso a los datos de la memoria principal. En consecuencia, se han desarrollado diversas técnicas para mejorar la velocidad de acceso a los bloques del disco. El almacenamiento de bloques en la memoria intermedia para satisfacer las solicitudes futuras es una de estas técnicas, la cual se estudia en el Apartado 11.5; aquí se examinan otras.

- **Planificación**. Si hay que transferir varios bloques de un mismo cilindro desde el disco a la memoria principal es posible disminuir el tiempo de acceso solicitando los bloques en el orden en el que pasarán por debajo de las cabezas. Si los bloques deseados se hallan en cilindros diferentes resulta ventajoso solicitar los bloques en un orden que minimice el movimiento del brazo del disco. Los algoritmos de **planificación del brazo del disco** intentan ordenar el acceso a las pistas de manera que se aumente el número de accesos que se puedan procesar. Un algoritmo utilizado con frecuencia es el **algoritmo del ascensor**, que funciona de manera parecida a muchos ascensores. Supóngase que, inicialmente, el brazo se desplaza desde la pista más interna hacia el exterior del disco. Bajo el control del algoritmo del ascensor, el brazo se detiene en cada pista para la que haya una solicitud de acceso, atiende las peticiones para esa pista y continúa desplazándose hacia el exterior hasta que no queden solicitudes pendientes para pistas más externas. En ese punto, el brazo cambia de dirección, se desplaza hacia el interior y vuelve a detenerse en cada pista solicitada hasta que alcanza una pista en la que no haya solicitudes para pistas más cercanas al centro del disco. Entonces cambia de dirección e inicia un nuevo ciclo. Los controladores de disco suelen realizar la labor de reordenar las solicitudes de lectura para mejorar el rendimiento, dado que conocen perfectamente la organización de los bloques del disco, la posición rotacional de los platos y la posición del brazo.
- **Organización de archivos**. Para reducir el tiempo de acceso a los bloques se pueden organizar los bloques del disco de una manera que se corresponda fielmente con la forma en que se espera tener acceso a los datos. Por ejemplo, si se espera tener acceso secuencial a un archivo, se deben guardar secuencialmente en cilindros adyacentes todos los bloques del archivo. Los sistemas operativos más antiguos, como los de IBM para grandes sistemas, ofrecían a los programadores un control detallado de la ubicación de los archivos, lo que permitía reservar un conjunto de cilindros para guardar un archivo. Sin embargo, este control supone una carga para el programador o el administrador del sistema porque que tienen que decidir, por ejemplo, los cilindros que debe asignar a cada archivo. Esto puede exigir una costosa reorganización si se insertan datos en el archivo o se borran de él.

Los sistemas operativos posteriores, como Unix y Windows, ocultan a los usuarios la organización del disco y gestionan la asignación de manera interna. Sin embargo, en el transcurso del tiempo, un archivo secuencial puede quedar **fragmentado**; es decir, sus bloques pueden quedar dispersos por el disco. Para reducir la fragmentación, el sistema puede hacer una copia de seguridad de los datos del disco y restaurarlo completamente. La operación de restauración vuelve a escribir de manera contigua (o casi) los bloques de cada archivo. Algunos sistemas (como las diferentes versiones del sistema operativo Windows) disponen de utilidades que examinan el disco y desplazan los bloques para reducir la fragmentación. Las mejoras de rendimiento obtenidas con estas técnicas son elevadas en algunos casos.

- **Memoria intermedia de escritura no volátil**. Dado que el contenido de la memoria se pierde durante los fallos de suministro eléctrico, hay que guardar en disco la información sobre las actualizaciones de las bases de datos para que superen las posibles caídas del sistema. Debido a esto, el rendimiento de las aplicaciones de bases de datos con gran cantidad de actualizaciones, como los sistemas de procesamiento de transacciones, depende mucho de la velocidad de escritura en el disco.

  Se puede utilizar **memoria no volátil de acceso aleatorio** (RAM no volátil, NV-RAM, **Nonvolatile Random-Access Memory**) para acelerar drásticamente la escritura en el disco. El contenido de la NV-RAM no se pierde durante los fallos del suministro eléctrico. Una manera habitual de implementar la NV-RAM es utilizar RAM alimentada por baterías. La idea es que, cuando el sistema de bases de datos (o el sistema operativo) solicita que se escriba un bloque en el disco, el controlador del disco escriba el bloque en una memoria intermedia de NV-RAM y comunique de manera inmediata al sistema operativo que la escritura se completó con éxito. El controlador escribirá los datos en el disco cuando no haya otras solicitudes o cuando se llene la memoria intermedia de NV-RAM. Cuando el sistema de bases de datos solicita la escritura de un bloque, sólo percibe un retraso si la memoria intermedia de NV-RAM se encuentra llena. Durante la recuperación de una caída del sistema se vuelven a escribir en el disco todas las operaciones de escritura pendientes en la memoria intermedia de NV-RAM.

  La memoria intermedia NV-RAM se encuentra en algunos discos de altas prestaciones, pero más frecuentemente en los "controladores RAID"; RAID se estudia en el Apartado 11.3.

- **Disco de registro histórico**. Otra manera de reducir las latencias de escritura es utilizar un disco de registro histórico (es decir, un disco destinado a escribir un registro secuencial) de manera muy parecida a la memoria intermedia RAM no volátil. Todos los accesos al disco de registro histórico son secuenciales, lo que básicamente elimina el tiempo de búsqueda y, así, se pueden escribir simultáneamente varios bloques consecutivos, lo que hace que los procesos de escritura en el disco de registro sean varias veces más rápidos que los procesos de escritura aleatorios. Igual que ocurría anteriormente, también hay que escribir los datos en su ubicación verdadera en el disco, pero el disco de registro puede realizar este proceso de escritura más adelante, sin que el sistema de bases de datos tenga que esperar a que se complete. Además, el disco de registro histórico puede reordenar las operaciones de escritura para reducir el movimiento del brazo. Si el sistema se cae antes de que se haya completado alguna operación de escritura en la ubicación real del disco, cuando el sistema se recupera, lee el disco de registro histórico para averiguar las operaciones de escritura que no se han completado y las lleva a cabo.

  Los sistemas de archivo que soportan los discos de registro histórico mencionados se denominan **sistemas de archivos de diario**. Los sistemas de archivos de diario se pueden implementar incluso sin un disco de registro histórico independiente, guardando los datos y el registro histórico en el mismo disco. Al hacerlo así se reduce el coste económico, a expensas de un menor rendimiento.

  La mayoría de los sistemas de archivos modernos implementan un diario y usan el disco de registro histórico al escribir información interna del sistema de archivos, como la relativa a la asignación de archivos. Los primeros sistemas de archivos permitían reordenar las operaciones de escritura sin emplear disco de registro, y corrían el riesgo de que las estructuras de datos del sistema de archivos del disco se corrompiesen si se producía una caída del sistema. Supóngase, por ejemplo, que un sistema de archivos utiliza una lista enlazada y se inserta un nuevo nodo

al final escribiendo primero los datos del nuevo nodo y actualizando posteriormente el puntero del nodo anterior. Supóngase también que las operaciones de escritura se reordenasen, de forma que se actualizase primero el puntero y que el sistema se cayese antes de escribir el nuevo nodo. El contenido del nodo sería la basura que hubiese anteriormente en el disco, lo que daría lugar a una estructura de datos corrupta.

Para evitar esta posibilidad los primeros sistemas de archivo tenían que realizar una comprobación de consistencia del sistema al reiniciar el sistema para asegurarse de que las estructuras de datos eran consistentes. Si no lo eran, había que tomar medidas adicionales para volver a hacerlas consistentes. Estas comprobaciones daban lugar a grandes retardos en el inicio del sistema después de las caídas, que se agravaron al aumentar la capacidad de las unidades de disco. Los sistemas de archivo de diario permiten un reinicio rápido sin necesidad de esas comprobaciones.

No obstante, las operaciones de escritura realizadas por las aplicaciones no se suelen escribir en el disco del registro histórico. Los sistemas de bases de datos implementan sus propias variantes de registro, que se estudian posteriormente en el Capítulo 17.

11.3 RAID

Los requisitos de almacenamiento de datos de algunas aplicaciones (en particular las aplicaciones Web, las de bases de datos y las multimedia) han crecido tan rápidamente que hace falta un gran número de discos para almacenar sus datos, pese a que la capacidad de los discos también ha aumentado mucho.

Tener gran número de discos en el sistema proporciona oportunidades para mejorar la velocidad a la que se pueden leer o escribir los datos si los discos funcionan en paralelo. También se pueden realizar varias operaciones de lectura o escritura independientes simultáneamente. Además, esta configuración ofrece la posibilidad de mejorar la fiabilidad del almacenamiento de datos, ya que se puede guardar información repetida en varios discos. Por tanto, el fallo de uno de los discos no provoca pérdidas de datos.

Para conseguir mayor rendimiento y fiabilidad se han propuesto varias técnicas de organización de los discos, denominadas colectivamente **disposición redundante de discos independientes** (**RAID**, **Redundant Array of Independent Disks**).

En el pasado, los diseñadores de sistemas consideraron los sistemas de almacenamiento compuestos de varios discos pequeños de bajo coste como alternativa económicamente efectiva al empleo de unidades grandes y caras; el coste por megabyte de los discos más pequeños era menor que el de los de gran tamaño. De hecho, la I de RAID, que ahora significa *independientes*, originalmente representaba *económicos* (*inexpensive*). Hoy en día, sin embargo, todos los discos son de pequeño tamaño y los discos de mayor capacidad tienen un menor coste por megabyte. Los sistemas RAID se utilizan por su mayor fiabilidad y por su mejor rendimiento, más que por motivos económicos. Otro motivo fundamental del empleo de RAID es su mayor facilidad de administración y de operación.

11.3.1 Mejora de la fiabilidad mediante la redundancia

Considérese en primer lugar la fiabilidad. La probabilidad de que falle, al menos, un disco de un conjunto de N discos es mucho más elevada que la posibilidad de que falle un único disco concreto. Supóngase que el tiempo medio entre fallos de un disco dado es de cien mil horas o, aproximadamente, once años. Por tanto, el tiempo medio entre fallos de algún disco de una disposición de cien discos será de $100.000/100 = 1.000$ horas, o cuarenta y dos días, ¡lo que no es mucho! Si sólo se guarda una copia de los datos, cada fallo de disco dará lugar a la pérdida de una cantidad significativa de datos (como ya se estudió en el Apartado 11.2.1). Una frecuencia de pérdida de datos tan elevada resulta inaceptable.

La solución al problema de la fiabilidad es introducir la **redundancia**; es decir, se guarda información adicional que normalmente no se necesita pero que se puede utilizar en caso de fallo de un disco para reconstruir la información perdida. Por tanto, aunque falle un disco, no se pierden datos, por lo que el tiempo medio efectivo entre fallos aumenta, siempre que se cuenten sólo los fallos que dan lugar a pérdida de datos o a su no disponibilidad.

El enfoque más sencillo (pero el más costoso) para la introducción de la redundancia es duplicar todos los discos. Esta técnica se denomina **creación de imágenes** (o, a veces, *creación de sombras*). Cada

unidad lógica consta, por tanto, de dos unidades físicas y cada proceso de escritura se lleva a cabo por duplicado. Si uno de los discos falla se pueden leer los datos del otro. Los datos sólo se perderán si falla el segundo disco antes de que se repare el primero que falló.

El tiempo medio entre fallos (entendiendo fallo como pérdida de datos) de un disco con imagen depende del tiempo medio entre fallos de cada disco y del **tiempo medio de reparación**, que es el tiempo que se tarda (en promedio) en sustituir un disco averiado y en restaurar sus datos. Supóngase que los fallos de los dos discos son *independientes*; es decir, no hay conexión entre el fallo de uno y el del otro. Por tanto, si el tiempo medio entre fallos de un solo disco es de cien mil horas y el tiempo medio de reparación es de diez horas, el **tiempo medio entre pérdidas de datos** de un sistema de discos con imagen es $100.000^2/(2 * 10) = 500 * 10^6$ horas, o ¡57.000 años! (aquí no se entra en detalle en los cálculos; se proporcionan en las referencias de las notas bibliográficas).

Hay que tener en cuenta que la suposición de independencia de los fallos de los discos no resulta válida. Los fallos en el suministro eléctrico y los desastres naturales como los terremotos, los incendios y las inundaciones pueden dar lugar a daños simultáneos de ambos discos. A medida que los discos envejecen, la probabilidad de fallo aumenta, lo que incrementa la posibilidad de que falle el segundo disco mientras se repara el primero. A pesar de todas estas consideraciones, sin embargo, los sistemas de discos con imagen ofrecen una fiabilidad mucho más elevada que los sistemas de disco único. Hoy en día se dispone de sistemas de discos con imagen con un tiempo medio entre pérdidas de datos de entre quinientas mil y un millón de horas, de cincuenta y cinco a ciento diez años.

Los fallos en el suministro eléctrico son una causa especial de preocupación, dado que tienen lugar mucho más frecuentemente que los desastres naturales. Los fallos en el suministro eléctrico no son un problema si no se está realizando ninguna transferencia de datos al disco cuando tienen lugar. Sin embargo, incluso con la creación de imágenes de los discos, si se hallan en curso procesos de escritura en el mismo bloque en ambos discos y el suministro eléctrico falla antes de que se haya acabado de escribir en ambos bloques, los dos pueden quedar en un estado inconsistente. La solución a este problema es escribir primero una copia y luego la otra, de modo que una de las copias siempre sea consistente. Cuando se vuelve a iniciar el sistema tras un fallo en el suministro eléctrico hacen falta algunas acciones adicionales para recuperar los procesos de escritura incompletos. Este asunto se examina en el Ejercicio práctico 11.2.

11.3.2 Mejora del rendimiento mediante el paralelismo

Considérense ahora las ventajas del acceso en paralelo a varios discos. Con la creación de imágenes de los discos la velocidad a la que se pueden procesar las solicitudes de lectura se duplica, dado que pueden enviarse a cualquiera de los discos (siempre y cuando los dos integrantes de la pareja estén operativos, como es el caso habitual). La velocidad de transferencia de cada proceso de lectura es la misma que en los sistemas de disco único, pero el número de procesos de lectura por unidad de tiempo se ha duplicado.

También se puede mejorar la velocidad de transferencia con varios discos **distribuyendo los datos** entre ellos. En su forma más sencilla, la distribución de datos consiste en dividir los bits de cada byte entre varios discos; esto se denomina **distribución en el nivel de bits**. Por ejemplo, si se dispone de una disposición de ocho discos, se puede escribir el bit i de cada byte en el disco i. La disposición de ocho discos puede tratarse como un solo disco con sectores que tienen ocho veces el tamaño normal y, lo que es más importante, que tiene ocho veces la velocidad de acceso habitual. En una organización así cada disco toma parte en todos los accesos (de lectura o de escritura) por lo que el número de accesos que pueden procesarse por segundo es aproximadamente el mismo que con un solo disco, pero cada acceso puede octuplicar el número de datos leídos por unidad de tiempo respecto a un solo disco. La distribución en el nivel de bits puede generalizarse a cualquier número de discos que sea múltiplo o divisor de ocho. Por ejemplo, si se utiliza una disposición de cuatro discos, los bits i y $4 + i$ de cada byte irán al disco i.

La **distribución en el nivel de bloques** reparte cada bloque entre varios discos. Trata la disposición de discos como un único disco de gran capacidad y asigna números lógicos a los bloques; se da por supuesto que los números de bloque comienzan en cero. Con una disposición de n discos, la distribución en el nivel de bloques asigna el bloque lógico i de la disposición de discos al disco $(i \bmod n) + 1$; utiliza

el bloque físico $\lfloor i/n \rfloor$-ésimo del disco para almacenar el bloque lógico i. Por ejemplo, con ocho discos, el bloque lógico 0 se almacena el bloque físico 0 del disco 1, mientras que el bloque lógico 11 se almacena en el bloque físico 1 del disco 4. Si se lee un archivo grande, la distribución en el nivel de bloques busca simultáneamente n bloques en paralelo en los n discos, lo que permite una gran velocidad de transferencia para lecturas de gran tamaño. Cuando se lee un solo bloque, la velocidad de transferencia de datos es igual que con un único disco, pero los restantes $n-1$ discos quedan libres para realizar otras acciones.

La distribución en el nivel de bloques es la forma de distribución de datos más usada. También son posibles otros niveles de distribución, como el de los bytes de cada sector o el de los sectores de cada bloque.

En resumen, en un sistema de discos el paralelismo tiene dos objetivos principales:

1. Equilibrar la carga de varios accesos de pequeño tamaño (accesos a bloques), de manera que aumente la productividad de ese tipo de accesos.
2. Convertir en paralelos los accesos de gran tamaño para que su tiempo de respuesta se reduzca.

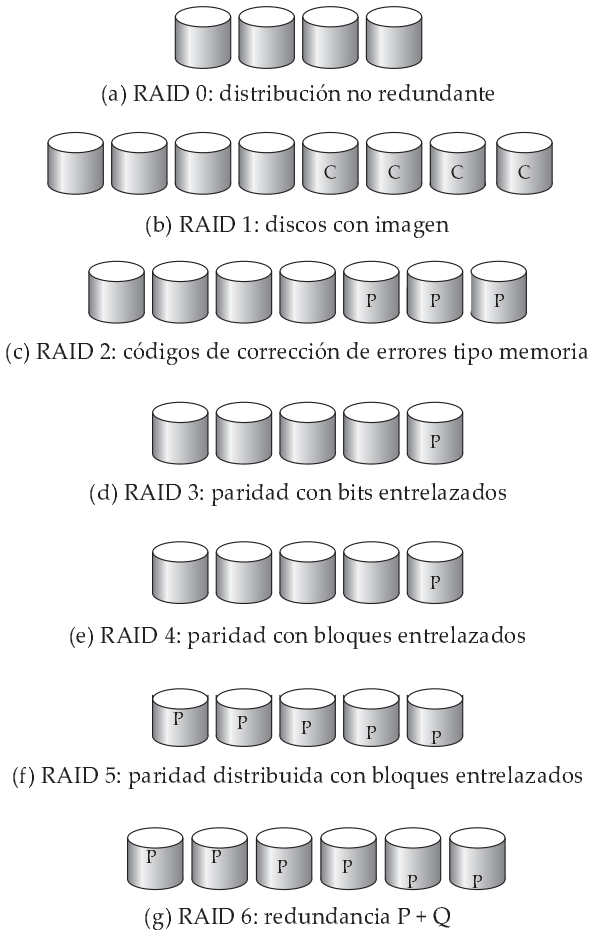
11.3.3 Niveles de RAID

La creación de imágenes proporciona gran fiabilidad pero resulta costosa. La distribución proporciona velocidades de transferencia de datos elevadas, pero no mejora la fiabilidad. Varios esquemas alternativos pretenden proporcionar redundancia a un coste menor combinando la distribución de los discos combinada con los bits de "paridad" (que se describen a continuación). Estos esquemas tienen diferentes compromisos entre el coste y el rendimiento y se clasifican en los denominados **niveles de RAID**, como se muestra en la Figura 11.3 (en la figura, P indica los bits para la corrección de errores y C indica una segunda copia de los datos). La figura muestra cuatro discos de datos para todos los niveles, y los discos adicionales se utilizan para guardar la información redundante para la recuperación en caso de fallo.

- **El nivel 0 de RAID** hace referencia a disposiciones de discos con distribución en el nivel de bloques, pero sin redundancia (como la creación de imágenes o los bits de paridad). La Figura 11.3a muestra una disposición de tamaño cuatro.
- **El nivel 1 de RAID** hace referencia a la creación de imágenes del disco con distribución de bloques. La Figura 11.3b muestra una organización con imagen que guarda cuatro discos de datos.

  Obsérvese que algunos fabricantes emplean el término **nivel 1+0 de RAID** o **nivel 10 de RAID** para referirse a imágenes del disco con distribución, y utilizan el término nivel 1 de RAID para las imágenes del disco sin distribución. Esto último también se puede emplear con disposiciones de discos para dar la apariencia de un disco grande y fiable: si cada disco tiene M bloques, los bloques lógicos 1 a M se almacenan en el disco 1; $M+1$ a $2M$ en el segundo, y así sucesivamente, y se crea una imagen de cada disco.[1]
- **El nivel 2 de RAID**, también conocido como organización de códigos de corrección de errores tipo memoria (memory-style-error-correcting-code organization, ECC), emplea bits de paridad. Hace tiempo que los sistemas de memoria utilizan los bits de paridad para la detección y corrección de errores. Cada byte del sistema de memoria puede tener asociado un bit de paridad que registra si el número de bits del byte que valen uno es par (paridad = 0) o impar (paridad = 1). Si alguno de los bits del byte se deteriora (un uno se transforma en cero y viceversa) la paridad del byte se modifica y, por tanto, no coincide con la paridad guardada. Análogamente, si el bit de paridad guardado se deteriora no coincide con la paridad calculada. Por tanto, el sistema de memoria detecta todos los errores que afectan a un número impar de bits de un mismo byte. Los esquemas

1. Obsérvese que algunos fabricantes usan el término RAID 0+1 para referirse a una versión de RAID que utiliza la distribución para crear una disposición RAID 0, y crea una imagen de esa disposición en otra disposición, con la diferencia respecto de RAID 1 de que, si falla un disco, la disposición RAID 0 que contiene el disco queda inutilizable. La disposición con imagen se puede seguir utilizando, así que no hay pérdida de datos. Esta organización es inferior a RAID 1 cuando falla un disco, ya que los demás discos de la disposición RAID 0 se pueden seguir usando en RAID 1, pero quedan inactivos en RAID 0+1.



(a) RAID 0: distribución no redundante

(b) RAID 1: discos con imagen

(c) RAID 2: códigos de corrección de errores tipo memoria

(d) RAID 3: paridad con bits entrelazados

(e) RAID 4: paridad con bloques entrelazados

(f) RAID 5: paridad distribuida con bloques entrelazados

(g) RAID 6: redundancia P + Q

Figura 11.3 Niveles de RAID.

de corrección de errores guardan dos o más bits adicionales y pueden reconstruir los datos si se deteriora un solo bit.

La idea de los códigos para la corrección de errores se puede utilizar directamente en las disposiciones de discos mediante la distribución de los bytes entre los diferentes discos. Por ejemplo, el primer bit de cada byte puede guardarse en el disco uno, el segundo en el disco dos, etc., hasta que se guarde el octavo bit en el disco ocho; los bits para la corrección de errores se guardan en discos adicionales.

La Figura 11.3c muestra el esquema de nivel 2. Los discos marcados como *P* guardan los bits para la corrección de errores. Si uno de los discos falla, los restantes bits del byte y los de corrección de errores asociados se pueden leer en los demás discos y se pueden utilizar para reconstruir los datos deteriorados. La Figura 11.3c muestra una disposición de tamaño cuatro; obsérvese que el RAID de nivel 2 sólo necesita tres discos adicionales para los cuatro discos de datos, a diferencia del RAID de nivel 1, que necesitaba cuatro discos adicionales.

- **El nivel 3 de RAID**, organización de paridad con bits entrelazados, mejora respecto al nivel 2 al aprovechar que los controladores de disco, a diferencia de los sistemas de memoria, pueden detectar si los sectores se han leído correctamente, por lo que se puede utilizar un solo bit de paridad para la corrección y para la detección de los errores. La idea es la siguiente: si uno de los sectores se deteriora, se sabe exactamente el sector que es y, para cada uno de sus bits, se puede determinar si es un uno o un cero calculando la paridad de los bits correspondientes de los sectores de los demás discos. Si la paridad de los bits restantes es igual que la paridad guardada, el bit perdido es un cero; en caso contrario, es un uno.

El nivel 3 de RAID es tan bueno como el nivel 2, pero resulta menos costoso en cuanto al número de discos adicionales (sólo tiene un disco adicional), por lo que el nivel 2 no se utiliza en la práctica. La Figura 11.3d muestra el esquema de nivel 3.

El nivel 3 de RAID tiene dos ventajas respecto al nivel 1. Sólo necesita un disco de paridad para varios discos normales, mientras que el nivel 1 necesita un disco imagen por cada disco, por lo que se reduce la necesidad de almacenamiento adicional. Dado que los procesos de lectura y de escritura de cada byte se distribuyen por varios discos, con la distribución de los datos en N fracciones, la velocidad de transferencia para la lectura o la escritura de un solo bloque multiplica por N la de una organización RAID de nivel 1 que emplee la distribución entre N discos. Por otro lado, el nivel 3 de RAID permite un menor número de operaciones de E/S por segundo, ya que todos los discos tienen que participar en cada solicitud de E/S.

- **El nivel 4 de RAID**, organización de paridad con bloques entrelazados, emplea la distribución del nivel de bloques, como RAID 0, y, además, guarda un bloque de paridad en un disco independiente para los bloques correspondientes de los otros N discos. Este esquema se muestra gráficamente en la Figura 11.3e. Si falla uno de los discos, se puede utilizar el bloque de paridad con los bloques correspondientes de los demás discos para restaurar los bloques del disco averiado.

  La lectura de un bloque sólo accede a un disco, lo que permite que los demás discos procesen otras solicitudes. Por tanto, la velocidad de transferencia de datos de cada acceso es menor, pero se pueden ejecutar en paralelo varios accesos de lectura, lo que produce una mayor velocidad global de E/S. Las velocidades de transferencia para los procesos de lectura de gran tamaño son elevadas, dado que se pueden leer todos los discos en paralelo; los procesos de escritura de gran tamaño también tienen velocidades de transferencia elevadas, dado que los datos y la paridad pueden escribirse en paralelo.

  Los procesos de escritura independientes de pequeño tamaño, por otro lado, no pueden realizarse en paralelo. La operación de escritura de un bloque tiene que acceder al disco en el que se guarda ese bloque, así como al disco de paridad, ya que hay que actualizar el bloque de paridad. Además, hay que leer tanto el valor anterior del bloque de paridad como el del bloque que se escribe para calcular la nueva paridad. Por tanto, un solo proceso de escritura necesita cuatro accesos a disco: dos para leer los dos bloques antiguos y otros dos para escribir los dos nuevos.

- **El nivel 5 de RAID**, paridad distribuida con bloques entrelazados, mejora respecto al nivel 4 al dividir los datos y la paridad entre todos los $N + 1$ discos, en vez de guardar los datos en N discos y la paridad en uno solo. En el nivel 5 todos los discos pueden atender a las solicitudes de lectura, a diferencia del nivel 4 de RAID, en el que el disco de paridad no puede participar, por lo que el nivel 5 aumenta el número total de solicitudes que pueden atenderse en una cantidad de tiempo dada. En cada conjunto de N bloques lógicos, uno de los discos guarda la paridad y los otros N guardan los bloques.

  La Figura 11.3f muestra esta configuración, en la que las P están distribuidas entre todos los discos. Por ejemplo, con una disposición de cinco discos, el bloque de paridad, marcado como Pk, para los bloques lógicos $4k, 4k + 1, 4k + 2, 4k + 3$ se guarda en el disco $(k \bmod 5) + 1$; los bloques correspondientes de los otros cuatro discos guardan los cuatro bloques de datos $4k$ to $4k + 3$. La siguiente tabla indica la manera en que se disponen los primeros veinte bloques, numerados de 0 to 19, y sus bloques de paridad. El patrón mostrado se repite para los siguientes bloques.

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4

Obsérvese que un bloque de paridad no puede guardar la paridad de los bloques del mismo disco, ya que un fallo del disco supondría la pérdida de los datos y de la paridad y, por tanto, no

sería recuperable. El nivel 5 incluye al nivel 4, dado que ofrece mejor rendimiento de lectura y de escritura por el mismo coste, por lo que el nivel 4 no se utiliza en la práctica.

- **El nivel 6 de RAID**, también denominado esquema de redundancia P+Q, es muy parecido al nivel 5, pero guarda información redundante adicional para la protección contra los fallos de disco múltiples. En lugar de utilizar la paridad, el nivel 6 utiliza códigos para la corrección de errores como los de Reed–Solomon (véanse las notas bibliográficas). En el esquema de la Figura 11.3g se guardan dos bits de datos redundantes por cada cuatro bits de datos (a diferencia del único bit de paridad del nivel 5) y el sistema puede tolerar dos fallos del disco.

Finalmente, se debe destacar que se han propuesto varias modificaciones a los esquemas RAID básicos aquí descritos, y que los diferentes fabricantes emplean su propia terminología para las variantes.

11.3.4 Elección del nivel RAID adecuado

Los factores que se deben tener en cuenta para elegir el nivel RAID son:

- El coste económico de los requisitos adicionales de almacenamiento en disco.
- Los requisitos de rendimiento en términos de número de operaciones de E/S.
- El rendimiento cuando falla un disco.
- El rendimiento durante la reconstrucción (esto es, mientras los datos del disco averiado se reconstruyen en un disco nuevo).

El tiempo que se tarda en reconstruir los datos que contenía el disco averiado puede ser significativo, y varía con el nivel RAID utilizado. La reconstrucción resulta más sencilla para el RAID de nivel 1, ya que se pueden copiar los datos de otro disco; para los otros niveles hay que acceder a todos los demás discos de la disposición para reconstruir los datos del disco averiado. El **rendimiento de la reconstrucción** de un sistema RAID puede ser un factor importante si se necesita que los datos estén disponibles ininterrumpidamente, como ocurre en los sistemas de bases de datos de alto rendimiento. Además, dado que el tiempo de reconstrucción puede suponer una parte importante del tiempo de reparación, el rendimiento de la reconstrucción también influye en el tiempo medio entre pérdidas de datos.

El nivel 0 de RAID se utiliza en aplicaciones de alto rendimiento en las que la seguridad de los datos no es crítica. Dado que los niveles 2 y 4 de RAID se subsumen en el 3 y en el 5, respectivamente, la elección de nivel RAID se limita a los niveles restantes. La distribución de bits (nivel 3) es inferior a la distribución de bloques (nivel 5) ya que ésta permite velocidades de transferencia de datos similares para las transferencias de gran tamaño y emplea menos discos para las pequeñas. Para las transferencias de pequeño tamaño, el tiempo de acceso al disco es el factor dominante, por lo que disminuye la ventaja de las operaciones de lectura paralelas. De hecho, el nivel 3 puede presentar un peor rendimiento que el nivel 5 para transferencias de pequeño tamaño, ya que sólo se completan cuando se han encontrado los sectores correspondientes de todos los discos; la latencia media de la disposición de discos se comporta, por tanto, de forma muy parecida a la máxima latencia en el caso de un solo disco, lo que anula las ventajas de la mayor velocidad de transferencia. Muchas implementaciones RAID no soportan actualmente el nivel 6, pero ofrece mayor fiabilidad que el nivel 5 y se puede utilizar en aplicaciones en las que la seguridad de los datos sea muy importante.

La elección entre el nivel 1 y el nivel 5 de RAID es más difícil de tomar. El nivel 1 se utiliza mucho en aplicaciones como el almacenamiento de archivos de registro histórico en sistemas de bases de datos, ya que ofrece el mejor rendimiento para escritura. El nivel 5 tiene menos sobrecarga de almacenamiento que el nivel 1, pero presenta una mayor sobrecarga temporal en las operaciones de escritura. Para las aplicaciones en las que los datos se leen con frecuencia y se escriben raramente, el nivel 5 es la opción preferida.

La capacidad de almacenamiento en disco ha aumentado anualmente más del cincuenta por ciento durante muchos años, y el coste por byte ha disminuido a la misma velocidad. En consecuencia, para muchas aplicaciones existentes de bases de datos con requisitos de almacenamiento moderados, el coste económico del almacenamiento adicional en disco necesario para la creación de imágenes ha pasado a

ser relativamente pequeño (sin embargo, el coste económico adicional, sigue siendo un aspecto significativo para las aplicaciones de almacenamiento intensivo como el almacenamiento de datos de vídeo). La velocidad de acceso ha mejorado mucho más lentamente (del orden de un factor tres en diez años), mientras que el número de operaciones E/S necesarias por segundo se ha incrementado enormemente, especialmente para los servidores de aplicaciones Web.

El nivel 5 de RAID que incrementa el número de operaciones E/S necesarias para escribir un solo bloque lógico, sufre una penalización de tiempo significativa en términos del rendimiento de escritura. El nivel 1 de RAID es, por tanto, la elección adecuada para muchas aplicaciones con requisitos moderados de almacenamiento y grandes de E/S.

Los diseñadores de sistemas RAID también tienen que tomar otras decisiones. Por ejemplo, la cantidad de discos que habrá en la disposición y los bits que debe proteger cada bit de paridad. Cuantos más discos haya en la disposición, mayores serán las velocidades de transferencia de datos, pero el sistema será más caro. Cuantos más bits proteja cada bit de paridad, menor será la necesidad adicional de espacio debida a los bits de paridad, pero habrá más posibilidades de que falle un segundo disco antes de que el primer disco en averiarse esté reparado y de que eso dé lugar a pérdidas de datos.

11.3.5 Aspectos hardware

Otro aspecto que debe tenerse en cuenta en la elección de implementaciones RAID es el hardware. RAID se puede implementar sin cambios en el nivel hardware, modificando sólo el software. Estas implementaciones se conocen como **RAID software**. Sin embargo, se pueden obtener ventajas significativas al crear hardware específico para dar soporte a RAID, que se describen a continuación; los sistemas con soporte hardware especial se denominan sistemas **RAID hardware**.

Las implementaciones RAID hardware pueden utilizar RAM no volátil para registrar las operaciones de escritura antes de llevarlas a cabo. En caso de fallo de corriente, cuando el sistema se restaura, recupera la información relativa a las operaciones de escritura incompletas de la memoria RAM no volátil y completa esas operaciones. Sin ese soporte hardware, hay que llevar a cabo trabajo adicional para detectar los bloques que se han escrito parcialmente antes del fallo de corriente (véase el Ejercicio práctico 11.2).

Algunas implementaciones RAID permiten el **intercambio en caliente**; esto es, se pueden retirar los discos averiados y sustituirlos por otros nuevos sin desconectar la corriente. El intercambio en caliente reduce el tiempo medio de reparación, ya que los cambios de disco no necesitan esperar a que se pueda apagar el sistema. De hecho, hoy en día muchos sistemas críticos trabajan con un horario 24×7; esto es, funcionan 24 horas al día, 7 días a la semana, lo que no ofrece ningún momento para apagar el sistema y cambiar los discos averiados. Además, muchas implementaciones RAID asignan un disco de repuesto para cada disposición (o para un conjunto de disposiciones de disco). Si falla algún disco, el disco de repuesto se utiliza como sustituto de manera inmediata. En consecuencia, el tiempo medio de reparación se reduce notablemente, lo que minimiza la posibilidad de pérdida de datos. El disco averiado se puede reemplazar cuando resulte más conveniente.

La fuente de alimentación, el controlador de disco o, incluso, la interconexión del sistema en un sistema RAID pueden ser el punto de fallo que detiene el funcionamiento del sistema RAID. Para evitar esta posibilidad, las buenas implementaciones RAID tienen varias fuentes de alimentación redundantes (con baterías de respaldo que les permiten seguir funcionando aunque se corte la corriente). Estos sistemas RAID tienen varias interfaces de disco y varias interconexiones para conectar el sistema RAID con el sistema informático (o con la red de sistemas informáticos). En consecuencia, el fallo de un solo componente no detiene el funcionamiento del sistema RAID.

11.3.6 Otras aplicaciones de RAID

Los conceptos de RAID se han generalizado a otros dispositivos de almacenamiento, como los conjuntos de cintas, e incluso a la transmisión de datos por radio. Cuando se aplican a los conjuntos de cintas, las estructuras RAID pueden recuperar datos aunque se haya dañado una de las cintas de la disposición. Cuando se aplican a la transmisión de datos, cada bloque de datos se divide en unidades menores y se transmite junto con una unidad de paridad; si, por algún motivo, no se recibe alguna de las unidades, se puede reconstruir a partir del resto.

11.4 Almacenamiento terciario

Puede que en los sistemas de bases de datos de gran tamaño parte de los datos tenga que residir en un almacenamiento terciario. Los dos medios de almacenamiento terciario más habituales son los discos ópticos y las cintas magnéticas.

11.4.1 Discos ópticos

Los discos compactos han sido un medio popular de distribución de software, datos multimedia como el sonido y las imágenes, y otra información editada de manera electrónica. Tienen una capacidad de almacenamiento de 640 a 700 megabytes y resultan baratos de producir en masa. Actualmente, los discos de vídeo digital (DVD, Digital Video Disk) han reemplazado a los discos compactos en las aplicaciones que necesitan grandes cantidades de datos. Los discos de formato DVD-5 pueden almacenar 4,7 gigabytes de datos (en una capa de grabación), mientras que los discos de formato DVD-9 pueden almacenar 8,5 gigabytes de datos (en dos capas de grabación). La grabación en las dos caras del disco ofrece capacidades mayores incluso; los formatos DVD-10 y DVD-18, que son las versiones de doble cara de DVD-5 y DVD-9, pueden almacenar 9,4 y 17 gigabytes, respectivamente. Los formatos más recientes, denominados *HD-DVD* y *Blu-Ray DVD* tienen una capacidad significativamente mayor: los discos HD-DVD pueden almacenar de 12 a 30 gigabytes por disco, mientras que los discos Blu-Ray DVD de 25 a 50 gigabytes por disco. Se espera que estén disponibles entre 2005 y 2006.

Las unidades de CD y de DVD presentan tiempos de búsqueda mucho mayores (100 milisegundos es un valor habitual) que las unidades de disco magnético, debido a que el dispositivo de cabezas es más pesado. Las velocidades rotacionales suelen ser menores, aunque las unidades de CD y de DVD más rápidas alcanzan alrededor de tres mil revoluciones por minuto, que son comparables a las velocidades de los discos magnéticos de gama baja. Las velocidades rotacionales de las unidades de CD se correspondían inicialmente con las normas de los CD de sonido; y las velocidades de las unidades de DVD, con las de los DVD de vídeo, pero las unidades actuales rotan a varias veces la velocidad indicada por las normas. Las velocidades de transferencia de datos son algo menores que para los discos magnéticos. Las unidades de CD actuales leen entre 3 y 6 megabytes por segundo; y las de DVD, de 8 a 20 megabytes por segundo. Al igual que las unidades de discos magnéticos, los discos ópticos almacenan más datos en las pistas exteriores y menos en las interiores. La velocidad de transferencia de las unidades ópticas se caracteriza por $n\times$, que significa que la unidad soporta transferencias a n veces la velocidad indicada por la norma; las velocidades de $50\times$ para CD y de $16\times$ para DVD son frecuentes hoy en día.

Las versiones de los discos ópticos en las que sólo se puede escribir una vez (CD-R, DVD-R y DVD+R) son populares para la distribución de datos y, en especial, para el almacenamiento de datos con fines de archivo, debido a que tienen una gran capacidad, una vida más larga que los discos magnéticos y pueden retirarse de la unidad y almacenarse en un lugar remoto. Dado que no pueden sobrescribirse, se pueden utilizar para almacenar información que no se deba modificar, como los registros de auditoría. Las versiones en las que se puede escribir más de una vez (CD-RW, DVD-RW, DVD+RW y DVD-RAM) también se emplean para archivo.

Los **cambiadores de discos** (**jukebox**) son dispositivos que guardan gran número de discos ópticos (hasta varios cientos) y los cargan automáticamente a petición de los usuarios en unas cuantas unidades (usualmente entre una y diez). La capacidad de almacenamiento agregada de estos sistemas puede ser de muchos terabytes. Cuando se accede a un disco, un brazo mecánico lo carga en la unidad desde una estantería (antes hay que volver a colocar en la estantería el disco que se hallara en la unidad). El tiempo de carga y descarga suele ser del orden de segundos (mucho mayor que los tiempos de acceso a disco).

11.4.2 Cintas magnéticas

Aunque las cintas magnéticas son relativamente permanentes y pueden albergar grandes volúmenes de datos, resultan lentas en comparación con los discos magnéticos y ópticos. Lo que es aún más importante, la cinta magnética está limitada al acceso secuencial. Por tanto, no puede proporcionar acceso aleatorio para los requisitos de almacenamiento secundario, aunque históricamente, antes del uso de los discos magnéticos, las cintas se utilizaran como medio de almacenamiento secundario.

Las cintas se emplean principalmente para copias de seguridad, para el almacenamiento de la información poco utilizada y como medio sin conexión para la transferencia de información de un sistema a otro. Las cintas también se utilizan para almacenar grandes volúmenes de datos, como los datos de vídeo o de imágenes, a los que no es necesario acceder rápidamente o que son tan voluminosos que su almacenamiento en disco magnético resultaría demasiado caro.

Cada cinta se guarda en una bobina y se enrolla o desenrolla por delante de una cabeza de lectura y escritura. El desplazamiento hasta el punto correcto de la cinta puede tardar segundos o, incluso, minutos, en vez de milisegundos; una vez en posición, sin embargo, las unidades de cinta pueden escribir los datos con densidades y velocidades que se aproximan a las de las unidades de disco. La capacidad varía en función de la longitud y de la anchura de la cinta y de la densidad con la que la cabeza pueda leer y escribir. El mercado ofrece actualmente una amplia variedad de formatos de cinta. La capacidad disponible actualmente varía de unos pocos gigabytes con el formato DAT (**Digital Audio Tape**, **cinta de audio digital**), de 10 a 40 gigabytes con el formato DLT (**Digital Linear Tape**, **cinta lineal digital**), de 100 gigabytes en adelante con el formato **Ultrium**, hasta 330 gigabytes con los formatos de cinta de **exploración helicoidal de Ampex**. Las velocidades de transferencia de datos llegan a las decenas de megabytes por segundo.

Las unidades de cinta son bastante fiables, y los buenos sistemas de unidades de cinta realizan una lectura de los datos recién escritos para asegurarse de que se han grabado correctamente. Sin embargo, las cintas presentan límites en cuanto al número de veces que se pueden leer o escribir con fiabilidad.

Los **cambiadores de cintas**, al igual que los cambiadores de discos ópticos, tienen gran número de cintas y unas cuantas unidades en las que se pueden montar esas cintas; se utilizan para guardar grandes volúmenes de datos, que pueden llegar a varios petabytes (10^{15} bytes) con tiempos de acceso que varían entre los segundos y unos cuantos minutos. Entre las aplicaciones que necesitan estas enormes cantidades de datos están los sistemas de imágenes que reunen los datos de los satélites de teledetección y las grandes videotecas de las emisoras de televisión.

Algunos formatos de cinta (como el formato Accelis) soportan menores tiempos de búsqueda (del orden de decenas de segundos) y están pensadas para las aplicaciones que recuperan información mediante cambiadores de cintas. La mayor parte del resto de los formatos de cinta proporcionan mayores capacidades, a cambio de un acceso más lento; estos formatos resultan idóneos para las copias de seguridad de los datos, en las que las búsquedas rápidas no son importantes.

Las unidades de cinta no han podido competir con la enorme mejora de capacidad de las unidades de disco y la correspondiente reducción del coste de almacenamiento. Aunque el coste de las cintas es bajo, el de las unidades y las bibliotecas de cintas es significativamente superior que el coste de las unidades de disco: una biblioteca de cintas capaz de almacenar algunos terabytes puede costar decenas de millares de euros. Las copias de seguridad en unidades de disco se han convertido en una alternativa rentable a las copias en cinta para gran número aplicaciones.

11.5 Acceso al almacenamiento

Cada base de datos se corresponde con varios archivos diferentes que el sistema operativo subyacente mantiene. Esos archivos residen permanentemente en los discos, con copias de seguridad en cinta. Cada archivo está dividido en unidades de almacenamiento de longitud constante denominadas **bloques**, que son las unidades de asignación de almacenamiento y de transferencia de datos. En el Apartado 11.6 se estudian varias maneras de organizar lógicamente los datos en archivos.

Cada bloque puede contener varios elementos de datos. El conjunto exacto de elementos de datos que contiene cada bloque viene determinado por la forma de organización física de los datos que se utilice (véase el Apartado 11.6). En nuestro caso se supone que ningún elemento de datos ocupa dos o más bloques. Esta suposición es realista para la mayor parte de las aplicaciones de procesamiento de datos, como el ejemplo bancario propuesto.

Uno de los principales objetivos del sistema de bases de datos es minimizar el número de transferencias de bloques entre el disco y la memoria. Una manera de reducir el número de accesos al disco es mantener en la memoria principal tantos bloques como sea posible. El objetivo es maximizar la posibilidad de que, cuando se acceda a un bloque, ya se encuentre en la memoria principal y, por tanto, no se necesite acceder al disco.

Dado que no resulta posible mantener en la memoria principal todos los bloques, hay que gestionar la asignación del espacio allí disponible para su almacenamiento. La **memoria intermedia** (buffer) es la parte de la memoria principal disponible para el almacenamiento de las copias de los bloques del disco. Siempre se guarda en el disco una copia de cada bloque, pero esta copia puede ser una versión del bloque más antigua que la de la memoria intermedia. El subsistema responsable de la asignación del espacio de la memoria intermedia se denomina **gestor de la memoria intermedia**.

11.5.1 Gestor de la memoria intermedia

Los programas de los sistemas de bases de datos formulan solicitudes (es decir, llamadas) al gestor de la memoria intermedia cuando necesitan bloques del disco. Si el bloque ya se encuentra en la memoria intermedia, el gestor pasa al solicitante la dirección del bloque en la memoria principal. Si el bloque no se halla en la memoria intermedia, asigna en primer lugar espacio al bloque en la memoria intermedia, descartando algún otro, si hace falta, para hacer sitio al nuevo. El bloque descartado sólo se vuelve a escribir en el disco si se ha modificado desde la última vez que se escribió. A continuación, el gestor de la memoria intermedia lee el bloque solicitado en el disco, lo escribe en la memoria intermedia y pasa la dirección del bloque en la memoria principal al solicitante. Las acciones internas del gestor de la memoria intermedia resultan transparentes para los programas que formulan solicitudes de bloques de disco.

Si se está familiarizado con los conceptos de los sistemas operativos, se observará que el gestor de la memoria intermedia no parece ser más que un gestor de memoria virtual, como los que se hallan en la mayor parte de los sistemas operativos. Una diferencia radica en que el tamaño de las bases de datos puede ser mucho mayor que el espacio de direcciones de hardware de la máquina, por lo que las direcciones de memoria no resultan suficientes para direccionar todos los bloques del disco. Además, para dar un buen servicio al sistema de bases de datos, el gestor de la memoria intermedia debe utilizar técnicas más complejas que los esquemas habituales de gestión de la memoria virtual:

- **Estrategia de sustitución**. Cuando no queda espacio libre en la memoria intermedia hay que eliminar un bloque de ésta antes de que se pueda escribir otro nuevo. La mayor parte de los sistemas operativos utilizan un esquema de **menos recientemente utilizado** (**Least Recently Used**, **LRU**), en el que se vuelve a escribir en el disco y se elimina de la memoria intermedia el bloque al que se ha hecho referencia menos recientemente. Este sencillo enfoque se puede mejorar para las aplicaciones de bases de datos.
- **Bloques clavados**. Para que el sistema de bases de datos pueda recuperarse de las caídas del sistema (Capítulo 17) hay que restringir las ocasiones en que los bloques se pueden volver a escribir en el disco. Por ejemplo, la mayor parte de los sistemas de recuperación exigen que no se escriban en disco los bloques mientras se esté procediendo a su actualización. Se dice que los bloques que no se permite que se vuelvan a escribir en el disco están **clavados**. Aunque muchos sistemas operativos no permiten trabajar con bloques clavados, esta característica resulta fundamental para los sistemas de bases de datos resistentes a las caídas.
- **Salida forzada de los bloques**. Hay situaciones en las que hace falta volver a escribir los bloques en el disco, aunque no se necesite el espacio de memoria intermedia que ocupan. Este proceso de escritura se denomina **salida forzada** del bloque. Se verá el motivo de las salidas forzadas en el Capítulo 17; en resumen, el contenido de la memoria principal y, por tanto, el de la memoria intermedia se pierden en las caídas, mientras que los datos del disco suelen sobrevivir a ellas.

11.5.2 Políticas para la sustitución de la memoria intermedia

El objetivo de las estrategias de sustitución de los bloques de la memoria intermedia es minimizar los accesos al disco. En los programas de propósito general no resulta posible predecir con precisión a qué bloques se hará referencia. Por tanto, el sistema operativo utiliza la pauta anterior de referencias a bloques para predecir las futuras. La suposición que suele hacerse es que es probable que se vuelva a hacer referencia a los bloques a los que se ha hecho referencia recientemente. Por tanto, si hay que sustituir

```
for each tupla b de prestatario do
    for each tupla c de cliente do
        if b[nombre_cliente] = c[nombre_cliente]
        then begin
                sea x una tupla definida de la manera siguiente:
                x[nombre_cliente] := p[nombre_cliente]
                x[número_préstamo] := p[número_préstamo]
                x[calle_cliente] := c[calle_cliente]
                x[ciudad_cliente] := c[ciudad_cliente]
                incluir la tupla x como parte del resultado de prestatario ⋈ cliente
             end
    end
end
```

Figura 11.4 Procedimiento para calcular la reunión.

un bloque, se sustituye el bloque al que se ha hecho referencia menos recientemente. Este enfoque se denomina **esquema de sustitución de bloques utilizados menos recientemente** (Least Recently Used, LRU).

LRU es un esquema de sustitución aceptable para los sistemas operativos. Sin embargo, los sistemas de bases de datos pueden predecir la pauta de referencias futuras con más precisión que los sistemas operativos. Las peticiones de los usuarios al sistema de bases de datos comprenden varias etapas. El sistema de bases de datos suele poder determinar con antelación los bloques que se necesitarán examinando cada una de las etapas necesarias para llevar a cabo la operación solicitada por el usuario. Por tanto, a diferencia de los sistemas operativos, que deben confiar en el pasado para predecir el futuro, los sistemas de bases de datos pueden tener información relativa, al menos, al futuro a corto plazo.

Para ilustrar la manera en que la información sobre el futuro acceso a los bloques permite mejorar la estrategia LRU considérese el procesamiento de la expresión del álgebra relacional

$$prestatario \bowtie cliente$$

Supóngase que la estrategia escogida para procesar esta solicitud viene dada por el programa en seudocódigo mostrado en la Figura 11.4 (se estudiarán otras estrategias más eficientes en el Capítulo 13).

Supóngase que las dos relaciones de este ejemplo se guardan en archivos diferentes. En este ejemplo se puede ver que, una vez que se haya procesado la tupla de *prestatario*, no vuelve a ser necesaria. Por tanto, una vez completado el procesamiento de un bloque completo de tuplas de *prestatario*, ese bloque ya no se necesita en la memoria principal, aunque se haya utilizado recientemente. Deben darse instrucciones al gestor de la memoria intermedia para que libere el espacio ocupado por el bloque de *prestatario* en cuanto se haya procesado la última tupla. Esta estrategia de gestión de la memoria intermedia se denomina estrategia **de extracción inmediata**.

Considérense ahora los bloques que contienen las tuplas de *cliente*. Hay que examinar cada bloque de las tuplas *cliente* una vez por cada tupla de la relación *prestatario*. Cuando se completa el procesamiento del bloque *cliente*, se sabe que no se accederá nuevamente a él hasta que no se hayan procesado todos los demás bloques de *cliente*. Por tanto, el bloque de *cliente* al que se haya hecho referencia más recientemente será el último bloque al que se vuelva a hacer referencia, y el bloque de *cliente* al que se haya hecho referencia menos recientemente será el bloque al que se vuelva a hacer referencia a continuación. Este conjunto de suposiciones es justo el contrario del que forma la base de la estrategia LRU. En realidad, la estrategia óptima de sustitución de bloques para el procedimiento anterior es la **estrategia más recientemente utilizado** (**Most Recently Used**, **MRU**). Si hay que eliminar de la memoria intermedia un bloque de *cliente*, la estrategia MRU escoge el bloque utilizado más recientemente (los bloques en uso no se pueden eliminar).

Para que la estrategia MRU funcione correctamente en el ejemplo propuesto, el sistema debe clavar el bloque de *cliente* que se esté procesando. Después de que se haya procesado la última tupla de *cliente* el bloque se desclava y pasa a ser el bloque utilizado más recientemente.

Además de utilizar la información que pueda tener el sistema respecto de la solicitud que se esté procesando, el gestor de la memoria intermedia puede utilizar información estadística relativa a la probabilidad de que una solicitud haga referencia a una relación concreta. Por ejemplo, el diccionario de datos (como se verá con detalle en el Apartado 11.8) que realiza un seguimiento del esquema lógico de las relaciones y de la información de su almacenamiento físico es una de las partes de la base de datos a la que se tiene acceso con mayor frecuencia. Por tanto, el gestor de la memoria intermedia debe intentar no eliminar de la memoria principal los bloques del diccionario de datos, a menos que se vea obligado a hacerlo. En el Capítulo 12 se estudian los índices de los archivos. Dado que puede que se acceda más frecuentemente al índice del archivo que al propio archivo, el gestor de la memoria intermedia no deberá, en general, eliminar de la memoria principal los bloques del índice si se dispone de alternativas.

La estrategia ideal para la sustitución de bloques necesita información sobre las operaciones de la bases de datos (las que se estén realizando y las que se vayan a realizar en el futuro). No se conoce una sola estrategia que sea adecuada para todas las situaciones posibles. En realidad, un número sorprendentemente grande de bases de datos utiliza LRU, a pesar de sus defectos. Las preguntas prácticas y los ejercicios exploran estrategias alternativas.

La estrategia utilizada por el gestor de la memoria intermedia para la sustitución de los bloques se ve influida por factores distintos del momento en que se volverá a hacer referencia a cada bloque. Si el sistema está procesando de manera concurrente las solicitudes de varios usuarios, puede que el subsistema para el control de concurrencia (Capítulo 16) tenga que posponer ciertas solicitudes para asegurar la conservación de la consistencia de la base de datos. Si se proporciona al gestor de la memoria intermedia información del subsistema de control de concurrencia que indique las solicitudes que se posponen, puede utilizar esa información para modificar su estrategia de sustitución de los bloques. Concretamente, los bloques que necesiten las solicitudes activas (no pospuestas) pueden conservarse en la memoria intermedia a expensas de los que necesiten las solicitudes pospuestas.

El subsistema para la recuperación de caídas (Capítulo 17) impone severas restricciones a la sustitución de bloques. Si se ha modificado un bloque, no se permite que el gestor de la memoria intermedia vuelva a copiar al disco la versión nueva del bloque existente en la memoria intermedia, dado que eso destruiría la versión anterior. Por el contrario, el gestor de bloques debe solicitar permiso del subsistema para la recuperación de caídas antes de escribir cada bloque. Puede que el subsistema para la recuperación de caídas exija que se fuerce la salida de otros bloques antes de conceder autorización al gestor de la memoria intermedia para que escriba el bloque solicitado. En el Capítulo 17 se define con precisión la interacción entre el gestor de la memoria intermedia y el subsistema para la recuperación de caídas.

11.6 Organización de los archivos

Los **archivos** se organizan lógicamente como secuencias de registros. Esos registros se corresponden con los bloques del disco. Los archivos constituyen un elemento fundamental de los sistemas operativos, por lo que se supone la existencia de un *sistema de archivos* subyacente. Hay que tomar en consideración diversas maneras de representar los modelos lógicos de datos en términos de los archivos.

Aunque los bloques son de un tamaño fijo determinado por las propiedades físicas del disco y por el sistema operativo, el tamaño de los registros varía. En las bases de datos relacionales, las tuplas de las diferentes relaciones suelen ser de tamaño diferente.

Un enfoque de la correspondencia entre base de datos y archivos es utilizar varios archivos y guardar los registros de la misma longitud en un mismo archivo. Una alternativa es estructurar los archivos de modo que puedan aceptar registros de longitudes diferentes; no obstante, los archivos con registros de longitud fija son más sencillos de implementar que los que tienen registros de longitud variable. Muchas de las técnicas empleadas para los primeros pueden aplicarse a los de longitud variable. Por tanto, se comienza por tomar en consideración los archivos con registros de longitud fija.

11.6.1 Registros de longitud fija

A manera de ejemplo, considérese un archivo con registros de *cuentas* de la base de datos bancaria. Cada registro de este archivo se define (en pseudocódigo) de la manera siguiente:

```
type depósito = record
                    número_cuenta char(10);
                    nombre_sucursal char(22);
                    saldo numeric(12,2);
                end
```

Si se supone que cada carácter ocupa un byte y que los valores de tipo numeric(12,2) ocupan ocho bytes, el registro de *cuenta* tiene cuarenta bytes de longitud. Un enfoque sencillo es utilizar los primeros cuarenta bytes para el primer registro, los cuarenta bytes siguientes para el segundo, y así sucesivamente (Figura 11.5). Sin embargo, hay dos problemas con este sencillo enfoque:

1. Resulta difícil borrar registros de esta estructura. Hay que rellenar el espacio ocupado por el registro que se va a borrar con algún otro registro del archivo, o tener alguna manera de marcar los registros borrados para poder pasarlos por alto.
2. A menos que el tamaño de los bloques sea múltiplo de cuarenta (lo que resulta improbable), algún registro se saltará los límites de los bloques. Es decir, parte del registro se guardará en un bloque y parte en otro. Harán falta, por tanto, dos accesos a bloques para leer o escribir esos registros.

Cuando se borra un registro, se puede desplazar el situado a continuación al espacio ocupado que ocupaba el registro borrado y hacer lo mismo con los demás, hasta que todos los registros situados a continuación del borrado se hayan desplazado hacia delante (Figura 11.6). Este tipo de enfoque necesita desplazar gran número de registros. Puede que fuera más sencillo desplazar simplemente el último registro del archivo al espacio ocupado por el registro borrado (Figura 11.7).

No resulta deseable desplazar los registros para que ocupen el espacio liberado por los registros borrados, ya que para hacerlo se necesitan accesos adicionales a los bloques. Dado que las operaciones de inserción tienden a ser más frecuentes que las de borrado, resulta aceptable dejar libre el espacio

registro 0	C-102	Navacerrada	400
registro 1	C-305	Collado Mediano	350
registro 2	C-215	Becerril	700
registro 3	C-101	Centro	500
registro 4	C-222	Moralzarzal	700
registro 5	C-201	Navacerrada	900
registro 6	C-217	Galapagar	750
registro 7	C-110	Centro	600
registro 8	C-218	Navacerrada	700

Figura 11.5 Archivo que contiene los registros de *cuenta*.

registro 0	C-102	Navacerrada	400
registro 1	C-305	Collado Mediano	350
registro 3	C-101	Centro	500
registro 4	C-222	Moralzarzal	700
registro 5	C-201	Navacerrada	900
registro 6	C-217	Galapagar	750
registro 7	C-110	Centro	600
registro 8	C-218	Navacerrada	700

Figura 11.6 El archivo de la Figura 11.5 con el registro 2 borrado y todos los registros desplazados.

registro 0	C-102	Navacerrada	400
registro 1	C-305	Collado Mediano	350
registro 8	C-218	Navacerrada	700
registro 3	C-101	Centro	500
registro 4	C-222	Moralzarzal	700
registro 5	C-201	Navacerrada	900
registro 6	C-217	Galapagar	750
registro 7	C-110	Centro	600

Figura 11.7 El archivo de la Figura 11.5 con el registro 2 borrado y el último registro desplazado.

ocupado por los registros borrados y esperar a una inserción posterior antes de volver a utilizar ese espacio. No basta con una simple marca en el registro borrado, ya que resulta difícil hallar el espacio disponible mientras se realiza una inserción. Por tanto, hay que introducir una estructura adicional.

Al comienzo del archivo se asigna cierto número de bytes como **cabecera del archivo**. La cabecera contiene gran variedad de información sobre el archivo. Por ahora, todo lo que hace falta guardar es la dirección del primer registro cuyo contenido se haya borrado. Se utiliza este primer registro para guardar la dirección del segundo registro disponible, y así sucesivamente. De manera intuitiva se pueden considerar estas direcciones guardadas como *punteros*, dado que indican la posición de un registro. Los registros borrados, por tanto, forman una lista enlazada a la que se suele denominar **lista libre**. La Figura 11.8 muestra el archivo de la Figura 11.5, con la lista libre, después de haberse borrado los registros 1, 4 y 6.

Al insertar un registro nuevo se utiliza el registro al que apunta la cabecera. Se modifica el puntero de la cabecera para que señale al siguiente registro disponible. Si no hay espacio disponible, se añade el nuevo registro al final del archivo.

La inserción y el borrado de archivos con registros de longitud fija son sencillas de implementar, dado que el espacio que deja libre cada registro borrado es exactamente el mismo que se necesita para insertar otro registro. Si se permiten en un archivo registros de longitud variable, esta coincidencia no se mantiene. Puede que el registro insertado no quepa en el espacio liberado por el registro borrado, o que sólo llene una parte.

11.6.2 Registros de longitud variable

Los registros de longitud variable surgen de varias maneras en los sistemas de bases de datos:

- Almacenamiento de varios tipos de registros en un mismo archivo.

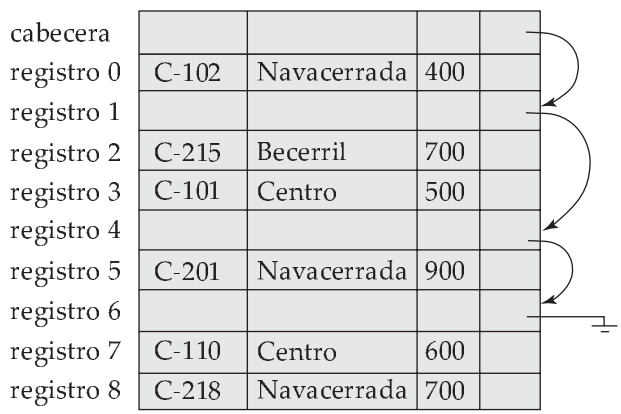


Figura 11.8 El archivo de la Figura 11.5 con la lista libre después del borrado de los registros 1, 4 y 6.

- Tipos de registro que permiten longitudes variables para uno o varios de los campos.
- Tipos de registro que permiten campos repetidos, como los arrays o los multiconjuntos.

Existen diferentes técnicas para implementar los registros de longitud variable.

La **estructura de páginas con ranuras** se utiliza habitualmente para organizar los registros en bloques, y puede verse en la Figura 11.9. Hay una cabecera al principio de cada bloque, que contiene la información siguiente:

1. El número de elementos del registro de la cabecera.
2. El final del espacio vacío del bloque.
3. Un *array* cuyas entradas contienen la ubicación y el tamaño de cada registro.

Los registros reales se ubican en el bloque *de manera contigua*, empezando por el final. El espacio libre dentro del bloque es contiguo, entre la última entrada del array de la cabecera y el primer registro. Si se inserta un registro, se le asigna espacio al final del espacio libre y se añade a la cabecera una entrada que contiene su tamaño y su ubicación.

Si se borra un registro, se libera el espacio que ocupa y se da el valor de borrada a su entrada (por ejemplo, se le da a su tamaño el valor de -1). Además, se desplazan los registros del bloque situados antes del registro borrado, de modo que se ocupe el espacio libre creado por el borrado y todo el espacio libre vuelve a hallarse entre la última entrada del array de la cabecera y el primer registro. También se actualiza de manera adecuada el puntero de final del espacio libre de la cabecera. Se puede aumentar o disminuir el tamaño de los registros mediante técnicas parecidas, siempre y cuando quede espacio en el bloque. El coste de trasladar los registros no es demasiado elevado, ya que el tamaño del bloque es limitado: un valor habitual es cuatro kilobytes.

La estructura de páginas con ranuras necesita que no haya punteros que señalen directamente a los registros. Por el contrario, los punteros deben apuntar a la entrada de la cabecera que contiene la ubicación verdadera del registro. Este nivel de tratamiento por la dirección permite que se desplacen los registros para evitar la fragmentación del espacio del bloque al tiempo que permite los punteros indirectos al registro.

Las bases de datos almacenan a menudo datos que pueden ser mucho más grandes que los bloques del disco. Por ejemplo, las imágenes o las grabaciones de sonido pueden tener un tamaño de varios megabytes, mientras que los objetos de vídeo pueden llegar a los gigabytes. Recuérdese que SQL soporta los tipos **blob** y **clob**, que almacenan objetos de gran tamaño de los tipos binario y carácter, respectivamente.

La mayor parte de las bases de datos relacionales limitan el tamaño de los registros para que no superen el tamaño de los bloques, con objeto de simplificar la gestión de la memoria intermedia y del espacio libre. Los objetos de gran tamaño y los campos largos suelen guardarse en archivos especiales (o conjuntos de archivos) en lugar de almacenarse con los otros atributos de pequeño tamaño de los registros en los que aparecen. Los objetos de gran tamaño se suelen representar en organizaciones de archivos de árboles B^+, que se estudian en el Apartado 12.3.4. Estas organizaciones permiten leer el objeto completo o rangos concretos de bytes del mismo, así como insertar y borrar partes del objeto.

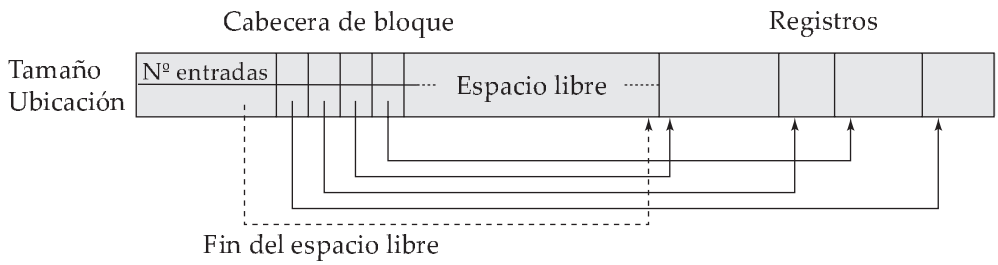


Figura 11.9 Estructura de páginas con ranuras.

11.7 Organización de los registros en archivos

Hasta ahora se ha estudiado la manera en que se representan los registros en la estructura de archivo. Las relaciones son conjuntos de registros. Dado un conjunto de registros, la pregunta siguiente es cómo organizarlos en archivos. A continuación se indican varias maneras de organizar los registros en archivos:

- **Organización de los archivos en montículos**. Se puede colocar cualquier registro en cualquier parte del archivo en que haya espacio suficiente. Los registros no se ordenan. Generalmente sólo hay un archivo para cada relación.
- **Organización secuencial de los archivos**. Los registros se guardan en orden secuencial, según el valor de la "clave de búsqueda" de cada uno. El Apartado 11.7.1 describe esta organización.
- **Organización asociativa (hash) de los archivos**. Se calcula una función de asociación (*hash*) para algún atributo de cada registro. El resultado de la función de asociación especifica el bloque del archivo en que se debe colocar cada registro. El Capítulo 12 describe esta organización; está estrechamente relacionada con las estructuras para la creación de índices que se describen en ese capítulo.

Generalmente se emplea un archivo separado para almacenar los registros de cada relación. No obstante, en cada **organización de archivos en agrupaciones de varias tablas** se pueden guardar en el mismo archivo registros de relaciones diferentes; además, los registros relacionados de las diferentes relaciones se guardan en el mismo bloque, por lo que cada operación de E/S afecta a registros relacionados de todas esas relaciones. Por ejemplo, los registros de dos relaciones se pueden considerar relacionados si casan en una reunión de las dos relaciones. El Apartado 11.7.2 describe esta organización.

11.7.1 Organización de archivos secuenciales

Los **archivos secuenciales** están diseñados para el procesamiento eficiente de los registros de acuerdo con un orden basado en alguna clave de búsqueda. La **clave de búsqueda** es cualquier atributo o conjunto de atributos; no tiene por qué ser la clave primaria, ni siquiera una superclave. Para permitir la recuperación rápida de los registros según el orden de la clave de búsqueda, éstos se vinculan mediante punteros. El puntero de cada registro señala al siguiente registro según el orden indicado por la clave de búsqueda. Además, para minimizar el número de accesos a los bloques en el procesamiento de los archivos secuenciales, los registros se guardan físicamente en el orden indicado por la clave de búsqueda, o lo más cercano posible.

La Figura 11.10 muestra un archivo secuencial de registros de *cuenta* tomado del ejemplo bancario propuesto. En ese ejemplo, los registros se guardan de acuerdo con el orden de la clave de búsqueda; en este caso, *nombre_sucursal*.

La organización secuencial de archivos permite que los registros se lean de forma ordenada, lo que puede resultar útil para la visualización, así como para ciertos algoritmos de procesamiento de consultas que se estudian en el Capítulo 13.

Sin embargo, resulta difícil mantener el orden físico secuencial a medida que se insertan y se borran registros, dado que resulta costoso desplazar muchos registros como consecuencia de una sola operación de inserción o de borrado. Se puede gestionar el borrado mediante cadenas de punteros, como ya se ha visto. Para la inserción se aplican las reglas siguientes:

1. Localizar el registro del archivo que precede al registro que se va a insertar según el orden de la clave de búsqueda.
2. Si existe algún registro vacío (es decir, un espacio que haya quedado libre después de una operación de borrado) dentro del mismo bloque que ese registro, el registro nuevo se insertará ahí. En caso contrario, el nuevo registro se insertará en un *bloque de desbordamiento*. En cualquier caso, hay que ajustar los punteros para vincular los registros según el orden de la clave de búsqueda.

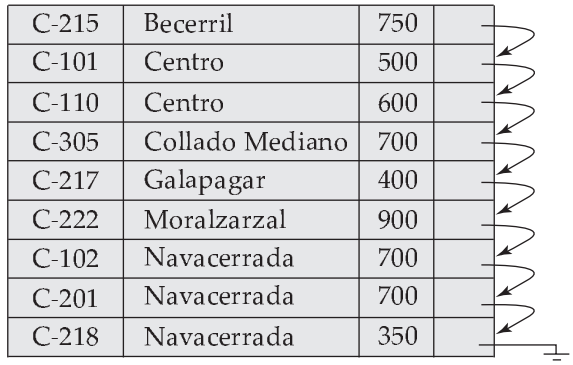


Figura 11.10 Archivo secuencial para los registros de *cuenta*.

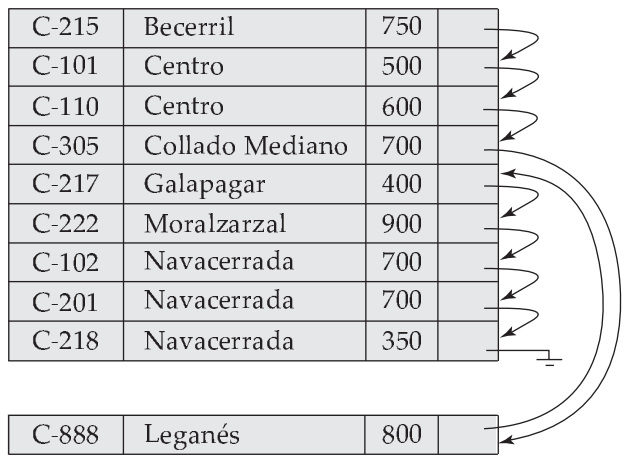


Figura 11.11 El archivo secuencial después de una inserción.

La Figura 11.11 muestra el archivo de la Figura 11.10 después de la inserción del registro (C-888, Leganés, 800). La estructura de la Figura 11.11 permite la inserción rápida de registros nuevos, pero obliga a las aplicaciones de procesamiento de archivos secuenciales a procesar los registros en un orden que no coincide con el físico.

Si hay que guardar un número relativamente pequeño de registros en los bloques de desbordamiento, este enfoque funciona bien. Finalmente, no obstante, la correspondencia entre el orden de la clave de búsqueda y el físico puede perderse totalmente, en cuyo caso el procesamiento secuencial acaba siendo mucho menos eficiente. Llegados a este punto, se debe **reorganizar** el archivo de modo que vuelva a estar físicamente en orden secuencial. Estas reorganizaciones resultan costosas y deben realizarse en momentos en los que la carga del sistema sea baja. La frecuencia con la que las reorganizaciones son necesarias depende de la frecuencia de inserción de registros nuevos. En el caso extremo en que rara vez se produzcan inserciones, es posible mantener siempre el archivo en el orden físico correcto. En ese caso, el campo puntero de la Figura 11.10 no es necesario.

11.7.2 Organización de archivos en agrupaciones de varias tablas

Muchos sistemas de bases de datos relacionales guardan cada relación en un archivo diferente, de modo que puedan aprovechar completamente el sistema de archivos proporcionado por el sistema operativo. Generalmente las tuplas de cada relación se pueden representar como registros de longitud fija. Por tanto, se puede hacer que las relaciones se correspondan con una estructura de archivos sencilla. Esta implementación sencilla de los sistemas de bases de datos relacionales resulta adecuada para las implementaciones de bajo coste de las bases de datos como, por ejemplo, los sistemas empotrados o los dispositivos portátiles. En estos sistemas el tamaño de la base de datos es pequeño, por lo que se obtiene

nombre_cliente	*número_cuenta*
López	C-102
López	C-220
López	C-503
Abril	C-305

Figura 11.12 La relación *impositor*.

poco provecho de una estructura de archivos avanzada. Además, en esos entornos, es fundamental que el tamaño total del código objeto del sistema de bases de datos sea pequeño. Una estructura de archivos sencilla reduce la cantidad de código necesaria para implementar el sistema.

Este enfoque sencillo de la implementación de las bases de datos relacionales resulta menos satisfactorio a medida que aumenta el tamaño de la base de datos. Ya se ha visto que se pueden obtener mejoras en el rendimiento mediante la asignación esmerada de los registros a los bloques y la cuidadosa organización de los propios bloques. Por tanto, resulta evidente que una estructura de archivos más compleja puede resultar beneficiosa, aunque se mantenga la estrategia de guardar cada relación en un archivo diferente.

Sin embargo, muchos sistemas de bases de datos de gran tamaño no utilizan directamente el sistema operativo subyacente para la gestión de los archivos. Por el contrario, se asigna al sistema de bases de datos un archivo de gran tamaño del sistema operativo. En este archivo, el sistema de bases de datos, que también lo administra, guarda todas las relaciones. Para comprender la ventaja de guardar muchas relaciones en un solo archivo considérese la siguiente consulta SQL de la base de datos bancaria:

select *número_cuenta, nombre_cliente, calle_cliente, ciudad_cliente*
from *impositor, cliente*
where *impositor.nombre_cliente = cliente.nombre_cliente*

Esta consulta calcula una reunión de las relaciones *impositor* y *cliente*. Por tanto, por cada tupla *impositor* el sistema debe encontrar las tuplas de *cliente* con el mismo valor de *nombre_cliente*. Lo ideal sería poder encontrar estos registros con la ayuda de *índices*, que se estudiarán en el Capítulo 12. Independientemente de la manera en que se encuentren esos registros, hay que transferirlos desde el disco a la memoria principal. En el peor de los casos, cada registro se hallará en un bloque diferente, lo que obligará a efectuar un proceso de lectura de bloque por cada registro necesario para la consulta.

A modo de ejemplo, considérense las relaciones *impositor* y *cliente* de las Figuras 11.12 y 11.13, respectivamente. En la Figura 11.14 se muestra una estructura de archivo diseñada para la ejecución eficiente de las consultas que implican $impositor \bowtie cliente$. Las tuplas *impositor* para cada *nombre_cliente* se guardan cerca de la tupla *cliente* para el *nombre_cliente* correspondiente. Esta estructura mezcla las tuplas de dos relaciones, pero permite el procesamiento eficaz de la reunión. Cuando se lee una tupla de la relación *cliente*, se copia del disco a la memoria principal todo el bloque que contiene esa tupla. Dado que las tuplas correspondientes de *impositor* se guardan en el disco cerca de la tupla *cliente*, el bloque que contiene la tupla *cliente* también contiene tuplas de la relación *impositor* necesarias para procesar la consulta. Si un cliente tiene tantas cuentas que los registros de *impositor* no caben en un solo bloque, los registros restantes aparecerán en bloques cercanos.

Una **organización de archivos en agrupaciones de varias tablas** es una organización de archivos, como la mostrada en la Figura 11.14, que almacena registros relacionados de dos o más relaciones en cada bloque. Este tipo de organización de archivos permite leer registros que satisfacen la condición de

nombre_cliente	*calle_cliente*	*ciudad_cliente*
López	Mayor	Arganzuela
Abril	Preciados	Valsaín

Figura 11.13 La relación *cliente*.

López	Mayor	Arganzuela
López	C-102	
López	C-220	
López	C-503	
Abril	Preciados	Valsaín
Abril	C-305	

Figura 11.14 Estructura de archivo en agrupaciones de varias tablas.

reunión en un solo proceso de lectura de bloques. Por tanto, esta consulta concreta se puede procesar de manera más eficiente.

El empleo de la agrupación de varias tablas en un único archivo ha mejorado el procesamiento de una reunión concreta (*impositor* ⋈ *cliente*) pero retarda el procesamiento de otros tipos de consulta. Por ejemplo:

select *
from *cliente*

necesita más accesos a los bloques que con el esquema en el que cada relación se guardaba en un archivo diferente. En lugar de aparecer varios registros de *cliente* en un mismo bloque, cada registro se halla en un bloque diferente. En realidad, lograr hallar todos los registros de *cliente* resulta imposible sin alguna estructura adicional. Para encontrar todas las tuplas de la relación *cliente* en la estructura de la Figura 11.14 se pueden enlazar todos los registros de esa relación mediante punteros, tal y como se muestra en la Figura 11.15.

El empleo de la agrupación de varias tablas depende de los tipos de consulta que el diseñador de la base de datos considere más frecuentes. El uso cuidadoso de la agrupación de varias tablas puede producir mejoras de rendimiento significativas en el procesamiento de las consultas.

11.8 Almacenamiento con diccionarios de datos

Hasta ahora sólo se ha considerado la representación de las propias relaciones. Un sistema de bases de datos relacionales necesita tener datos *sobre* las relaciones, como puede ser su esquema. Esta información se denomina **diccionario de datos** o **catálogo del sistema**. Entre los tipos de información que debe guardar el sistema figuran los siguientes:

- El nombre de las relaciones.
- El nombre de los atributos de cada relación.
- El dominio y la longitud de los atributos.
- El nombre de las vistas definidas en la base de datos, y la definición de esas vistas.
- Las restricciones de integridad (por ejemplo, las restricciones de las claves).

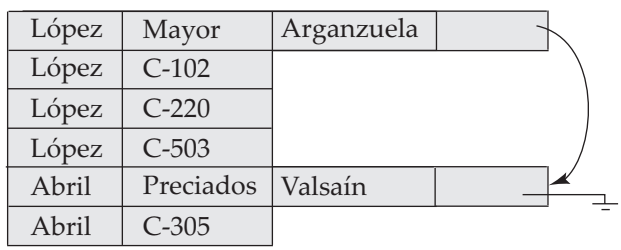


Figura 11.15 Estructura de archivo con agrupaciones de varias tablas y cadenas de punteros.

Metadatos_relación (*nombre_relación*, *número_de_atributos*, *organización_almacenamiento*, *ubicación*)
Metadatos_atributos (*nombre_atributo*, *nombre_relación*, *tipo_dominio*, *posición*, *longitud*)
Metadatos_usuarios (*nombre_usuario*, *contraseña_cifrada*, *grupo*)
Metadatos_índices (*nombre_índice*, *nombre_relación*, *tipo_índice*, *atributos_índice*)
Metadatos_vistas (*nombre_vista*, *definición*)

Figura 11.16 Base de datos relacional que representa datos del sistema.

Además, muchos sistemas guardan los siguientes datos de los usuarios del sistema:

- El nombre de los usuarios autorizados.
- La autorización y la información sobre las cuentas de los usuarios.
- Las contraseñas u otra información utilizada para autentificar a los usuarios.

Además, la base de datos puede guardar información estadística y descriptiva sobre las relaciones, como:

- El número de tuplas de cada relación.
- El método de almacenamiento utilizado para cada relación (por ejemplo, con agrupaciones o sin agrupaciones).

El diccionario de datos puede también tener en cuenta la organización del almacenamiento (secuencial, asociativa o en montículos) de las relaciones y la ubicación donde se guarda cada relación:

- Si las relaciones se almacenan en archivos del sistema operativo, el diccionario tendrá en cuenta el nombre del archivo (o archivos) que guarda cada relación.
- Si la base de datos almacena todas las relaciones en un solo archivo, puede que el diccionario tenga en cuenta los bloques que almacenan los registros de cada relación en una estructura de datos como, por ejemplo, una lista enlazada.

En el Capítulo 12, en el que se estudian los índices, se verá que hace falta guardar información sobre cada índice de cada una de las relaciones:

- El nombre del índice.
- El nombre de la relación para la que se crea.
- Los atributos sobre los que se define.
- El tipo de índice formado.

Toda esta información constituye, en efecto, una base de datos en miniatura. Algunos sistemas de bases de datos guardan esta información utilizando código y estructuras de datos especiales. Suele resultar preferible guardar los datos sobre la base de datos en la misma base de datos. Al utilizar la base de datos para guardar los datos del sistema se simplifica la estructura global del sistema y se dedica toda la potencia de la base de datos a obtener un acceso rápido a los datos del sistema.

La elección exacta de la manera de representar los datos del sistema mediante las relaciones debe tomarla el diseñador del sistema. La Figura 11.16 ofrece una representación posible, con las claves primarias subrayadas. En esta representación se da por supuesto que el atributo *atributos_índice* de la relación *Metadatos_índices* contiene una lista de uno o varios atributos, que se pueden representar mediante una cadena de caracteres como "*nombre_sucursal*, *ciudad_sucursal*". Por tanto, la relacion *Metadatos_índices* no está en la primera forma normal; se puede normalizar, pero es probable que la representación anterior sea más eficiente en el acceso. El diccionario de datos se suele almacenar de forma no normalizada para conseguir un acceso rápido.

La organización del almacenamiento y de la ubicación de la propia *Metadatos_relación* se debe registrar en otro lugar (por ejemplo, en el propio código de la base de datos) ya que esa información es necesaria para encontrar el contenido de *Metadatos_relación*.

11.9 Resumen

- En la mayor parte de los sistemas informáticos hay varios tipos de almacenamiento de datos. Se clasifican según la velocidad con la que se puede acceder a los datos, el coste de adquisición de la memoria por unidad de datos y su fiabilidad. Entre los medios disponibles figuran la memoria caché, la principal, la flash, los discos magnéticos, los discos ópticos y las cintas magnéticas.
- La fiabilidad de los medios de almacenamiento la determinan dos factores: si un corte en el suministro eléctrico o una caída del sistema hace que los datos se pierdan y la probabilidad de fallo físico del dispositivo de almacenamiento.
- Se puede reducir la probabilidad del fallo físico conservando varias copias de los datos. Para los discos se pueden crear imágenes. También se pueden usar métodos más sofisticados como las disposiciones redundantes de discos independientes (RAID). Mediante la distribución de los datos entre los discos estos métodos ofrecen elevados niveles de productividad en los accesos de gran tamaño; la introducción de la redundancia entre los discos se mejora mucho la fiabilidad. Se han propuesto varias organizaciones RAID diferentes, con características de coste, rendimiento y fiabilidad diferentes. Las organizaciones RAID de nivel 1 (creación de imágenes) y de nivel 5 son las más utilizadas.
- Una manera de reducir el número de accesos al disco es conservar todos los bloques posibles en la memoria principal. Dado que allí no se pueden guardar todos, hay que gestionar la asignación del espacio disponible en la memoria principal para el almacenamiento de los bloques. La *memoria intermedia* (*buffer*) es la parte de la memoria principal disponible para el almacenamiento de las copias de los bloques del disco. El subsistema responsable de la asignación del espacio de la memoria intermedia se denomina *gestor de la memoria intermedia*.
- Los archivos se pueden organizar lógicamente como secuencias de registros asignados a bloques de disco. Un enfoque de la asociación de la base de datos con los archivos es utilizar varios archivos y guardar registros de una única longitud fija en cualquier archivo dado. Una alternativa es estructurar los archivos de modo que puedan aceptar registros de longitud variable. El método de la página con ranuras se usa mucho para manejar los registros de longitud variable en los bloques de disco.
- Dado que los datos se transfieren entre el almacenamiento en disco y la memoria principal en unidades de bloques merece la pena asignar los registros de los archivos a los bloques de modo que cada bloque contenga registros relacionados entre sí. Si se puede tener acceso a varios de los registros deseados utilizando sólo un acceso a bloques, se evitan accesos al disco. Dado que los accesos al disco suelen ser el cuello de botella del rendimiento de los sistemas de bases de datos, la esmerada asignación de registros a los bloques puede ofrecer mejoras significativas del rendimiento.
- El diccionario de datos, también denominado catálogo del sistema, guarda información de los metadatos, que son datos relativos a los datos, como el nombres de las relaciones, el nombre de los atributos y su tipo, la información de almacenamiento, las restricciones de integridad y la información de usuario.

Términos de repaso

- Medios de almacenamiento físico:
  - ☐ Caché.
  - ☐ Memoria principal.
  - ☐ Memoria flash.
  - ☐ Disco magnético.
  - ☐ Almacenamiento óptico.
- Disco magnético

- ☐ Plato.
- ☐ Discos duros.
- ☐ Disquetes.
- ☐ Pistas.
- ☐ Sectores.
- ☐ Cabeza de lectura y escritura.
- ☐ Brazo del disco.
- ☐ Cilindro.
- ☐ Controlador de discos.
- ☐ Suma de comprobación.
- ☐ Reasignación de sectores defectuosos.

- Medidas de rendimiento de los discos:
  - ☐ Tiempo de acceso.
  - ☐ Tiempo de búsqueda.
  - ☐ Latencia rotacional.
  - ☐ Velocidad de transferencia de datos.
  - ☐ Tiempo medio entre fallos.
- Bloque de disco.
- Optimización del acceso a bloques de disco.
  - ☐ Planificación del brazo.
  - ☐ Algoritmo del ascensor.
  - ☐ Organización de archivos.
  - ☐ Desfragmentación.
  - ☐ Memorias intermedias de escritura no volátiles.
  - ☐ Memoria no volátil de acceso aleatorio (NV-RAM).
  - ☐ Disco del registro histórico.
- Disposición redundante de discos independientes (RAID).
  - ☐ Creación de imágenes.
  - ☐ Distribución de datos.
  - ☐ Distribución en el nivel de bits.
  - ☐ Distribución en el nivel de bloques.
- Niveles de RAID:
  - ☐ Nivel 0 (distribución de bloques, sin redundancia).
  - ☐ Nivel 1 (distribución de bloques, con creación de imágenes).
  - ☐ Nivel 3 (distribución de bits, con paridad).
  - ☐ Nivel 5 (distribución de bloques, con paridad distribuida).
  - ☐ Nivel 6 (distribución de bloques, con redundancia P+Q).
- Rendimiento de la reconstrucción.
- RAID software.
- RAID hardware.
- Intercambio en caliente.
- Almacenamiento terciario:
  - ☐ Discos ópticos.
  - ☐ Cintas magnéticas.
  - ☐ Cambiadores automáticos.
- Memoria intermedia (buffer).
  - ☐ Gestor de la memoria intermedia.
  - ☐ Bloques clavados.
  - ☐ Salida forzada de bloques.
- Políticas de sustitución de la memoria intermedia:
  - ☐ Menos recientemente utilizado (Least Recently Used, LRU).
  - ☐ Extracción inmediata.
  - ☐ Más recientemente utilizado (Most Recently Used, MRU).
- Archivo.
- Organización de archivos.
  - ☐ Cabecera del archivo.
  - ☐ Lista libre.
- Registros de longitud variable.
  - ☐ Estructura de páginas con ranuras.
- Objetos de gran tamaño.
- Organización de archivos en montículos.
- Organización secuencial de archivos.
- Organización asociativa (hash) de archivos.
- Organización de archivos en agrupaciones de varias tablas.
- Clave de búsqueda.
- Diccionario de datos.
- Catálogo del sistema.

Ejercicios prácticos

11.1 Considérese la siguiente disposición de los bloques de datos y de paridad de cuatro discos:

Disco 1	Disco 2	Disco 3	Disco 4
B_1	B_2	B_3	B_4
P_1	B_5	B_6	B_7
B_8	P_2	B_9	B_{10}
⋮	⋮	⋮	⋮

B_i representa los bloques de datos; P_i, los bloques de paridad. El bloque de paridad P_i es el bloque de paridad para los bloques de datos B_{4i-3} a B_{4i}. Indíquense los problemas que puede presentar esta disposición.

11.2 Un fallo en el suministro eléctrico que se produzca mientras se escribe un bloque del disco puede dar lugar a que el bloque sólo se escriba parcialmente. Supóngase que se pueden detectar los bloques escritos parcialmente. Un proceso atómico de escritura de bloque es aquél en el que se escribe el bloque entero o no se escribe en absoluto (es decir, no hay procesos de escritura parciales). Propónganse esquemas para conseguir el efecto de los procesos atómicos de escritura de bloques con los siguientes esquemas RAID. Los esquemas deben implicar procesos de recuperación de fallos.

- **a.** RAID de nivel 1 (creación de imágenes).
- **b.** RAID de nivel 5 (entrelazado de bloques, paridad distribuida).

11.3 Dese un ejemplo de expresión de álgebra relacional y de estrategia de procesamiento de consultas en cada una de las situaciones siguientes:

- **a.** MRU es preferible a LRU.
- **b.** LRU es preferible a MRU.

11.4 Considérese el borrado del registro 5 del archivo de la Figura 11.7. Compárense las ventajas relativas de las siguientes técnicas para implementar el borrado:

- **a.** Trasladar el registro 6 al espacio que ocupaba el registro 5 y desplazar el registro 7 al espacio ocupado por el registro 6.
- **b.** Trasladar el registro 7 al espacio que ocupaba el registro 5.
- **c.** Marcar el registro 5 como borrado y no desplazar ningún registro.

11.5 Muéstrese la estructura del archivo de la Figura 11.8 después de cada uno de los pasos siguientes:

- **a.** Insertar (Galapagar, C-323, 1600).
- **b.** Borrar el registro 2.
- **c.** Insertar (Galapagar, C-626, 2000).

11.6 Considérese una base de datos relacional con dos relaciones:

asignatura (*nombre_asignatura*, *aula*, *profesor*)
matrícula (*nombre_asignatura*, *nombre_estudiante*, *nota*)

Defínanse ejemplos de estas relaciones para tres asignaturas, en cada una de los cuales se matriculan cinco estudiantes. Dese una estructura de archivos de estas relaciones que utilice la agrupación de varias tablas.

11.7 Considérese la siguiente técnica de mapa de bits para realizar el seguimiento del espacio libre de un archivo. Por cada bloque del archivo se mantienen dos bits en el mapa. Si el bloque está lleno entre el 0 y el 30 por ciento, los bits son 00; entre el 30 y el 60 por ciento, 01; entre el 60 y el 90 por ciento, 10; y, por encima del 90 por ciento, 11. Estos mapas de bits se pueden mantener en memoria incluso para archivos de gran tamaño.

- **a.** Descríbase la manera de mantener actualizado el mapa de bits mientras se insertan y se eliminan registros.
- **b.** Descríbase la ventaja de la técnica de los mapas de bits frente a las listas libres en la búsqueda de espacio libre y en la actualización de la información.

Ejercicios

11.8 El lector debe indicar los medios de almacenamiento físico disponibles en las computadoras que utiliza habitualmente. Indique la velocidad con la que se puede acceder a los datos en cada medio.

11.9 ¿Cómo afecta la reasignación por los controladores de disco de los sectores dañados a la velocidad de recuperación de los datos?

11.10 Los sistemas RAID suelen permitir la sustitución de los discos averiados sin interrupción del acceso al sistema. Por tanto, los datos del disco averiado deben reconstruirse y escribirse en el disco de repuesto mientras el sistema se halla en funcionamiento. ¿Con cuál de los niveles RAID es menor la interferencia entre la reconstrucción y el acceso a los discos? Explíquese la respuesta.

11.11 Explíquese por qué la asignación de registros a los bloques afecta de manera significativa al rendimiento de los sistemas de bases de datos.

11.12 Si es posible, determine la estrategia de gestión de la memoria intermedia del sistema operativo que se ejecuta en su computadora y los mecanismos de que dispone para controlar la sustitución de páginas. Explíquese el modo en que el control sobre la sustitución que proporciona pudiera ser útil para la implementación de sistemas de bases de datos.

11.13 En la organización secuencial de los archivos, ¿por qué se utiliza un *bloque* de desbordamiento aunque sólo haya, en ese momento, un registro de desbordamiento?

11.14 Indíquense dos ventajas y dos inconvenientes de cada una de las estrategias siguientes para el almacenamiento de bases de datos relacionales:

a. Guardar cada relación en un archivo.
b. Guardar varias relaciones (quizá toda la base de datos) en un archivo.

11.15 Dese una versión normalizada de la relación *Metadatos_índices* y explíquese por qué el empleo de la versión normalizada supondría una pérdida de rendimiento.

11.16 Si se tienen datos que no se deben perder en un fallo de disco y la escritura de datos es intensiva, ¿cómo se almacenarán esos datos?

11.17 En discos de generaciones anteriores el número de sectores por pista era el mismo en todas las pistas. Los discos de las generaciones actuales tienen más sectores por pista en las externas y menos en las internas (ya que son más cortas). ¿Cuál es el efecto de este cambio en cada uno de los tres indicadores principales de la velocidad de los discos?

11.18 Los gestores habituales de la memoria intermedia dan por supuesto que cada página es del mismo tamaño y cuesta lo mismo leerla. Considérese un gestor que, en lugar de LRU, utilice la tasa de referencia a objetos, es decir, la frecuencia con que se ha accedido a ese objeto en los últimos n segundos. Supóngase que se desea almacenar en la memoria intermedia objetos de distinto tamaño y con costes de lectura diferentes (como las páginas Web, cuyo coste de lectura depende del sitio desde el que se busquen). Sugiérase cómo puede elegir el gestor la página que debe descartar de la memoria intermedia.

Notas bibliográficas

Hennessy et al. [2002] es un popular libro de texto de arquitectura de computadoras, que incluye los aspectos de hardware de la memoria intermedia con traducción anticipada, de las cachés y de las unidades de gestión de la memoria. Rosch [2003] presenta una excelente visión general del hardware de las computadoras, incluido un extenso tratamiento de todos los tipos de tecnologías de almacenamiento, como los disquetes, los discos magnéticos, los discos ópticos, las cintas y las interfaces de almacenamiento. Patterson [2004] ofrece un buen estudio sobre el modo en que la mejora de la latencia ha ido por detrás de la mejora del ancho de banda (velocidad de transferencia).

Con el rápido crecimiento de las velocidades de la CPU, las memorias caché ubicadas en la CPU han llegado a ser mucho más rápidas que la memoria principal. Aunque los sistemas de bases de datos no controlan lo que se almacena en la caché, cada vez hay más motivos para organizar los datos en memoria y escribir los programas de forma que se maximice el empleo de la caché. Entre los trabajos en este área figuran Rao y Ross [2000], Ailamaki et al. [2001] y Zhou y Ross [2004].

Las especificaciones de las unidades de disco actuales se pueden obtener de los sitios Web de sus fabricantes, como Hitachi, IBM (que ha vendido recientemente su línea de fabricación de discos a Hitachi), Seagate y Maxtor.

La distribución en los discos se describe en Salem y Garcia-Molina [1986]. Las disposiciones redundantes de discos independientes (RAID) se explican en Patterson et al. [1988] y en Chen y Patterson [1990]. Chen et al. [1994] presenta una excelente revisión de los principios y de la aplicación de RAID. Los códigos de Reed–Solomon se tratan en Pless [1998]. Entre las organizaciones de disco alternativas que proporcionan un alto grado de tolerancia frente a fallos están las descritas en Gray et al. [1990] y en Bitton y Gray [1988]. El sistema de archivos basado en el registro histórico, que transforma en secuencial la escritura en el disco, se describe en Rosenblum y Ousterhout [1991]. Rosenblum y Ousterhout [1991].

En los sistemas que permiten la informática portátil se pueden transmitir los datos de manera reiterada. El medio de transmisión puede considerarse un nivel de la jerarquía de almacenamiento (como los discos transmisores con latencia elevada). Estos problemas se estudian en Acharya et al. [1995]. La gestión de la caché y de las memorias intermedias en la informática portátil se trata en Barbará y Imielinski [1994].

La estructura de almacenamiento de sistemas concretos de bases de datos, como DB2 de IBM, Oracle o SQL Server de Microsoft, se documentan en sus respectivos manuales de sistema.

La gestión de las memorias intermedias se explica en la mayor parte de los textos sobre sistemas operativos, incluido Silberschatz et al. [2001]. Chou y Dewitt [1985] presenta algoritmos para la gestión de la memoria intermedia en los sistemas de bases de datos y describe un método de medida del rendimiento.

CAPÍTULO 12

Indexación y asociación

Muchas consultas hacen referencia sólo a una pequeña parte de los registros de un archivo. Por ejemplo, la consulta "Buscar todas las cuentas de la sucursal Pamplona" o "Buscar el saldo del número de cuenta C-101" hace referencia solamente a una fracción de los registros de la relación cuenta. No es eficiente que el sistema tenga que leer cada registro y comprobar que el campo *nombre_sucursal* contiene el nombre "Pamplona" o el valor C-101 del campo *número_cuenta*. Lo más adecuado sería que el sistema fuese capaz de localizar directamente esos registros. Para facilitar estas formas de acceso se diseñan estructuras adicionales que se asocian con los archivos.

12.1 Conceptos básicos

Un índice para un archivo del sistema funciona como el índice de este libro. Si se va a buscar un tema (especificado por una palabra o una frase) se puede buscar en el índice al final del libro, encontrar las páginas en las que aparece y después leerlas para encontrar la información buscada. Las palabras del índice están ordenadas alfabéticamente, lo cual facilita la búsqueda. Además, el índice es mucho más pequeño que el libro, con lo que se reduce aún más el esfuerzo necesario para encontrar las palabras en cuestión.

Los índices de los sistemas de bases de datos juegan el mismo papel que los índices de los libros en las bibliotecas. Por ejemplo, para recuperar un registro *cuenta* dado su número de cuenta, el sistema de bases de datos buscaría en un índice para encontrar el bloque de disco en que se localice el registro correspondiente, y entonces extraería ese bloque de disco para obtener el registro *cuenta*.

Almacenar una lista ordenada de números de cuenta no funcionaría bien en bases de datos muy grandes con millones de cuentas, ya que el propio índice sería muy grande; más aún, incluso el mantener ordenado el índice reduce el tiempo de búsqueda, por lo que encontrar una cuenta podría consumir mucho tiempo. En su lugar se usan técnicas más sofisticadas de indexación. Algunas de estas técnicas se estudiarán más adelante.

Hay dos tipos básicos de índices:

- **Índices ordenados**. Estos índices están basados en una disposición ordenada de los valores.
- **Índices asociativos**. Estos índices están basados en una distribución uniforme de los valores a través de una serie de cajones (*buckets*). El valor asignado a cada cajón está determinado por una función, llamada *función de asociación* (*hash function)*.

Se considerarán varias técnicas de indexación y asociación. Ninguna de ellas es la mejor. Sin embargo, para cada aplicación específica de bases de datos existe una técnica más apropiada. Cada una de ellas debe ser valorada según los siguientes criterios:

- **Tipos de acceso**. Los tipos de acceso que se soportan eficazmente. Estos tipos podrían incluir la búsqueda de registros con un valor concreto en un atributo, o la búsqueda de los registros cuyos atributos contengan valores en un rango especificado.
- **Tiempo de acceso**. El tiempo que se tarda en hallar un determinado elemento de datos, o conjunto de elementos, usando la técnica en cuestión.
- **Tiempo de inserción**. El tiempo empleado en insertar un nuevo elemento de datos. Este valor incluye el tiempo utilizado en hallar el lugar apropiado donde insertar el nuevo elemento de datos, así como el tiempo empleado en actualizar la estructura del índice.
- **Tiempo de borrado**. El tiempo empleado en borrar un elemento de datos. Este valor incluye el tiempo utilizado en hallar el elemento a borrar, así como el tiempo empleado en actualizar la estructura del índice.
- **Espacio adicional requerido**. El espacio adicional ocupado por la estructura del índice. Como normalmente la cantidad necesaria de espacio adicional suele ser moderada, es razonable sacrificar el espacio para alcanzar un rendimiento mejor.

A menudo se desea tener más de un índice por archivo. Por ejemplo se puede buscar un libro según el autor, la materia o el título.

Los atributos o conjunto de atributos usados para buscar en un archivo se llaman **claves de búsqueda**. Hay que observar que esta definición de *clave* difiere de la usada en *clave primaria*, *clave candidata* y *superclave*. Este doble significado de *clave* está (desafortunadamente) muy extendido en la práctica. Usando el concepto de clave de búsqueda se determina que, si existen varios índices para un archivo, existirán varias claves de búsqueda.

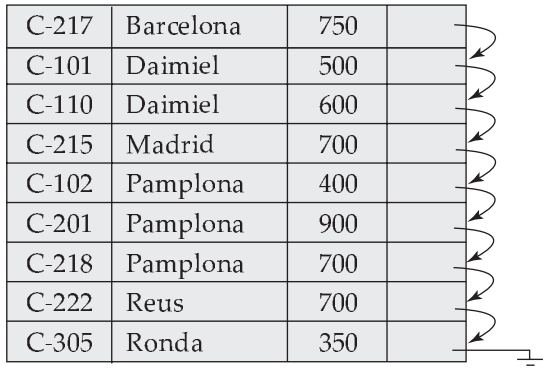
12.2 Índices ordenados

Para permitir un acceso directo rápido a los registros de un archivo se puede usar una estructura de índice. Cada estructura de índice está asociada con una clave de búsqueda concreta. Al igual que en el catálogo de una biblioteca, un índice almacena de manera ordenada los valores de las claves de búsqueda, y asocia a cada clave los registros que contienen esa clave de búsqueda.

Los registros en el archivo indexado pueden estar a su vez almacenados siguiendo un orden, semejante a como los libros están ordenados en una biblioteca por algún atributo como el número decimal Dewey. Un archivo puede tener varios índices según diferentes claves de búsqueda. Si el archivo que contiene los registros está ordenado secuencialmente, el índice cuya clave de búsqueda especifica el orden secuencial del archivo es el **índice con agrupación** (clustering index). Los índices con agrupación también se llaman **índices primarios**; el término índice primario se usa algunas veces para hacer alusión a un índice según una clave primaria pero en realidad se puede usar sobre cualquier clave de búsqueda. La clave de búsqueda de un índice con agrupación es normalmente la clave primaria, aunque no es así necesariamente. Los índices cuyas claves de búsqueda especifican un orden diferente del orden secuencial del archivo se llaman **índices sin agrupación** (non clustering indices) o **secundarios**. Se suelen usar los términos "agrupado" y "no agrupado" en lugar de "con agrupación" y "sin agrupación".

Desde el Apartado 12.2.1 hasta el 12.2.3 se asume que todos los archivos están ordenados secuencialmente según alguna clave de búsqueda. Estos archivos con un índice con agrupación según la clave de búsqueda se llaman **archivos secuenciales indexados**. Representan uno de los esquemas de índices más antiguos usados por los sistemas de bases de datos. Se usan en aquellas aplicaciones que demandan un procesamiento secuencial del archivo completo, así como un acceso directo a sus registros. En el Apartado 12.2.4 se tratan los índices secundarios.

En la Figura 12.1 se muestra un archivo secuencial de los registros *cuenta* tomados del ejemplo bancario. En esta figura, los registros están almacenados según el orden de la clave de búsqueda: *nombre_sucursal*.



C-217	Barcelona	750	
C-101	Daimiel	500	
C-110	Daimiel	600	
C-215	Madrid	700	
C-102	Pamplona	400	
C-201	Pamplona	900	
C-218	Pamplona	700	
C-222	Reus	700	
C-305	Ronda	350	

Figura 12.1 Archivo secuencial para los archivos de *cuenta*.

12.2.1 Índices densos y dispersos

Un **registro índice** o **entrada del índice** consiste en un valor de la clave de búsqueda y punteros a uno o más registros con ese valor de la clave de búsqueda. El puntero a un registro consiste en el identificador de un bloque de disco y un desplazamiento en el bloque de disco para identificar el registro dentro del bloque.

Existen dos clases de índices ordenados que se pueden usar:

- **Índice denso**. Aparece un registro índice por cada valor de la clave de búsqueda en el archivo. En un índice denso con agrupación el registro índice contiene el valor de la clave y un puntero al primer registro con ese valor de la clave de búsqueda. El resto de registros con el mismo valor de la clave de búsqueda se almacenan consecutivamente después del primer registro, dado que, ya que el índice es con agrupación, los registros se ordenan sobre la misma clave de búsqueda.

  Las implementaciones de índices densos pueden almacenar una lista de punteros a todos los registros con el mismo valor de la clave de búsqueda; esto no es esencial para los índices con agrupación.

- **Índice disperso**. Sólo se crea un registro índice para algunos de los valores. Al igual que en los índices densos, cada registro índice contiene un valor de la clave de búsqueda y un puntero al primer registro con ese valor de la clave. Para localizar un registro se busca la entrada del índice con el valor más grande que sea menor o igual que el valor que se está buscando. Se empieza por el registro apuntado por esa entrada del índice y se continúa con los punteros del archivo hasta encontrar el registro deseado.

Las Figuras 12.2 y 12.3 son ejemplos de índices densos y dispersos, respectivamente, para el archivo *cuenta*. Supóngase que se desea buscar los registros de la sucursal Pamplona. Mediante el índice denso de la Figura 12.2, se sigue el puntero que va directo al primer registro de Pamplona. Se procesa el registro y se sigue el puntero en ese registro hasta localizar el siguiente registro según el orden de la clave de búsqueda (*nombre_sucursal*). Se continuaría procesando registros hasta encontrar uno cuyo nombre de sucursal fuese distinto de Pamplona. Si se usa un índice disperso (Figura 12.3), no se encontraría entrada del índice para "Pamplona". Como la última entrada (en orden alfabético) antes de "Pamplona" es "Madrid", se sigue ese puntero. Entonces se lee el archivo *cuenta* en orden secuencial hasta encontrar el primer registro Pamplona, y se continúa procesando desde este punto.

Como se ha visto, generalmente es más rápido localizar un registro si se usa un índice denso en vez de un índice disperso. Sin embargo, los índices dispersos tienen algunas ventajas sobre los índices densos, como el utilizar un espacio más reducido y un mantenimiento adicional menor para las inserciones y borrados.

Existe un compromiso que el diseñador del sistema debe mantener entre el tiempo de acceso y el espacio adicional requerido. Aunque la decisión sobre este compromiso depende la aplicación en particular, un buen compromiso es tener un índice disperso con una entrada del índice por cada bloque. La razón por la cual este diseño alcanza un buen compromiso reside en que el mayor coste de procesar

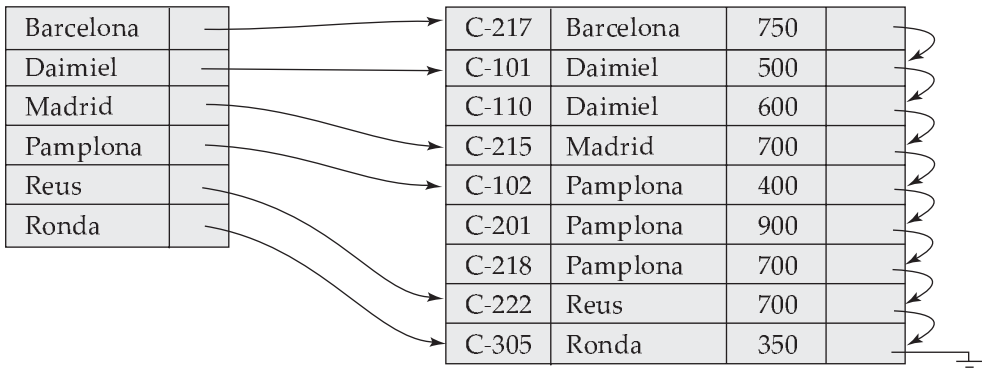


Figura 12.2 Índice denso.

una petición en la base de datos es el empleado en traer un bloque de disco a la memoria. Una vez el bloque en memoria, el tiempo en examinarlo es prácticamente inapreciable. Usando este índice disperso se localiza el bloque que contiene el registro solicitado. De este manera, a menos que el registro esté en un bloque de desbordamiento (véase el Apartado 12.7.1) se minimizan los accesos a bloques mientras el tamaño del índice se mantiene (y así, el espacio adicional requerido) tan pequeño como sea posible.

Para generalizar la técnica anterior hay que tener en cuenta si los registros de una clave de búsqueda ocupan varios bloques. Es fácil modificar el esquema para acomodarse a esta situación.

12.2.2 Índices multinivel

Aunque se use un índice disperso, puede que el propio índice sea demasiado grande para un procesamiento eficiente. En la práctica no es excesivo tener un archivo con 100.000 registros, con 10 registros almacenados en cada bloque. Si se dispone de un registro índice por cada bloque, el índice tendrá 10.000 registros. Como los registros índices son más pequeños que los registros de datos, se puede suponer que caben 100 registros índices en un bloque. Por tanto el índice ocuparía 100 bloques. Estos índices de mayor tamaño se almacenan como archivos secuenciales en disco.

Si un índice es lo bastante pequeño como para guardarlo en la memoria principal, el tiempo de búsqueda para encontrar una entrada será breve. Sin embargo, si el índice es tan grande que se debe guardar en disco, buscar una entrada implicará leer varios bloques de disco. Para localizar una entrada en el archivo índice se puede realizar una búsqueda binaria, pero aun así ésta conlleva un gran coste. Si el índice ocupa b bloques, la búsqueda binaria tendrá que leer a lo sumo $\lceil \log_2(b) \rceil$ bloques. ($\lceil x \rceil$ denota el menor entero que es mayor o igual que x; es decir, se redondea hacia abajo). Para el índice de 100 bloques, la búsqueda binaria necesitará leer siete bloques. En un disco en el que la lectura de un bloque tarda 30 milisegundos, la búsqueda empleará 210 milisegundos, que es bastante. Obsérvese que si se están usando bloques de desbordamiento, la búsqueda binaria no será posible. En ese caso, lo normal es una

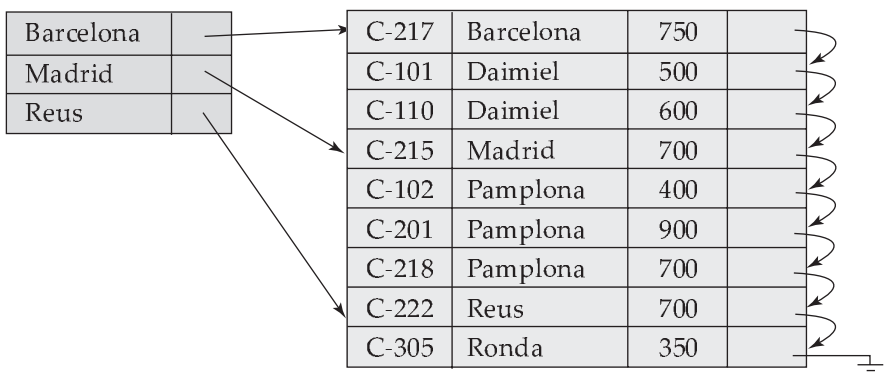


Figura 12.3 Índice disperso.

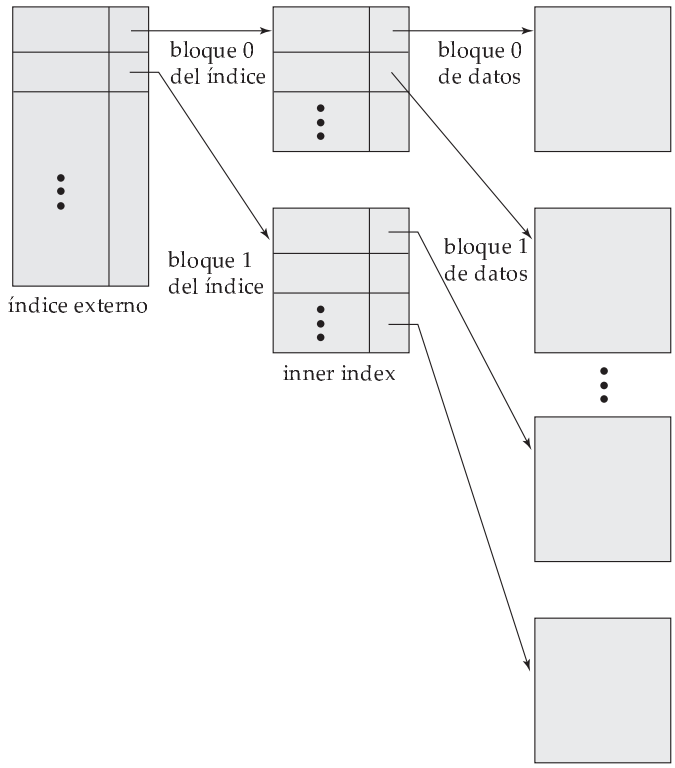


Figura 12.4 Índice disperso de dos niveles.

búsqueda secuencial, y eso requiere leer *b* bloques, lo que podría consumir incluso más tiempo. Así, el proceso de buscar en un índice grande puede ser muy costoso.

Para resolver este problema el índice se trata como si fuese un archivo secuencial y se construye un índice disperso sobre el índice con agrupación, como se muestra en la Figura 12.4. Para localizar un registro se usa en primer lugar una búsqueda binaria sobre el índice más externo para buscar el registro con el mayor valor de la clave de búsqueda que sea menor o igual al valor deseado. El puntero apunta a un bloque en el índice más interno. Hay que examinar este bloque hasta encontrar el registro con el mayor valor de la clave que sea menor o igual que el valor deseado. El puntero de este registro apunta al bloque del archivo que contiene el registro buscado.

Usando los dos niveles de indexación y con el índice más externo en memoria principal hay que leer un único bloque índice, en vez de los siete que se leían con la búsqueda binaria. Si al archivo es extremadamente grande, incluso el índice exterior podría crecer demasiado para caber en la memoria principal. En este caso se podría crear todavía otro nivel más de indexación. De hecho, se podría repetir este proceso tantas veces como fuese necesario. Los índices con dos o más niveles se llaman índices **multinivel**. La búsqueda de registros usando un índice multinivel necesita claramente menos operaciones de E/S que las que se emplean en la búsqueda de registros con la búsqueda binaria. Cada nivel de índice se podría corresponder con una unidad del almacenamiento físico. Así, es posible disponer de índices para pistas, cilindros y discos.

Un diccionario normal es un ejemplo de un índice multinivel en el mundo ajeno a las bases de datos. La cabecera de cada página lista la primera palabra en el orden alfabético en esa página. Este índice es multinivel: las palabras en la parte superior de la página del índice del libro forman un índice disperso sobre los contenidos de las páginas del diccionario.

Los índices multinivel están estrechamente relacionados con la estructura de árbol, tales como los árboles binarios usados para la indexación en memoria. Esta relación se examinará posteriormente en el Apartado 12.3.

12.2.3 Actualización del índice

Sin importar el tipo de índice que se esté usando, los índices se deben actualizar siempre que se inserte o borre un registro del archivo. A continuación se describirán los algoritmos para actualizar índices de un solo nivel.

- **Inserción**. Primero se realiza una búsqueda usando el valor de la clave de búsqueda del registro a insertar. Las acciones que emprende el sistema a continuación dependen de si el índice es denso o disperso.
  - Índices densos:
    1. Si el valor de la clave de búsqueda no aparece en el índice, el sistema inserta en éste un registro índice con el valor de la clave de búsqueda en la posición adecuada.
    2. En caso contrario se emprenden las siguientes acciones:
       a. Si el registro índice almacena punteros a todos los registros con el mismo valor de la clave de búsqueda, el sistema añade un puntero al nuevo registro en el registro índice.
       b. En caso contrario, el registro índice almacena un puntero sólo hacia el primer registro con el valor de la clave de búsqueda. El sistema sitúa el registro insertado después de los otros con los mismos valores de la clave de búsqueda.
  - Índices dispersos: se asume que el índice almacena una entrada por cada bloque. Si el sistema crea un bloque nuevo, inserta el primer valor de la clave de búsqueda (en el orden de la clave de búsqueda) que aparezca en el nuevo bloque del índice. Por otra parte, si el nuevo registro tiene el menor valor de la clave de búsqueda en su bloque, el sistema actualiza la entrada del índice que apunta al bloque; si no, el sistema no realiza ningún cambio sobre el índice.
- **Borrado**. Para borrar un registro, primero se busca el índice a borrar. Las acciones que emprende el sistema a continuación dependen de si el índice es denso o disperso.
  - Índices densos:
    1. Si el registro borrado era el único registro con ese valor de la clave de búsqueda, el sistema borra el registro índice correspondiente del índice.
    2. En caso contrario se emprenden las siguientes acciones:
       a. Si el registro índice almacena punteros a todos los registros con el mismo valor de la clave de búsqueda, el sistema borra del registro índice el puntero al registro borrado.
       b. En caso contrario, el registro índice almacena un puntero sólo al primer registro con el valor de la clave de búsqueda. En este caso, si el registro borrado era el primer registro con el valor de la clave de búsqueda, el sistema actualiza el registro índice para apuntar al siguiente registro.
  - Índices dispersos:
    1. Si el índice no contiene un registro índice con el valor de la clave de búsqueda del registro borrado, no hay que hacer nada.
    2. En caso contrario se emprenden las siguientes acciones:
       a. Si el registro borrado era el único registro con la clave de búsqueda, el sistema reemplaza el registro índice correspondiente con un registro índice para el siguiente valor de la clave de búsqueda (en el orden de la clave de búsqueda). Si el siguiente valor de la clave de búsqueda ya tiene una entrada en el índice, se borra en lugar de reemplazarla.

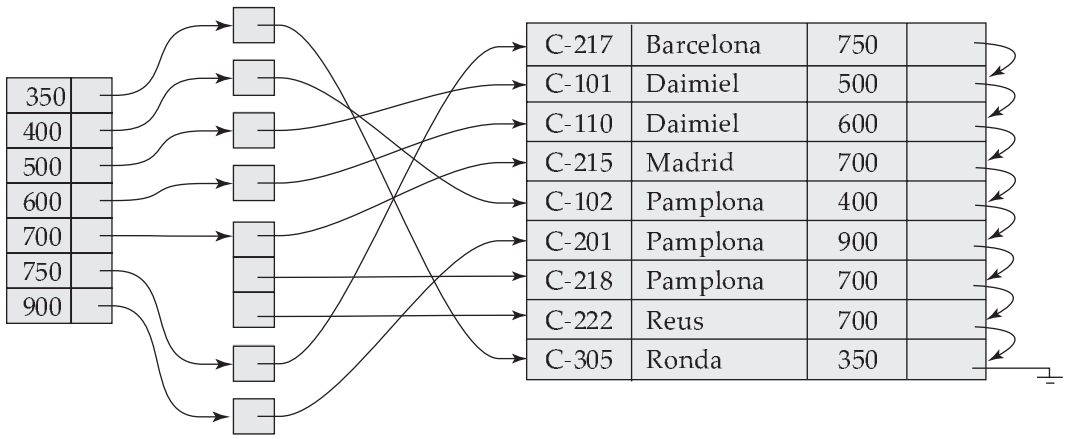


Figura 12.5 Índice secundario del archivo *cuenta*, con la clave no candidata *saldo*.

b. En caso contrario, si el registro índice para el valor de la clave de búsqueda apunta al registro a borrar, el sistema actualiza el registro índice para que apunte al siguiente registro con el mismo valor de la clave de búsqueda.

Los algoritmos de inserción y borrado para los índices multinivel se extienden de manera sencilla a partir del esquema descrito anteriormente. Al borrar o al insertar se actualiza el índice de nivel más bajo como se describió anteriormente. Por lo que respecta al índice del segundo nivel, el índice de nivel más bajo es simplemente un archivo de registros—así, si hay algún cambio en el índice de nivel más bajo, se tendrá que actualizar el índice del segundo nivel como ya se describió. La misma técnica se aplica al resto de niveles del índice, si los hubiera.

12.2.4 Índices secundarios

Los índices secundarios deben ser densos, con una entrada en el índice por cada valor de la clave de búsqueda, y un puntero a cada registro del archivo. Un índice con agrupación puede ser disperso, almacenando sólo algunos de los valores de la clave de búsqueda, ya que siempre es posible encontrar registros con valores de la clave de búsqueda intermedios mediante un acceso secuencial a parte del archivo, como se describió antes. Si un índice secundario almacena sólo algunos de los valores de la clave de búsqueda, los registros con los valores de la clave de búsqueda intermedios pueden estar en cualquier lugar del archivo y, en general, no se pueden encontrar sin explorar el archivo completo.

Un índice secundario sobre una clave candidata es como un índice denso con agrupación, excepto en que los registros apuntados por los sucesivos valores del índice no están almacenados secuencialmente. Generalmente los índices secundarios están estructurados de manera diferente a como lo están los índices con agrupación. Si la clave de búsqueda de un índice con agrupación no es una clave candidata, es suficiente si el valor de cada entrada en el índice apunta al primer registro con ese valor en la clave de búsqueda, ya que los otros registros podrían ser alcanzados por una búsqueda secuencial del archivo.

En cambio, si la clave de búsqueda de un índice secundario no es una clave candidata, no sería suficiente apuntar sólo al primer registro de cada valor de la clave. El resto de registros con el mismo valor de la clave de búsqueda podría estar en cualquier otro sitio del archivo, ya que los registros están ordenados según la clave de búsqueda del índice con agrupación, en vez de la clave de búsqueda del índice secundario. Por tanto, un índice secundario debe contener punteros a todos los registros.

Se puede usar un nivel adicional de referencia para implementar los índices secundarios sobre claves de búsqueda que no sean claves candidatas. Los punteros en estos índices secundarios no apuntan directamente al archivo. En vez de eso, cada puntero apunta a un cajón que contiene punteros al archivo. En la Figura 12.5 se muestra la estructura del archivo *cuenta*, con un índice secundario que usa un nivel de referencia adicional, y teniendo como clave de búsqueda el *saldo.*

Siguiendo el orden de un índice primario, una búsqueda secuencial es eficiente porque los registros del archivo están guardados físicamente de la misma manera a como está ordenado el índice. Sin embar-

go, no se puede (salvo en casos excepcionales) almacenar el archivo ordenado físicamente por el orden de la clave de búsqueda del índice con agrupación y la clave de búsqueda del índice secundario. Ya que el orden de la clave secundaria y el orden físico difieren, si se intenta examinar el archivo secuencialmente por el orden de la clave secundaria, es muy probable que la lectura de cada bloque suponga la lectura de un nuevo bloque del disco, lo cual es muy lento.

El procedimiento ya descrito para borrar e insertar se puede aplicar también a los índices secundarios; las acciones a emprender son las descritas para los índices densos que almacenan un puntero a cada registro del archivo. Si un archivo tiene varios índices, siempre que se modifique el archivo, se debe actualizar *cada* uno de ellos.

Los índices secundarios mejoran el rendimiento de las consultas que usan claves que no son la de búsqueda del índice con agrupación. Sin embargo, implican un tiempo adicional importante al modificar la base de datos. El diseñador de la base de datos debe decidir qué índices secundarios son deseables, según una estimación sobre las frecuencias de las consultas y de las modificaciones.

12.3 Archivos de índices de árbol B$^+$

El inconveniente principal de la organización de un archivo secuencial indexado reside en que el rendimiento, tanto para buscar en el índice como para buscar secuencialmente a través de los datos, se degrada según crece el archivo. Aunque esta degradación se puede remediar reorganizando el archivo, no es deseable hacerlo frecuentemente.

La estructura de índice de **árbol B$^+$** es la más extendida de las estructuras de índices que mantienen su eficiencia a pesar de la inserción y borrado de datos. Un índice de árbol B$^+$ toma la forma de un **árbol equilibrado** donde los caminos de la raíz a cada hoja del árbol son de la misma longitud. Cada nodo que no sea hoja tiene entre $\lceil n/2 \rceil$ y n hijos, donde n es fijo para cada árbol concreto.

Se verá que la estructura de árbol B$^+$ implica una degradación del rendimiento al insertar y al borrar, además de un espacio extra. Este tiempo adicional es aceptable incluso en archivos con altas frecuencias de modificación, ya que se evita el coste de reorganizar el archivo. Además, puesto que los nodos podrían estar a lo sumo medio llenos (si tienen el mínimo número de hijos) se desperdicia algo de espacio. Este gasto de espacio adicional también es aceptable dados los beneficios en el rendimiento aportados por las estructura de árbol B$^+$.

12.3.1 Estructura de árbol B$^+$

Un índice de árbol B$^+$ es un índice multinivel pero con una estructura que difiere del índice multinivel de un archivo secuencial. En la Figura 12.6 se muestra un nodo típico de un árbol B$^+$. Puede contener hasta $n - 1$ claves de búsqueda $K_1, K_2, \ldots, K_{n-1}$ y n punteros $P_1, P_2, \ldots, P_n$. Los valores de la clave de búsqueda de un nodo se mantienen ordenados; así, si $i < j$, entonces $K_i < K_j$.

Considérese primero la estructura de los nodos hoja. Para $i = 1, 2, \ldots, n-1$, el puntero P_i apunta, o bien a un registro del archivo con valor de la clave de búsqueda K_i, o bien a un cajón de punteros, cada uno de los cuales apunta a un registro del archivo con valor de la clave de búsqueda K_i. La estructura cajón se usa solamente si la clave de búsqueda no es una clave candidata y si el archivo no está ordenado según la clave de búsqueda (en el Apartado 12.5.3 se estudia la forma de evitar la creación de cajones haciendo que parezca que la clave de búsqueda sea única). El puntero P_n tiene un propósito especial que se estudiará más adelante.

En la Figura 12.7 se muestra un nodo hoja en el árbol B$^+$ del archivo *cuenta*, donde n vale tres y la clave de búsqueda es *nombre_sucursal*. Obsérvese que, como el archivo cuenta está ordenado por *nombre_sucursal*, los punteros en el nodo hoja apuntan directamente al archivo.

Ahora que se ha visto la estructura de un nodo hoja, se mostrará cómo los valores de la clave de búsqueda se asignan a nodos concretos. Cada hoja puede guardar hasta $n - 1$ valores. Está permitido

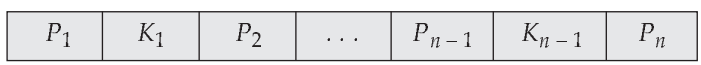


Figura 12.6 Nodo típico de un árbol B$^+$.

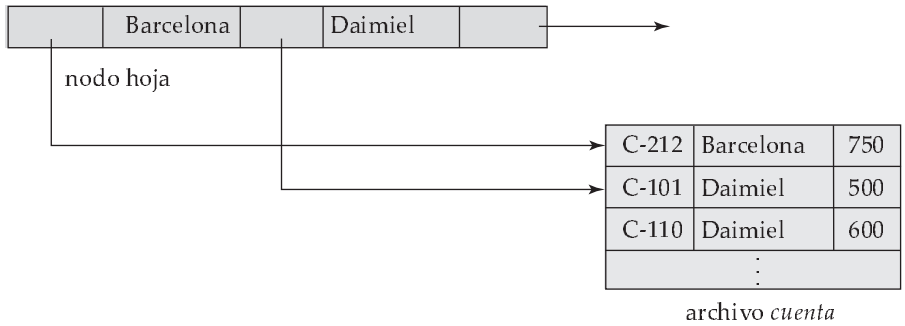


Figura 12.7 Nodo hoja para el índice del árbol B+ de *cuenta* ($n = 3$).

que los nodos hojas contengan al menos $\lceil (n-1)/2 \rceil$ valores. Los rangos de los valores en cada hoja no se solapan. Así, si L_i y L_j son nodos hoja e $i < j$, entonces cada valor de la clave de búsqueda en L_i es menor que cada valor de la clave en L_j. Si el índice de árbol B+ es un índice denso, cada valor de la clave de búsqueda debe aparecer en algún nodo hoja.

Ahora se puede explicar el uso del puntero P_n. Dado que existe un orden lineal en las hojas basado en los valores de la clave de búsqueda que contienen, se usa P_n para encadenar juntos los nodos hojas en el orden de la clave de búsqueda. Esta ordenación permite un procesamiento secuencial eficaz del archivo.

Los nodos internos del árbol B+ forman un índice multinivel (disperso) sobre los nodos hoja. La estructura de los nodos internos es la misma que la de los nodos hoja, excepto que todos los punteros son punteros a nodos del árbol. Un nodo interno podría guardar hasta *n* punteros y *debe* guardar al menos $\lceil n/2 \rceil$ punteros. El número de punteros de un nodo se llama *grado de salida* del nodo.

Considérese un nodo que contiene *m* punteros. Para $i = 2, 3, \ldots, m-1$, el puntero P_i apunta al subárbol que contiene los valores de la clave de búsqueda menores que K_i y mayor o igual que K_{i-1}. El puntero P_m apunta a la parte del subárbol que contiene los valores de la clave mayores o iguales que K_{m-1}, y el puntero P_1 apunta a la parte del subárbol que contiene los valores de la clave menores que K_1.

A diferencia de otros nodos internos, el nodo raíz puede tener menos de $\lceil n/2 \rceil$; sin embargo, debe tener al menos dos punteros, salvo que el árbol tenga un solo nodo. Siempre es posible construir un árbol B+, para cualquier *n*, que satisfaga los requisitos anteriores. En la Figura 12.8 se muestra un árbol B+ completo para el archivo *cuenta* ($n = 3$). Por simplicidad se omiten los punteros al propio archivo y los punteros nulos. Como ejemplo de un árbol B+ en el cual la raíz debe tener menos de $\lceil n/2 \rceil$ valores, en la Figura 12.9 se muestra un árbol B+ para el archivo *cuenta* con $n = 5$.

En todos los ejemplos mostrados de árboles B+, éstos están equilibrados. Es decir, la longitud de cada camino desde la raíz a cada nodo hoja es la misma. Esta propiedad es un requisito de los árboles B+. De hecho, la "B" de árbol B+ proviene del inglés "*balanced*" (equilibrado). Es esta propiedad de equilibrio de los árboles B+ la que asegura un buen rendimiento para las búsquedas, inserciones y borrados.

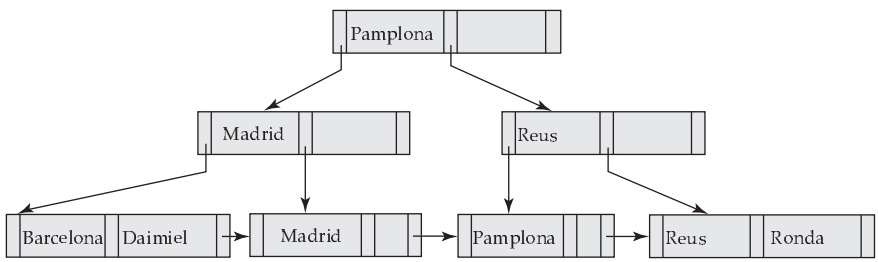


Figura 12.8 Árbol B+ para el archivo *cuenta* ($n = 3$).

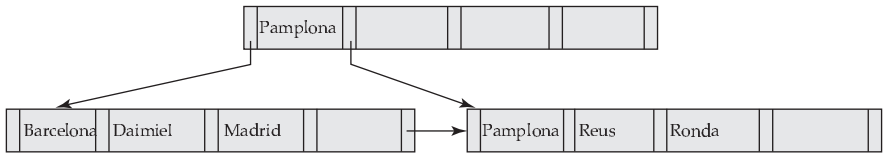


Figura 12.9 Árbol B$^+$ para el archivo *cuenta* ($n = 5$).

12.3.2 Consultas con árboles B$^+$

Considérese ahora cómo procesar consultas usando árboles B$^+$. Supóngase que se desea encontrar todos los registros cuyo valor de la clave de búsqueda sea V. La Figura 12.10 muestra el pseudocódigo para hacerlo. Primero se examina el nodo raíz para buscar el menor valor de la clave de búsqueda mayor que V. Supóngase que este valor de la clave de búsqueda es K_i. Después se sigue el puntero P_i hasta otro nodo. Si no se encuentra ese valor, entonces $k \geq K_{m-1}$, donde m es el número de punteros del nodo. Es este caso se sigue P_m hasta otro nodo. En el nodo alcanzado anteriormente se busca de nuevo el menor valor de la clave de búsqueda que es mayor que V para seguir el puntero correspondiente. Finalmente se alcanza un nodo hoja. En este nodo hoja, si se encuentra que el valor K_i es igual a V, entonces el puntero P_i nos ha conducido al registro o cajón deseado. Si no se encuentra el valor V en el nodo hoja, no existe ningún registro con el valor clave V.

De esta manera, para procesar una consulta, se tiene que recorrer un camino en el árbol desde la raíz hasta algún nodo hoja. Si hay K valores de la clave de búsqueda en el archivo, este camino no será más largo que $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.

En la práctica sólo se accede a unos cuantos nodos. Generalmente un nodo se construye para tener el mismo tamaño que un bloque de disco, el cual ocupa normalmente 4 KB . Con una clave de búsqueda del tamaño de 12 bytes y un tamaño del puntero a disco de 8 bytes, n está alrededor de 200. Incluso con una estimación más conservadora de 32 bytes para el tamaño de la clave de búsqueda, n está entorno a 100. Con $n = 100$, si se tienen un millón de valores de la clave de búsqueda en el archivo, una búsqueda necesita solamente $\lceil \log_{50}(1.000.000) \rceil = 4$ accesos a nodos. Por tanto, se necesitan leer a lo sumo cuatro bloques del disco para realizar la búsqueda. Normalmente, se accede mucho al nodo raíz del árbol y, por ello, suele guardarse en una memoria intermedia; por tanto, solamente hace falta leer, como máximo, tres bloques del disco.

Una diferencia importante entre las estructuras de árbol B$^+$ y los árboles en memoria, tales como los árboles binarios, está en el tamaño de los nodos y, por tanto, la altura del árbol. En los árboles binarios, todos los nodos son pequeños y tienen, a lo sumo, dos punteros. En los árboles B$^+$, cada nodo es grande —normalmente un bloque del disco—y puede tener gran número de punteros. Así, los árboles B$^+$ tienden a ser bajos y anchos, en lugar de los altos y estrechos árboles binarios. En un árbol equilibrado, el camino de una búsqueda puede tener una longitud de $\lceil \log_2(K) \rceil$, donde K es el número de valores

```
procedure buscar(valor V)
    C = nodo raíz
    while C no sea un nodo raíz begin
        K_i = mínimo valor de la clave de búsqueda, si lo hay, mayor que V
        if no hay tal valor then begin
            m = número de punteros en el nodo
            C = nodo al que apunta P_m
        end
            else C = el nodo al que apunta P_i
    end
    if hay un valor de la clave K_i en C tal que K_i = V
        then el puntero P_i conduce al registro o cajón deseado
        else no existe ningún registro con el valor de la clave k
```

Figura 12.10 Consulta con un árbol B$^+$.

de la clave de búsqueda. Con $K = 1.000.000$, como en el ejemplo anterior, un árbol binario equilibrado necesita alrededor de 20 accesos a nodos. Si cada nodo estuviera en un bloque del disco distinto, serían necesarias 20 lecturas a bloques para procesar la búsqueda, en contraste con las cuatro lecturas del árbol B+.

12.3.3 Actualizaciones en árboles B+

El borrado y la inserción son operaciones más complicadas que las búsquedas, ya que podría ser necesario **dividir** un nodo que fuese demasiado grande como resultado de una inserción, o **fusionar** nodos si un nodo se volviera demasiado pequeño (menor que $\lceil n/2 \rceil$ punteros). Además, cuando se divide un nodo o se fusionan un par de ellos, se debe asegurar que el equilibrio del árbol se mantenga. Para presentar la idea que hay detrás del borrado y la inserción en un árbol B+, se asumirá que los nodos nunca serán demasiado grandes ni demasiado pequeños. Bajo esta suposición, el borrado y la inserción se realizan como se indica a continuación.

- **Inserción**. Usando la misma técnica que para buscar, se busca un nodo hoja donde tendría que aparecer el valor de la clave de búsqueda. Si el valor de la clave de búsqueda ya aparece en el nodo hoja, se inserta un nuevo registro en el archivo y, si es necesario, un puntero al cajón. Si el valor de la clave de búsqueda no aparece, se inserta el valor en el nodo hoja de tal manera que las claves de búsqueda permanezcan ordenadas. Luego se inserta el nuevo registro en el archivo y, si es necesario, se crea un nuevo cajón con el puntero apropiado.
- **Borrado**. Usando la misma técnica que para buscar, se busca el registro a borrar y se elimina del archivo. Si no existe un cajón asociado con el valor de la clave de búsqueda o si el cajón se queda vacío como resultado del borrado, se borra el valor de la clave de búsqueda del nodo hoja.

A continuación se considera un ejemplo en el que se tiene que dividir un nodo. Por ejemplo, insertar un registro en el árbol B+ de la Figura 12.8, cuyo valor de *nombre_sucursal* es "Cádiz". Usando el algoritmo de búsqueda, "Cádiz" debería aparecer en el nodo que incluye "Barcelona" y "Daimiel". No hay sitio para insertar el valor de la clave de búsqueda "Cádiz". Por tanto, se *divide* el nodo en otros dos nodos. En la Figura 12.11 se muestran los nodos hoja que resultan de insertar "Cádiz" y de dividir el nodo que incluía "Barcelona" y "Daimiel". En general, si existen n valores de la clave de búsqueda (los $n-1$ valores del nodo hoja más el valor que se inserta), se asignará $\lceil n/2 \rceil$ al nodo existente y el resto de valores en el nuevo nodo.

Después de dividir el nodo hoja hay que insertar el nuevo nodo hoja en el árbol B+. En el nuevo nodo del ejemplo, "Daimiel" es el valor más pequeño de la clave de búsqueda. Después hay que insertar este valor de la clave de búsqueda en el padre del nodo hoja dividido. En el árbol B+ de la Figura 12.12 se muestra el resultado de la inserción. El valor "Daimiel" de la clave de búsqueda se ha insertado en el padre. Ha sido posible llevar a cabo esta inserción porque había sitio para añadir un valor de la clave de búsqueda. En caso de no haber sitio se dividiría el padre. En el peor de los casos, todos los nodos en el camino hacia la raíz se tendrían que dividir. Si la propia raíz se tuviera que dividir, el árbol hubiera sido más profundo.

La técnica general para la inserción en un árbol B+ es determinar el nodo hoja h en el cual realizar la inserción. Si es necesario dividir, se inserta el nuevo nodo dentro del padre del nodo h. Si esta inserción produce otra división, se procedería recursivamente o bien hasta que una inserción no produzca otra división o bien hasta crear una nueva raíz.

En la Figura 12.13 se bosqueja el algoritmo de inserción en pseudocódigo. El procedimiento inserta un par valor de la clave y puntero en el índice usando los procedimientos insertar_en_hoja e inser-

Figura 12.11 División del nodo hoja tras la inserción de "Cádiz".

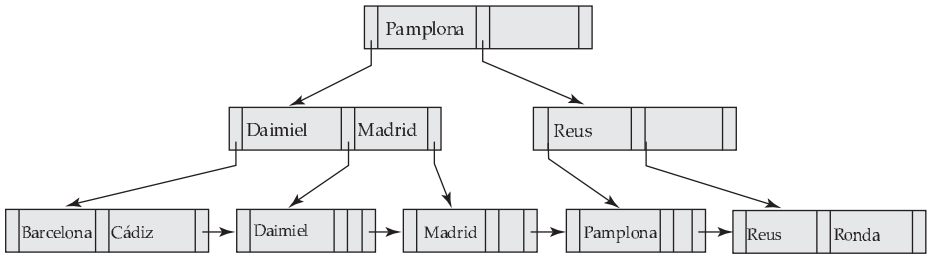


Figura 12.12 Inserción de "Cádiz" en el árbol B$^+$ de la Figura 12.8.

tar_en_padre. En el pseudocódigo L, N, P y T denotan punteros a nodos y L se usa para denotar un nodo hoja. $L.K_i$ y $L.P_i$ denotan el i-ésimo valor y el i-ésimo puntero en el nodo L, respectivamente. $T.K_i$ y $T.P_i$ se usan de forma análoga. El pseudocódigo también emplea la función $padre(N)$ para encontrar el padre del nodo N. Se puede obtener una lista de los nodos en el camino de la raíz a la hoja al buscar inicialmente el nodo hoja, y se puede usar después para encontrar eficazmente el padre de cualquier nodo del camino.

El procedimiento insertar_en_padre tiene como parámetros N, K' y N', donde el nodo N se ha dividido en N y N', siendo K' el valor mínimo en N'. El procedimiento modifica el padre de N para registrar la división. Los procedimientos insertar_en_hoja e insertar_en_padre usan el área temporal de memoria T para almacenar los contenidos del nodo que se está dividiendo. Los procedimientos se pueden modificar para que copien directamente los datos del nodo que se divide en el nodo que se crea, reduciendo el tiempo necesario para la copia. Sin embargo, el uso del espacio temporal T simplifica los procedimientos.

A continuación se consideran los borrados que provocan que el árbol se quede con muy pocos punteros. Primero se borra "Daimiel" del árbol B$^+$ de la Figura 12.12. Para ello se localiza la entrada "Daimiel" usando el algoritmo de búsqueda. Cuando se borra la entrada "Daimiel" de su nodo hoja, la hoja se queda vacía. Ya que en el ejemplo $n = 3$ y $0 < \lceil (n-1)/2 \rceil$, este nodo se debe borrar del árbol B$^+$. Para borrar un nodo hoja se tiene que borrar el puntero que le llega desde su padre. En el ejemplo, este borrado deja al nodo padre, el cual contenía tres punteros, con sólo dos punteros. Ya que $2 \geq \lceil n/2 \rceil$, el nodo es todavía lo suficientemente grande y la operación de borrado se completa. El árbol B$^+$ resultante se muestra en la Figura 12.14.

Cuando un borrado se hace sobre el padre de un nodo hoja, el propio nodo padre podría quedar demasiado pequeño. Esto es exactamente los que ocurre si se borra "Pamplona" del árbol B$^+$ de la Figura 12.14. El borrado de la entrada Pamplona provoca que un nodo hoja se quede vacío. Cuando se borra el puntero a este nodo en su padre, éste sólo se queda con un puntero. Así, $n = 3$, $\lceil n/2 \rceil = 2$ y queda tan sólo un puntero, que es demasiado poco. Sin embargo, ya que el nodo padre contiene información útil, no basta simplemente con borrarlo. En vez de eso, se busca el nodo hermano (el nodo interno que contiene al menos una clave de búsqueda, Madrid). Este nodo hermano dispone de sitio para colocar la información contenida en el nodo que quedó demasiado pequeño, así que se fusionan estos nodos, de tal manera que el nodo hermano ahora contiene las claves "Madrid" y "Reus". El otro nodo (el nodo que contenía solamente la clave de búsqueda "Reus") ahora contiene información redundante y se puede borrar desde su padre (el cual resulta ser la raíz del ejemplo). En la Figura 12.15 se muestra el resultado. Hay que observar que la raíz tiene solamente un puntero hijo como resultado del borrado, así que éste se borra y el hijo solitario se convierte en la nueva raíz. De esta manera la profundidad del árbol ha disminuido en una unidad.

No siempre es posible fusionar nodos. Como ejemplo se borrará "Pamplona" del árbol B$^+$ de la Figura 12.11. En este ejemplo, la entrada "Daimiel" es todavía parte del árbol. Una vez más, el nodo hoja que contiene "Pamplona" se queda vacío. El padre del nodo hoja se queda también demasiado pequeño (únicamente con un puntero). De cualquier modo, en este ejemplo, el nodo hermano contiene ya el máximo número de punteros: tres. Así, no puede acomodar a un puntero adicional. La solución en este caso es **redistribuir** los punteros de tal manera que cada hermano tenga dos punteros. El resultado se muestra en la Figura 12.16. Obsérvese que la redistribución de los valores necesita de un cambio en el valor de la clave de búsqueda en el padre de los dos hermanos.

```
procedure insertar(valor K, puntero P)
    hallar el nodo hoja L que debe contener el valor de la clave K
    if (L tiene menos de n − 1 valores de la clave)
        then insertar_en_hoja (L, K, P)
        else begin /* L ya tiene n − 1 valores de la clave, dividirlo */
            Crear nodo L'
            Copiar L.P1 . . . L.Kn−1 a un bloque de memoria T que pueda
                almacenar n pares (puntero, valor de la clave)
            insertar_en_hoja (T, K, P)
            L'.Pn = L.Pn; L.Pn = L'
            Borrar desde L.P1 hasta L.Kn−1 de L
            Copiar desde T.P1 hasta T.K⌈n/2⌉ de T a L comenzando en L.P1
            Copiar desde T.P⌈n/2⌉+1 hasta T.Kn de T a L' comenzando en L'.P1
            K' el valor mínimo de la clave en L'
            insertar_en_padre(L, K', L')
        end

procedure insertar_en_hoja (nodo L, valor K, puntero P)
    if K es menor que L.K1
        then insertar P, K en L justo antes de L.P1
        else begin
            Ki el mayor valor de L que sea menor que K
            insertar P, K en L justo después de T.Ki
        end

procedure insertar_en_padre(nodo N, valor K', nodo N')
    if N es la raíz del árbol
        then begin
            crear un nuevo nodo R que contenga N, K', N'    /* N y N' son punteros */
            hacer de R la raíz del árbol
            return
        end
    P = padre (N)
    if (P tiene menos de n punteros)
        then insertar (K', N') en P justo después de N
        else begin /* Dividir P */
            Copiar P a un bloque de memoria T que pueda almacenar P y (K', N')
            Insertar (K', N') en T justo después de N
            Borrar todas las entradas de P; Crear el nodo P'
            Copiar T.P1 . . . T.P⌈n/2⌉ en P
            K'' = T.K⌈n/2⌉
            Copiar T.P⌈n/2⌉+1 . . . T.Pn+1 en P'
            insertar_en_padre(P, K'', P')
        end
```

Figura 12.13 Inserción de una entrada en un árbol B$^+$.

En general, para borrar un valor en un árbol B$^+$ se realiza una búsqueda según el valor y se borra. Si el nodo es demasiado pequeño, se borra desde su padre. Este borrado se realiza como una aplicación recursiva del algoritmo de borrado hasta que se alcanza la raíz, un nodo padre queda lleno de manera adecuada después de borrar, o hasta aplicar una redistribución.

En la Figura 12.17 se describe el pseudocódigo para el borrado en un árbol B$^+$. El procedimiento intercambiar_variables(N, N') simplemente cambia de lugar los valores (punteros) de las variables N y N'; este cambio no afecta al árbol en sí mismo. El pseudocódigo utiliza la condición "muy pocos va-

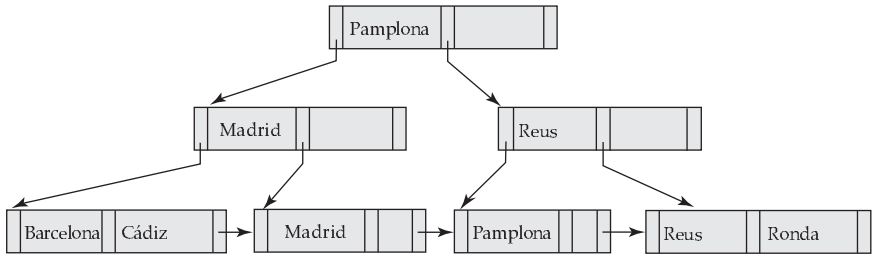


Figura 12.14 Borrado de "Daimiel" del árbol B$^+$ de la Figura 12.12.

lores/punteros". Para nodos internos, este criterio quiere decir: menos que $\lceil n/2 \rceil$ punteros; para nodos hoja, quiere decir: menos que $\lceil (n-1)/2 \rceil$ valores. El pseudocódigo realiza la redistribución tomando prestada una sola entrada desde un nodo adyacente. También se puede redistribuir mediante la distribución equitativa de entradas entre dos nodos. El pseudocódigo hace referencia al borrado de una entrada (K, P) desde un nodo. En el caso de los nodos hoja, el puntero a una entrada realmente precede al valor de la clave; así, el puntero P precede al valor de la clave K. Para nodos internos, P sigue al valor de la clave K.

Es interesante señalar que, como resultado de un borrado, puede que haya valores de la clave de nodos internos de un árbol B$^+$ que no estén en ninguna hoja del árbol.

Aunque las operaciones inserción y borrado en árboles B$^+$ son complicadas, requieren relativamente pocas operaciones E/S, lo que es un beneficio importante dado su coste. Se puede demostrar que el número de operaciones E/S necesarias para una inserción o borrado es, en el peor de los casos, proporcional a $\log_{\lceil n/2 \rceil}(K)$, donde n es el número máximo de punteros en un nodo y K es el número de valores de la clave de búsqueda. En otras palabras, el coste de las operaciones inserción y borrado es proporcional a la altura del árbol B$^+$ y es, por tanto, bajo. Debido a la velocidad de las operaciones en los árboles B$^+$, estas estructuras de índice se usan frecuentemente al implementar las bases de datos.

12.3.4 Organización de archivos con árboles B$^+$

Como se mencionó en el Apartado 12.3, el inconveniente de la organización de archivos secuenciales de índices es la degradación del rendimiento según crece el archivo: con el crecimiento, un porcentaje mayor de registros índice y registros reales se desaprovechan y se almacenan en bloques de desbordamiento. La degradación de las búsquedas en el índice se resuelve mediante el uso de índices de árbol B$^+$ en el archivo. También se soluciona el problema de la degradación al almacenar los registros reales

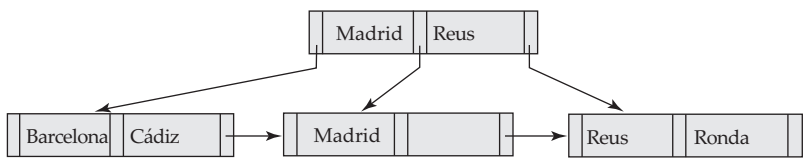


Figura 12.15 Borrado de "Pamplona" del árbol B$^+$ de la Figura 12.14.

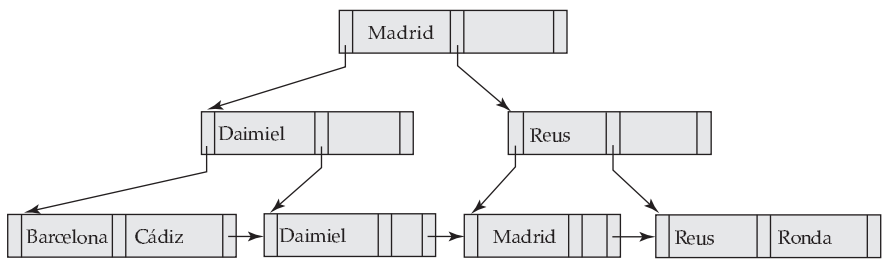


Figura 12.16 Borrado de "Pamplona" del árbol B$^+$ de la Figura 12.12.

```
procedure borrar(valor K, puntero P)
    hallar el nodo hoja L que contiene (K, P)
    borrar_entrada(L, K, P)

procedure borrar_entrada(nodo N, valor K, puntero P)
    borrar (K, P) de N
    if (N es la raíz and a N sólo le queda un hijo)
    then hacer del hijo de N la nueva raíz del árbol y borrar N
    else if (N tiene muy pocos valores/punteros) then begin
        sea N' el hijo anterior o siguiente de padre(N)
        sea K' el valor entre los punteros N y N' en padre(N)
        if (las entradas en N y N' caben en un solo nodo)
            then begin /* Fusionar los nodos */
                if (N es predecesor de N') then intercambiar_variables(N, N')
                if (N no es una hoja)
                    then concatenar K' y todos los punteros y valores en N a N'
                    else  concatenar todos los pares (K_i, P_i) en N a N'; sea N'.P_n = N.P_n
                borrar_entrada(padre(N), K', N); borrar el nodo N
            end
        else begin /* Redistribución: tomar prestada una entrada de N' */
            if (N' es predecesor de N) then begin
                if (N es un nodo interno) then begin
                    sea m tal que N'.P_m es el último puntero en N'
                    suprimir (N'.K_{m-1}, N'.P_m) de N'
                    insertar (N'.P_m, K') como primer puntero y valor en N,
                        desplazando otros punteros y valores a la derecha
                    sustituir K' en padre(N) por N'.K_{m-1}
                end
                else begin
                    sea m tal que (N'.P_m, N'.K_m) es el último par puntero/valor
                        en N'
                    suprimir (N'.P_m, N'.K_m) de N'
                    insertar (N'.P_m, N'.K_m) como primer puntero y valor en N,
                        desplazando otros punteros y valores a la derecha
                    sustituir K' en padre(N) por N'.K_m
                end
            end
            else  ... simétrico al caso then ...
        end
    end
```

Figura 12.17 Borrado de una entrada de un árbol B+.

utilizando el nivel de hoja del árbol B+ para almacenar los registros reales en los bloques. En estas estructuras, la estructura del árbol B+ se usa no sólo como un índice, sino también como un organizador de los registros dentro del archivo. En la **organización de archivo con árboles B+**, los nodos hoja del árbol almacenan registros, en lugar de almacenar punteros a registros. En la Figura 12.18 se muestra un ejemplo de la organización de un archivo con un árbol B+. Ya que los registros son normalmente más grandes que los punteros, el número máximo de registros que se pueden almacenar en un nodo hoja es menor que el número de punteros en un nodo interno. Sin embargo, todavía se requiere que los nodos hoja estén llenos al menos hasta la mitad.

La inserción y el borrado de registros en las organizaciones de archivos con árboles B+ se tratan del mismo modo que la inserción y el borrado de entradas en los índices de árboles B+. Cuando se inserta un registro con un valor de clave v, se localiza el bloque que debería contener ese registro mediante la

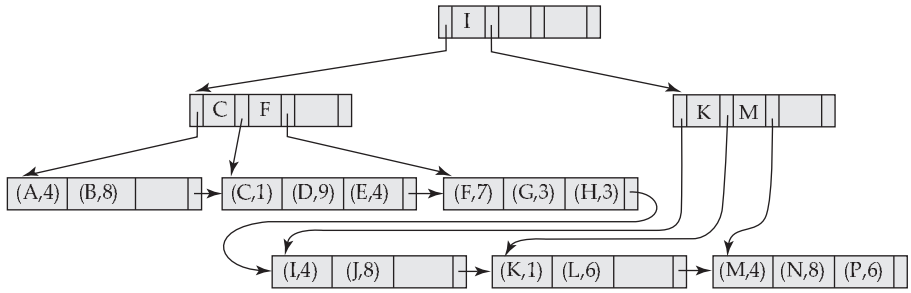


Figura 12.18 Organizacion de archivos con árboles B^+.

búsqueda en el árbol B^+ de la mayor clave que sea $\leq v$. Si el bloque localizado tiene bastante espacio libre para el registro, se almacena el registro en el bloque. De no ser así, como en una inserción en un árbol B^+, se divide el bloque en dos y se redistribuyen sus registros (en el orden de la clave del árbol B^+) creando espacio para el nuevo registro. Esta división se propaga hacia arriba en el árbol B+ de la manera usual. Cuando se borra un registro, primero se elimina del bloque que lo contiene. Si como resultado un bloque B llega a ocupar menos que la mitad, los registros en B se redistribuyen con los registros en un bloque B' adyacente. Si se asume que los registros son de tamaño fijo, cada bloque contendrá por lo menos la mitad de los registros que pueda contener como máximo. Los nodos internos del árbol B^+ se actualizan por tanto de la manera habitual.

Cuando un árbol B^+ se utiliza para la organización de un archivo, la utilización del espacio es particularmente importante, ya que el espacio ocupado por los registros es mucho mayor que el espacio ocupado por las claves y punteros. Se puede mejorar la utilización del espacio en un árbol B^+ implicando a más nodos hermanos en la redistribución durante las divisiones y fusiones. La técnica es aplicable a los nodos hoja y nodos internos y funciona como sigue.

Durante la inserción, si un nodo está lleno se intenta redistribuir algunas de sus entradas en uno de los nodos adyacentes para hacer sitio a una nueva entrada. Si este intento falla porque los nodos adyacentes están llenos, se divide el nodo y las entradas entre uno de los nodos adyacentes y los dos nodos que se obtienen al dividir el nodo original. Puesto que los tres nodos juntos contienen un registro más que puede encajar en dos nodos, cada nodo estará lleno aproximadamente hasta sus dos terceras partes. Para ser más precisos, cada nodo tendrá por lo menos $\lfloor 2n/3 \rfloor$ entradas, donde n es el número máximo de entradas que puede tener un nodo ($\lfloor x \rfloor$ denota el mayor entero menor o igual que x; es decir, se elimina la parte fraccionaria si la hay).

Durante el borrado de un registro, si la ocupación de un nodo está por debajo de $\lfloor 2n/3 \rfloor$, se intentará tomar prestada una entrada desde uno de sus nodos hermanos. Si ambos nodos hermanos tienen $\lfloor 2n/3 \rfloor$ registros, en lugar de tomar prestada una entrada, se redistribuyen las entradas en el nodo y en los dos nodos hermanos uniformemente entre dos de los nodos y se borra el tercer nodo. Se puede usar este enfoque porque el número total de entradas es $3\lfloor 2n/3 \rfloor - 1$, lo cual es menor que $2n$. Utilizando tres nodos adyacentes para la redistribución se puede garantizar que cada nodo tenga $\lfloor 3n/4 \rfloor$ entradas. En general, si hay m nodos ($m - 1$ hermanos) implicados en la redistribución se puede garantizar que cada nodo contenga, al menos, $\lfloor (m - 1)n/m \rfloor$ entradas. Sin embargo, el coste de la actualización se vuelve mayor según haya más nodos hermanos involucrados en la redistribución.

Obsérvese que en un índice de árbol B^++ los nodos hoja adyacentes en el árbol se puede ubicar en diferentes lugares del disco. Cuando se crea un índice sobre un conjunto de registros es posible asignar bloques contiguos en disco para nodos hoja que también lo sean en el árbol. Así, una exploración secuencial de los nodos hoja correspondería a una exploración secuencial en el disco. Como se dan inserciones y borrados en el árbol, se pierde la secuencialidad, y las esperas a disco en el acceso secuencial se incrementan. Sería necesario una reorganización del índice para mantener la secuencialidad.

Las organizaciones de archivo del tipo árbol B^+ se pueden usar para almacenar grandes objetos, como los tipos clob y blob de SQL, que pueden ser más grandes que un bloque de disco y de tamaño de varios gigabytes. Estos objetos se pueden almacenar dividiéndolos en secuencias de registros más pequeños que se organizan en un archivo B^+. Los registros se pueden numerar secuencialmente o por

el desplazamiento en bytes del registro dentro del objeto, y este número se puede usar como clave de búsqueda.

12.3.5 Índices sobre cadenas de caracteres

La creación de índices de árboles B^+ sobre atributos de tipo cadena de caracteres plantea dos problemas. El primero es que las cadenas pueden ser de longitud variable. El segundo es que pueden ser largas, lo que produce un grado de salida bajo y a una altura del árbol incrementada de manera acorde.

Con las claves de búsqueda de longitud variable varios nodos pueden tener diferentes grados de salida incluso estando llenos. Un nodo se debe dividir si está lleno, es decir, si no hay espacio para añadir una nueva entrada, independientemente de cuántas entradas contenga. Análogamente, los nodos se pueden fusionar o las entradas se pueden redistribuir dependiendo de la fracción de espacio que se use en los nodos, en lugar de basarse en el número máximo de entradas que el nodo pueda contener.

Se puede incrementar el grado de salida usando la técnica denominada **compresión del prefijo**. Con esta compresión no se almacena la clave de búsqueda completa en los nodos internos. Sólo se almacena el prefijo de la clave de búsqueda que sea suficiente para distinguir las claves de los subárboles bajo ella. Por ejemplo, si se indexa el nombre, el valor de la clave en un nodo interno podría ser un prefijo del nombre; sería suficiente almacenar "Silb" en un nodo interno en lugar del nombre completo "Silberschatz" si los valores más próximos en los dos subárboles bajo esta clave sean, por ejemplo, "Silas" y "Silver" respectivamente.

12.4 Archivos de índices de árbol B

Los *índices de árbol B* son similares a los índices de árbol B^+. La diferencia principal entre los dos enfoques es que un árbol B elimina el almacenamiento redundante de los valores de la clave búsqueda. En el árbol B^+ de la Figura 12.12, las claves de búsqueda "Daimiel", "Madrid", "Reus" y "Pamplona" aparecen dos veces. Cada valor de clave de búsqueda aparece en algún nodo hoja; algunos se repiten en nodos internos.

Los árboles B permiten que los valores de la clave de búsqueda aparezcan solamente una vez. En la Figura 12.19 se muestra un árbol B que representa las mismas claves de búsqueda que el árbol B^+ de la Figura 12.12. Ya que las claves de búsqueda no están repetidas en el árbol B, sería posible almacenar el índice usando menos nodos del árbol que con el correspondiente índice de árbol B^+. Sin embargo, puesto que las claves de búsqueda que aparecen en los nodos internos no aparecen en ninguna otra parte del árbol B, es necesario incluir un campo adicional para un puntero por cada clave de búsqueda de un nodo interno. Estos punteros adicionales apuntan a registros del archivo o a los cajones de la clave de búsqueda asociada.

En la Figura 12.20a aparece un nodo hoja generalizado de un árbol B; en la Figura 12.20b aparece un nodo interno. Los nodos hoja son como en los árboles B^+. En los nodos internos los punteros P_i son los punteros del árbol que se utilizan también para los árboles B^+, mientras que los punteros B_i en los nodos internos son punteros a cajones o registros del archivo. En la figura del árbol B generalizado hay

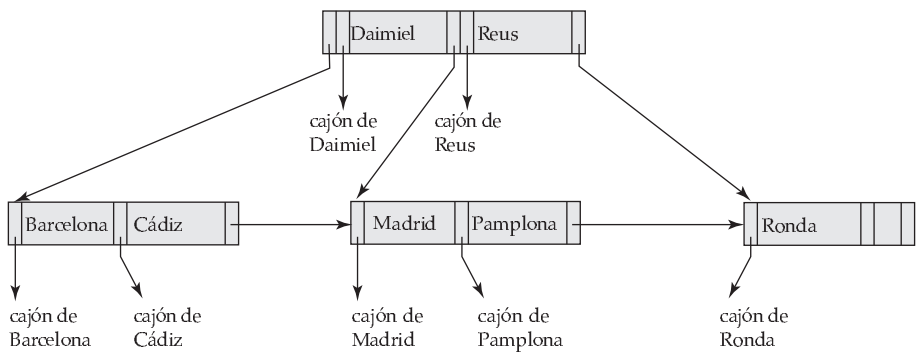


Figura 12.19 Árbol B equivalente al árbol B^+ de la Figura 12.12.

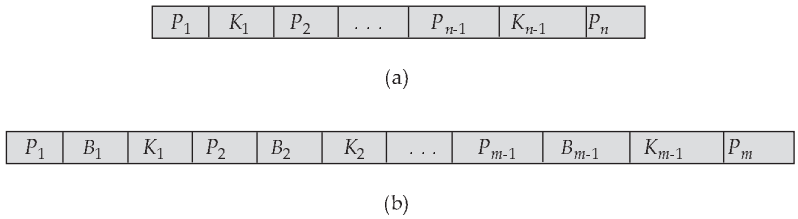


Figura 12.20 Nodos típicos de un árbol B. (a) Nodo hoja. (b) Nodo interno.

$n - 1$ claves en el nodo hoja, mientras que hay $m - 1$ claves en el nodo interno. Esta discrepancia ocurre porque los nodos internos deben incluir los punteros B_i, y de esta manera se reduce el número de claves de búsqueda que pueden contener estos nodos. Claramente, $m < n$, pero la relación exacta entre m y n depende del tamaño relativo de las claves de búsqueda y de los punteros.

El número de nodos a los que se accede en una búsqueda en un árbol B depende de dónde esté situada la clave de búsqueda. Una búsqueda en un árbol B^+ requiere atravesar un camino desde la raíz del árbol hasta algún nodo hoja. En cambio, algunas veces es posible encontrar en un árbol B el valor deseado antes de alcanzar el nodo hoja. Sin embargo, hay que realizar aproximadamente n accesos según cuántas claves haya almacenadas tanto en el nivel de hoja de un árbol B como en los niveles internos de hoja y, dado que n es normalmente grande, la probabilidad de encontrar ciertos valores pronto es relativamente pequeña. Por otra parte, el hecho de que aparezcan menos claves de búsqueda en los nodos internos del árbol B, comparado con los árboles B^+, implica que un árbol B tiene un grado de salida menor y, por tanto, puede que tenga una profundidad mayor que la correspondiente al árbol B^+. Así, la búsqueda en un árbol B es más rápida para algunas claves de búsqueda pero más lenta para otras, aunque en general, el tiempo de la búsqueda es todavía proporcional al logaritmo del número de claves de búsqueda.

El borrado en un árbol B es más complicado. En un árbol B^+ la entrada borrada siempre aparece en una hoja. En un árbol B, la entrada borrada podría aparecer en un nodo interno. El valor apropiado a colocar en su lugar se debe elegir del subárbol del nodo que contiene la entrada borrada. Concretamente, si se borra la clave de búsqueda K_i, la clave de búsqueda más pequeña que aparezca en el subárbol del puntero P_{i+1} se debe trasladar al campo ocupado anteriormente por K_i. Será necesario tomar otras medidas si ahora el nodo hoja tuviera pocas entradas. Por el contrario, la inserción en un árbol B es sólo un poco más complicada que la inserción en un árbol B^+.

Las ventajas de espacio que tienen los árboles B son escasas para índices grandes y normalmente no son de mayor importancia los inconvenientes advertidos. De esta manera, muchos implementadores de sistemas de bases de datos aprovechan la sencillez estructural de un árbol B^+. Los detalles de los algoritmos de inserción y borrado para árboles B se estudian en los ejercicios.

12.5 Accesos bajo varias claves

Hasta ahora se ha asumido implícitamente que se utiliza solamente un índice (o tabla asociativa) para procesar una consulta en una relación. Sin embargo, para ciertos tipos de consultas es ventajoso el uso de varios índices si existen o usar un índice construido sobre una clave de búsqueda de varios atributos.

12.5.1 Uso de varios índices de clave única

Considérese que el archivo *cuenta* tiene dos índices: uno para el *nombre_sucursal* y otro para *saldo*. Dada la consulta: "Encontrar el número de todas las cuentas de la sucursal de Pamplona con saldo igual a 1.000 €", se escribe

select *número_préstamo*
from *cuenta*
where *nombre_sucursal* = "Pamplona" **and** *saldo* = 1000

Hay tres estrategias posibles para procesar esta consulta:

1. Usar el índice de *nombre_sucursal* para encontrar todos los registros correspondientes a la sucursal de Pamplona. Examinar luego esos registros para ver si *saldo* = 1.000.
2. Usar el índice de *saldo* para encontrar todos los registros pertenecientes a cuentas con un saldo de 1.000 €. Examinar luego esos registros para ver si *nombre_sucursal* = "Pamplona".
3. Usar el índice de *nombre_sucursal* para encontrar *punteros* a registros correspondientes a la sucursal de Pamplona. Usar también el índice de *saldo* para encontrar los punteros a todos los registros correspondientes a cuentas con un saldo de 1.000 €. Realizar la intersección de esos dos conjuntos de punteros. Los punteros que están en la intersección apuntan a los registros correspondientes a la vez a Pamplona y a las cuentas con un saldo de 1.000 €.

La tercera estrategia es la única de las tres que aprovecha la ventaja de tener varios índices. Sin embargo, incluso esta estrategia podría ser una mala elección si hubiese:

- Muchos registros correspondientes a la sucursal Pamplona.
- Muchos registros correspondientes a cuentas con un saldo de 1.000 €.
- Sólo unos cuantos registros pertenecientes *tanto a* la sucursal de Pamplona como a las cuentas con un saldo de 1.000 €.

Si ocurrieran estas condiciones se tendrían que examinar un gran número de punteros para producir un resultado pequeño. La estructura de índices denominada "índice de mapas de bits" acelera significativamente la operación de inserción usada en la tercera estrategia. Los índices de mapas de bits se describen en el Apartado 12.9.

12.5.2 Índices sobre varias claves

Una estrategia más eficiente para este caso es crear y utilizar un índice con una clave de búsqueda (*nombre_sucursal*, *saldo*)—esto es, la clave de búsqueda consistente en el nombre de la sucursal concatenado con el saldo de la cuenta. Esa clave de búsqueda, que contiene más de un atributo, se denomina a veces **clave de búsqueda compuesta**. La estructura del índice es la misma que para cualquier otro índice, con la única diferencia de que la clave de búsqueda no es un simple atributo sino una lista de atributos. La clave de búsqueda se puede representar como una tupla de valores, de la forma $(a_1, \ldots, a_n)$, donde los atributos indexados son $A_1, \ldots, A_n$. El orden de los valores de la clave de búsqueda es el *orden lexicográfico*. Por ejemplo, para el caso de dos atributos en la clave de búsqueda, $(a_1, a_2) < (b_1, b_2)$ si $a_1 < b_1$ o bien $a_1 = b_1$ y $a_2 < b_2$. El orden lexicográfico es básicamente el mismo que el alfabético.

Se puede usar un índice ordenado (árbol B^+) para responder eficientemente consultas de la forma:

select *número_préstamo*
from *cuenta*
where *nombre_sucursal* = 'Pamplona' **and** *saldo* = 1000

Las consultas como la siguiente, que especifica una condición de igualdad sobre el primer atributo de la clave de búsqueda (*nombre_sucursal*) y un rango sobre el segundo (*saldo*) se pueden también tratar eficientemente ya que corresponden a una consulta por rangos sobre el atributo de búsqueda.

select *número_préstamo*
from *cuenta*
where *nombre_sucursal* = 'Pamplona' **and** *saldo* < 1000

Incluso se puede utilizar un índice ordenado sobre la clave de búsqueda (*nombre_sucursal*, *saldo*) para responder de manera eficiente a la siguiente consulta sobre un solo atributo:

select *número_préstamo*
from *cuenta*
where *nombre_sucursal* = 'Pamplona'

La condición de igualdad *nombre_sucursal* = "Pamplona" es equivalente a una consulta por rangos sobre el rango con extremo inferior (Pamplona, $-\infty$) y superior (Pamplona, $+\infty$). Las consultas por rangos sobre sólo el atributo *nombre_sucursal* se pueden tratar de forma similar.

Sin embargo, el uso de una estructura de índice ordenado con múltiples atributos presenta algunas deficiencias. Como ilustración considérese la consulta:

select *número_préstamo*
from *cuenta*
where *nombre_sucursal* $<$ "Pamplona" **and** *saldo* $=$ 1000

Se puede responder a esta consulta usando un índice ordenado con la clave de búsqueda (*nombre_sucursal*, *saldo*) de la manera siguiente: para cada valor de *nombre_sucursal* que está alfabéticamente por delante de "Pamplona", hay que localizar los registros con un *saldo* de 1.000. Sin embargo, debido a la ordenación de los registros en el archivo, es probable que cada registro esté en un bloque de disco diferente, lo que genera muchas operaciones de E/S.

La diferencia entre esta consulta y la anterior es que la condición sobre *nombre_sucursal* es de comparación y no de igualdad. La condición no se corresponde con una consulta por rangos sobre la clave de búsqueda.

Para acelerar el procesamiento en general de consultas con claves de búsqueda compuestas (que pueden implicar una o más operaciones de comparación) se pueden emplear varias estructuras especiales. En el Apartado 12.9.3 se considerará la estructura de *índices de mapas de bits*. Existe otra estructura, denominada *árbol R*, que también se puede usar para este propósito. Los árboles R son una extensión de los árboles B^+ para el tratamiento de índices en varias dimensiones. Dado que se emplean fundamentalmente con datos de tipo geográfico, su estructura se describe en el Capítulo 23.

12.5.3 Claves de búsqueda duplicadas

La creación de cajones de punteros para el tratamiento de las claves de búsqueda duplicadas (es decir, claves sobre archivos que puedan tener más de un registro con el mismo valor de la clave) plantea varias complicaciones cuando se implementan árboles B^+. Si los cajones se guardan en un nodo hoja se necesita código extra para tratar cajones de tamaño variable y para tratar el caso en que crecen por encima del tamaño del nodo hoja. Si los cajones se almacenan en páginas separadas puede ser necesario una operación E/S adicional para acceder a los registros.

Una solución sencilla a este problema, empleada por la mayoría de los sistemas de bases de datos, es hacer que las claves de búsqueda sean únicas añadiendo un atributo único adicional a la clave. El valor de este atributo podría ser un identificador de registro (si el sistema de bases de datos les da soporte) o simplemente un número único para todos los registros con el mismo valor de la clave de búsqueda. Por ejemplo, si se tuviese un índice sobre el atributo *nombre_cliente* de la tabla *impositor*, las entradas correspondientes a un cliente en particular tendrían diferentes valores para el atributo adicional. Así se garantizaría que la clave de búsqueda extendida fuese única.

Las búsquedas con el atributo original de la clave de búsqueda se convierten en búsquedas por rangos en la clave de búsqueda extendida, como se vio en el Apartado 12.5.2; al realizar la búsqueda, se ignora el valor del atributo adicional.

12.5.4 Índices de cobertura

Los **índices de cobertura** almacenan los valores de algunos atributos (distintos de los atributos de la clave de búsqueda) junto con los punteros a los registros. El almacenamiento de valores de atributo adicionales es útil en los índices secundarios, ya que permite responder algunas consultas utilizando únicamente el índice, sin ni siquiera buscar en los registros de datos.

Por ejemplo, supóngase un índice sin agrupación sobre el atributo *número_cuenta* de la relación *cuenta*. Si se almacena el valor del atributo *saldo* junto con el puntero a registro, se pueden responder consultas que soliciten el saldo (sin el otro atributo, *nombre_sucursal*) sin acceder al registro de *cuenta*.

Se obtendría el mismo efecto creando un índice sobre la clave de búsqueda (*número_cuenta*, *saldo*), pero los índices de cobertura reducen el tamaño de las claves de búsqueda, lo que permite un mayor grado de salida en los nodos internos y puede reducir la altura del índice.

12.5.5 Índices secundarios y reubicación de registros

Algunas organizaciones de archivos, como los árboles B^+, pueden cambiar la ubicación de los registros aunque éstos no se hayan modificado. A modo de ejemplo, considérese que cuando en una organización de archivo de árbol B^+ se divide una página hoja, varios registros se trasladan a una nueva página. En esos casos hay que actualizar todos los índices secundarios que almacenan punteros a los registros reubicados, aunque sus contenidos no hayan cambiado. Cada página hoja puede contener gran número de registros, y cada uno de ellos puede estar en diferentes ubicaciones de cada índice secundario. Así, la división de una página hoja puede exigir decenas o incluso centenas de operaciones de E/S para actualizar todos los índices secundarios afectados, lo que puede convertirla en una operación muy costosa.

A continuación se explica una técnica para resolver este problema. En los índices secundarios, en lugar de punteros a los registros indexados, se almacenan los valores de los atributos de la clave de búsqueda del índice primario. Por ejemplo, supóngase que se tiene un índice primario sobre el atributo *número_cuenta* de la relación *cuenta*; un índice secundario sobre *nombre_sucursal* almacenaría con cada nombre de sucursal una lista de valores de *número_cuenta* de los registros correspondientes, en lugar de almacenar punteros a los registros.

Por tanto, la reubicación de los registros debido a la división de las páginas hoja no exige ninguna actualización de los índices secundarios. Sin embargo, la ubicación de un registro mediante el índice secundario exige ahora dos pasos: primero se utiliza el índice secundario para buscar los valores de la clave de búsqueda del índice primario y luego se utiliza el índice primario para buscar los registros correspondientes.

Este enfoque reduce en gran medida el coste de actualización de los índices debido a la reorganización de los archivos, aunque incrementa el coste de acceso a los datos mediante un índice secundario.

12.6 Asociación estática

Un inconveniente de la organización de archivos secuenciales es que hay que acceder a una estructura de índices para localizar los datos o utilizar una búsqueda binaria y, como resultado, más operaciones de E/S. La organización de archivos basada en la técnica de **asociación** (hashing) permite evitar el acceso a la estructura de índice. La asociación también proporciona una forma de construir índices. En los apartados siguientes se estudian las organizaciones de archivos y los índices basados en asociación.

En esta descripción de la asociación, se usará el término **cajón** (bucket) para indicar una unidad de almacenamiento que puede guardar uno o más registros. Un cajón es normalmente un bloque de disco, aunque también se podría elegir de tamaño mayor o menor que un bloque de disco.

Formalmente, sea K el conjunto de todos los valores de clave de búsqueda y sea B el conjunto de todas las direcciones de cajón. Una **función de asociación** h es una función de K a B. Sea h una función asociación.

Para insertar un registro con clave de búsqueda K_i, se calcula $h(K_i)$, lo que proporciona la dirección del cajón para ese registro. De momento se supone que hay espacio en el cajón para almacenar el registro. Luego, el registro se almacena en ese cajón.

Para realizar una búsqueda con el valor K_i de la clave de búsqueda, basta con calcular $h(K_i)$ y buscar luego el cajón con esa dirección. Supóngase que dos claves de búsqueda, K_5 y K_7, tienen el mismo valor de asociación; es decir, $h(K_5) = h(K_7)$. Si se realiza una búsqueda en K_5, el cajón $h(K_5)$ contendrá registros con valores de la clave de búsqueda K_5 y registros con valores de la clave de búsqueda K_7. Por tanto, hay que comprobar el valor de clave de búsqueda de todos los registros del cajón para asegurarse de que el registro es el deseado.

El borrado es igual de sencillo. Si el valor de la clave de búsqueda del registro que se debe borrar es K_i, se calcula $h(K_i)$, después se busca el cajón correspondiente a ese registro y se borra el registro del cajón.

La asociación se puede usar para dos propósitos diferentes. En la **organización de archivo asociativo** se obtiene directamente la dirección del bloque de disco que contiene el registro deseado calculando una función del valor de la clave de búsqueda del registro. En la **organización de índice asociativo** se organizan las claves de búsqueda con sus punteros asociados en una estructura de archivo asociativo.

12.6.1 Funciones de asociación

La peor función de asociación posible asigna todos los valores de la clave de búsqueda al mismo cajón. Una función así no es deseable, ya que hay que guardar todos los registros en el mismo cajón. Durante una búsqueda hay que examinar todos esos registros hasta encontrar el deseado. Una función de asociación ideal distribuye las claves almacenadas uniformemente entre todos los cajones, de modo que cada uno de ellos tenga el mismo número de registros.

Puesto que durante la etapa de diseño no se conocen con precisión los valores de la clave de búsqueda que se almacenarán en el archivo, se pretende elegir una función de asociación que asigne los valores de la clave de búsqueda a los cajones de manera que la distribución tenga las propiedades siguientes:

- Distribución *uniforme*. Esto es, la función de asociación asigna a cada cajón el mismo número de valores de la clave de búsqueda dentro del conjunto de *todos* los valores posibles de la misma.
- Distribución *aleatoria*. Esto es, en el caso promedio, cada cajón tendrá asignado casi el mismo número de valores, independientemente de la distribución real de los valores de la clave de búsqueda. Para ser más exactos, el valor de asociación no estará correlacionado con ningún ordenamiento de los valores de la clave de búsqueda visible exteriormente como, por ejemplo, el orden alfabético o el orden determinado por la longitud de las claves de búsqueda; parecerá que la función de asociación es aleatoria.

Como ilustración de estos principios, se escogerá una función de asociación para el archivo *cuenta* que utilice la clave búsqueda *nombre_sucursal*. La función de asociación que se escoja debe tener las propiedades deseadas no sólo para el ejemplo del archivo *cuenta* que se ha estado utilizando, sino también para el archivo *cuenta* de tamaño real de un gran banco con muchas sucursales.

Supóngase que se decide tener 26 cajones y se define una función de asociación que asigna a los nombres que empiezan con la letra *i*-ésima del alfabeto el cajón *i*-ésimo. Esta función de asociación tiene la virtud de la simplicidad, pero no logra proporcionar una distribución uniforme, ya que se espera que haya más nombres de sucursales que comiencen con letras como B y R que con Q y con X, por ejemplo.

Supóngase ahora que se desea una función de asociación en la clave de búsqueda *saldo*. Supóngase que el saldo mínimo es 1, el saldo máximo es 100.000 y se utiliza una función de asociación que divide el valor en 10 rangos, 1—10.000, 10.001—20.000, y así sucesivamente. La distribución de los valores de la clave de búsqueda es uniforme (ya que cada cajón tiene el mismo número de valores del *saldo* diferentes), pero no es aleatoria. Los registros con saldos entre 1 y 10.000 son más frecuentes que los registros con saldos entre 90.001 y 100.000. En consecuencia, la distribución de los registros no es uniforme—algunos cajones reciben más registros que otros. Si la función presenta una distribución aleatoria, aunque se dieran esas correlaciones entre las claves de búsqueda, la aleatoriedad de la distribución haría, muy probablemente, que todos los cajones tuvieran más o menos el mismo número de registros, siempre y cuando cada clave de búsqueda apareciera sólo en una pequeña parte de los registros (si una sola clave de búsqueda aparece en gran parte de registros, es probable que el cajón que la contiene contenga más registros que otros cajones, independientemente de la función de asociación empleada).

Las funciones de asociación habituales realizan cálculos sobre la representación binaria interna de la máquina de los caracteres de la clave de búsqueda. Una función de asociación sencilla de este tipo calcula en primer lugar la suma de las representaciones binarias de los caracteres de la clave y, luego, devuelve el resto de la división entre la suma y el número de cajones. En la Figura 12.21 se muestra la aplicación de este esquema, con 10 cajones, al archivo *cuenta*, con la suposición de que la letra *i*-ésima del alfabeto está representada por el número entero *i*.

La siguiente función de asociación se puede usar para asociar una cadena en una implementación Java.

$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \cdots + s[n-1]$$

La función se puede implementar eficientemente estableciendo el valor asociativo inicialmente como 0 e iterando desde el primero hasta el último carácter de la cadena, multiplicando en cada paso el valor asociativo por 31 y añadiendo el siguiente carácter (tratado como un entero). El resto de la división entre el resultado de esta función y el número de cajones se puede usar para la indexación.

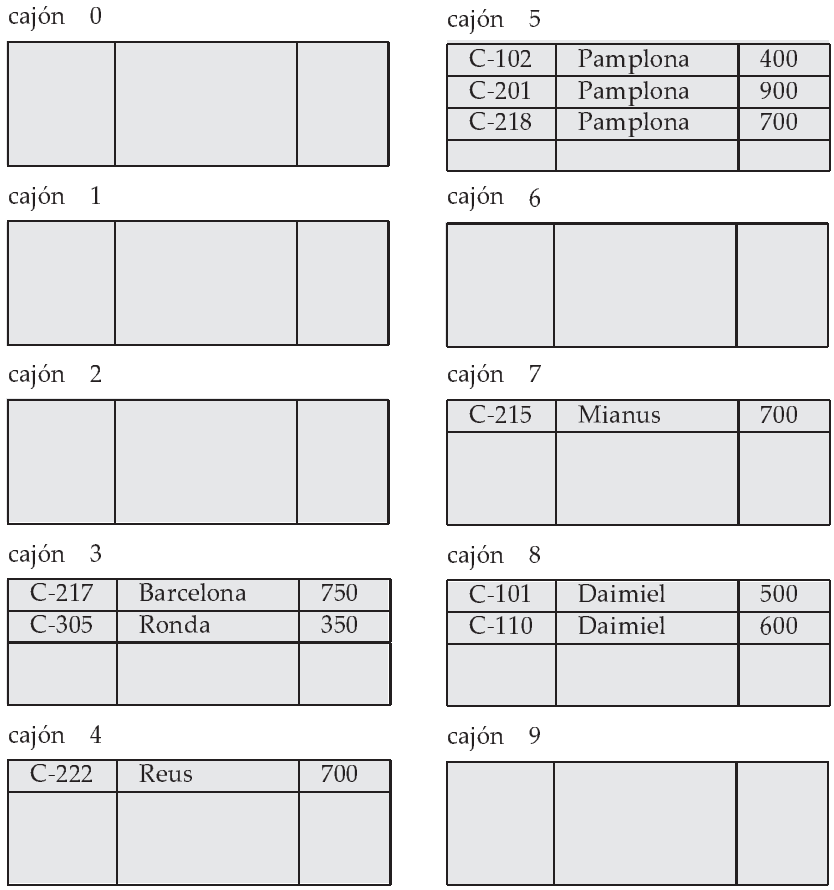


Figura 12.21 Organización asociativa del archivo *cuenta* utilizando *nombre_sucursal* como clave.

Las funciones de asociación requieren un diseño cuidadoso. Una mala función de asociación puede hacer que la búsqueda tarde un tiempo proporcional al número de claves de búsqueda del archivo. Una función bien diseñada da un tiempo de búsqueda para casos promedio que es una constante (pequeña), independiente del número de claves de búsqueda del archivo.

12.6.2 Gestión de desbordamientos de cajones

Hasta ahora se ha asumido que cuando se inserta un registro, el cajón al que se asigna tiene espacio para almacenarlo. Si el cajón no tiene suficiente espacio, sucede lo que se denomina **desbordamiento de cajones**. Los desbordamientos de cajones se pueden producir por varias razones:

- **Cajones insuficientes**. Se debe elegir un número de cajones, que se denota con n_B, tal que $n_B > n_r/f_r$, donde n_r denota el número total de registros que se van a almacenar y f_r denota el número de registros que caben en cada cajón. Esta designación, por supuesto, supone que se conoce el número total de registros en el momento de definir la función de asociación.
- **Atasco**. Algunos cajones tienen asignados más registros que otros, por lo que algún cajón se puede desbordar, aunque otros cajones tengan todavía espacio libre. Esta situación se denomina **atasco** de cajones. El atasco puede producirse por dos motivos:
  1. Puede que varios registros tengan la misma clave de búsqueda.
  2. Puede que la función de asociación elegida genere una distribución no uniforme de las claves de búsqueda.

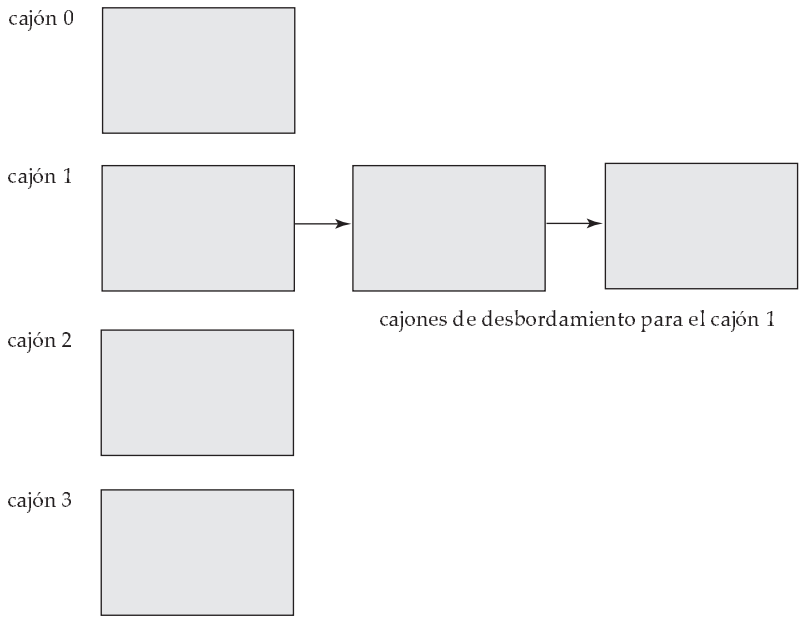


Figura 12.22 Cadena de desbordamiento en una estructura asociativa.

Para que la probabilidad de desbordamiento de cajones se reduzca, se escoge un número de cajones igual a $(n_r/f_r) * (1 + d)$, donde d es un factor de corrección, normalmente cercano a 0.2. Se pierde algo de espacio: alrededor del veinte por ciento del espacio en los cajones queda vacío. Pero la ventaja es que la probabilidad de desbordamiento se reduce.

Pese a la asignación de unos pocos cajones más de los necesarios, todavía se puede producir el desbordamiento de los cajones. El desbordamiento de los cajones se trata mediante **cajones de desbordamiento**. Si hay que insertar un registro en un cajón c y c está ya lleno, el sistema proporcionará un cajón de desbordamiento para c y el registro se insertará en ese cajón de desbordamiento. Si el cajón de desbordamiento también se encuentra lleno, el sistema proporcionará otro cajón de desbordamiento, y así sucesivamente. Todos los cajones de desbordamiento de un cajón determinado se encadenan en una lista enlazada, como se muestra en la Figura 12.22. El tratamiento del desbordamiento mediante una lista enlazada, se denomina **cadena de desbordamiento**.

Para tratar la cadena de desbordamiento es necesario modificar ligeramente el algoritmo de búsqueda. Como se dijo antes, el sistema utiliza la función de asociación sobre la clave de búsqueda para identificar un cajón c. Luego debe examinar todos los registros del cajón c para ver si coinciden con la clave de búsqueda, como antes. Además, si el cajón c tiene cajones de desbordamiento, también hay que examinar los registros de todos los cajones de desbordamiento.

La forma de la estructura asociativa que se acaba de describir se denomina a veces **asociación cerrada**. En una aproximación alternativa, conocida como **asociación abierta**, se fija el conjunto de cajones y no hay cadenas de desbordamiento. En su lugar, si un cajón está lleno, el sistema inserta los registros en algún otro cajón del conjunto inicial C de cajones. Un criterio es utilizar el siguiente cajón (en orden cíclico) que tenga espacio; esta política se llama *ensayo lineal*. Se utilizan también otros criterios, como el cálculo de funciones de asociación adicionales. La asociación abierta se emplea en la construcción de tablas de símbolos para compiladores y ensambladores, pero la asociación cerrada es preferible para los sistemas de bases de datos. El motivo es que el borrado con asociación abierta es problemático. Normalmente, los compiladores y los ensambladores sólo realizan operaciones de búsqueda e inserción en sus tablas de símbolos. Sin embargo, en los sistemas de bases de datos es importante poder tratar tanto el borrado como la inserción. Por tanto, la asociatividad abierta sólo tiene una importancia menor en la implementación de bases de datos.

Un inconveniente importante de la forma de asociación que se ha descrito, es que hay que elegir la función de asociación cuando se implementa el sistema y no se puede cambiar fácilmente después si el archivo que se está indexando aumenta o disminuye de tamaño. Como la función h asigna valores

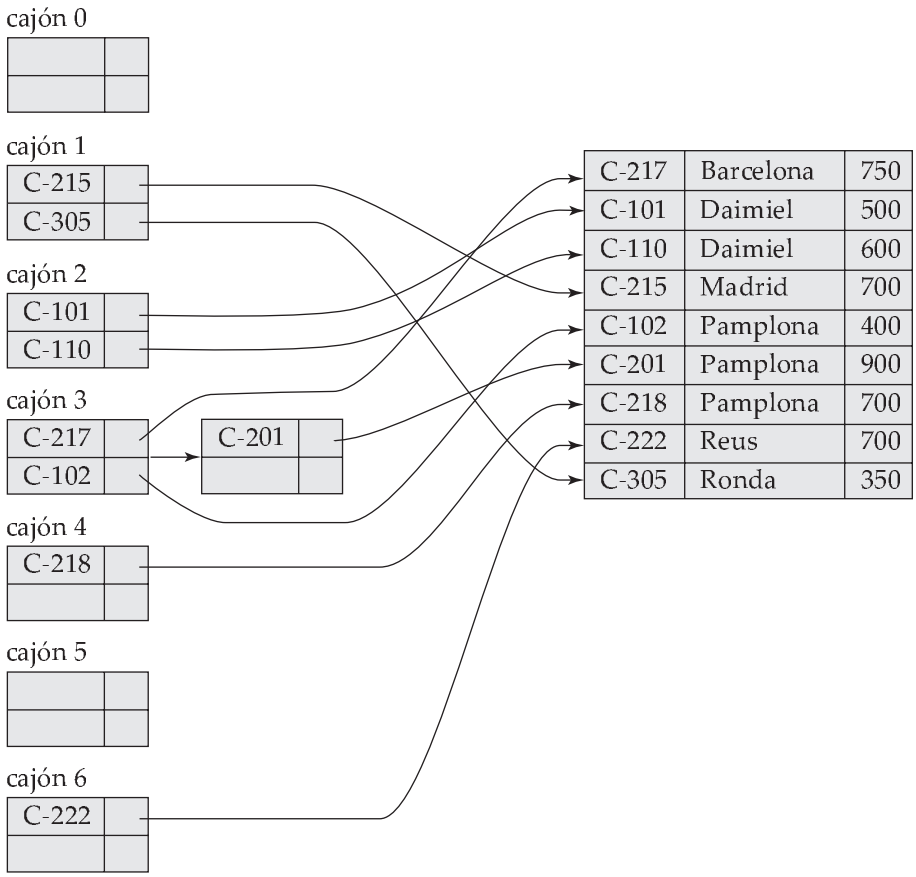


Figura 12.23 Índice asociativo de la clave de búsqueda *número_cuenta* del archivo *cuenta*.

de la clave búsqueda a un conjunto fijo *C* de direcciones de cajón, se desperdicia espacio si se hace *C* de gran tamaño para manejar el futuro crecimiento del archivo. Si *C* es demasiado pequeño, cada cajón contendrá registros de muchos valores de la clave de búsqueda y se pueden provocar desbordamientos de cajones. A medida que el archivo aumenta de tamaño, el rendimiento se degrada. Más adelante, en el Apartado 12.7, se estudiará la manera de cambiar dinámicamente el número de cajones y la función de asociación.

12.6.3 Índices asociativos

La asociatividad se puede utilizar no solamente para la organización de archivos, sino también para la creación de estructuras de índice. Cada **índice asociativo** (hash index) organiza las claves de búsqueda, con sus punteros asociados, en una estructura de archivo asociativo. Los índices asociativos se construyen como se indica a continuación. Primero se aplica una función de asociación sobre la clave de búsqueda para identificar un cajón, luego se almacenan la clave y los punteros asociados en el cajón (o en los cajones de desbordamiento). En la Figura 12.23 se muestra un índice asociativo secundario del archivo *cuenta* para la clave de búsqueda *número_cuenta*. La función de asociación utilizada calcula la suma de las cifras del número de cuenta módulo siete. El índice asociativo posee siete cajones, cada uno de tamaño dos (un índice realista tendría, por supuesto, cajones de mayor tamaño). Uno de los cajones tiene tres claves asignadas, por lo que tiene un cajón de desbordamiento. En este ejemplo, *número_cuenta* es clave primaria de *cuenta*, por lo que cada clave de búsqueda sólo tiene asociado un puntero. En general, se pueden asociar varios punteros con cada clave.

Se usa el término *índice asociativo* para denotar las estructuras de archivo asociativo, así como los índices secundarios asociativos. Estrictamente hablando, los índices asociativos son sólo estructuras de índices secundarios. Los índices asociativos no se necesitan nunca como estructuras de índices con agru-

pación ya que, si un archivo está organizado mediante asociatividad, no hay necesidad de ninguna estructura de índice asociativo adicional. Sin embargo, como la organización en archivos asociativos proporciona el mismo acceso directo a los registros que el indexado, se finge que la organización de un archivo mediante asociación también tiene en él un índice con agrupación asociativo.

12.7 Asociación dinámica

Como se ha visto, la necesidad de fijar el conjunto *C* de direcciones de los cajones presenta un problema serio con la técnica de asociación estática vista en el apartado anterior. La mayor parte de las bases de datos aumenta de tamaño con el tiempo. Si se va a utilizar la asociación estática para esas bases de datos, hay tres opciones:

1. Elegir una función de asociación de acuerdo con el tamaño actual del archivo. Esta opción provocará una degradación del rendimiento a medida que la base de datos aumente de tamaño.
2. Elegir una función de asociación de acuerdo con el tamaño previsto del archivo en un momento determinado del futuro. Aunque se evite la degradación del rendimiento, puede que inicialmente se pierda una cantidad de espacio significativa.
3. Reorganizar periódicamente la estructura asociativa en respuesta al crecimiento del archivo. Esta reorganización supone elegir una nueva función de asociación, volver a calcular la función de asociación de cada registro del archivo y generar nuevas asignaciones de cajones. Esta reorganización es una operación masiva que consume mucho tiempo. Además, hay que prohibir el acceso al archivo durante la reorganización.

Algunas técnicas de **asociación dinámica** permiten modificar dinámicamente la función de asociación para adaptarse al aumento o disminución de tamaño de la base de datos. En esta apartado se describe una forma de asociación dinámica, denominada **asociación extensible**. Las notas bibliográficas proporcionan referencias a otras formas de asociación dinámica.

12.7.1 Estructura de datos

La asociación extensible hace frente a los cambios del tamaño de la base de datos dividiendo y fusionando los cajones a medida que la base de datos aumenta o disminuye. En consecuencia, se conserva la eficiencia espacial. Además, puesto que la reorganización sólo se lleva a cabo en un cajón simultáneamente, la degradación del rendimiento resultante es aceptablemente baja.

Con la asociación extensible se elige una función de asociación h con las propiedades deseadas de uniformidad y aleatoriedad. Sin embargo, esa función de asociación genera valores dentro de un rango relativamente amplio—por ejemplo, los enteros binarios de b bits. Un valor normal de b es 32.

No se crea un cajón para cada valor de la función de asociación. De hecho, 2^{32} es más de cuatro mil millones, y no son razonables tantos cajones salvo para las mayores bases de datos. En vez de eso, se crean cajones bajo demanda, a medida que se insertan registros en el archivo. Inicialmente no se utilizan todos los b bits del valor de la función de asociación. En cualquier momento se utilizan i bits, donde $0 \leq i \leq b$. Esos i bits son utilizados como reserva en una tabla adicional de direcciones de los cajones. El valor de i aumenta o disminuye con el tamaño de la base de datos.

En la Figura 12.24 se muestra una estructura general de asociación extensible. La i que aparece en la figura encima de la tabla de direcciones de los cajones indica que se requieren i bits del valor de la función de asociación $h(K)$ para determinar el cajón apropiado para K. Obviamente, ese número cambia a medida que el archivo aumenta de tamaño. Aunque se requieren i bits para encontrar la entrada correcta en la tabla de direcciones de los cajones, varias entradas consecutivas de la tabla pueden apuntar al mismo cajón. Todas esas entradas tendrán un prefijo de asociación común, pero la longitud de ese prefijo puede ser menor que i. Por lo tanto, se asocia con cada cajón un número entero que proporciona la longitud del prefijo de asociación común. En la Figura 12.24, el entero asociado con el cajón j aparece como i_j. El número de entradas de la tabla de direcciones de cajones que apuntan al cajón j es

$$2^{(i - i_j)}$$

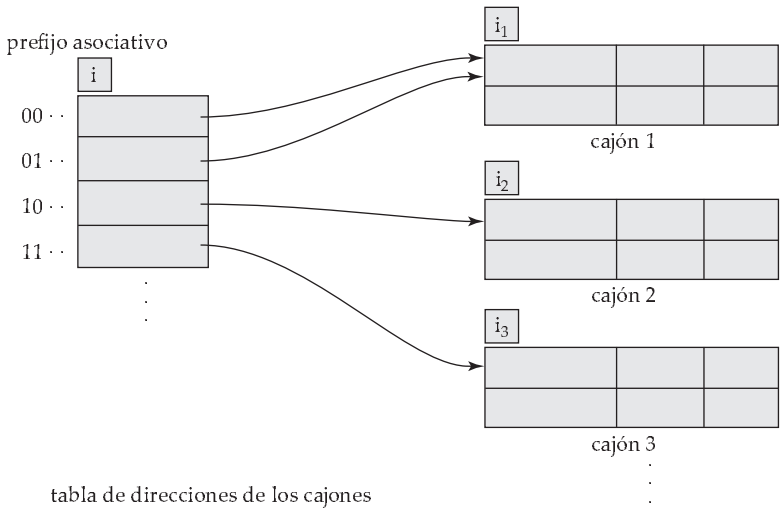


Figura 12.24 Estructura asociativa general extensible.

12.7.2 Consultas y actualizaciones

A continuación se estudiará la forma de realizar la búsqueda, la inserción y el borrado en una estructura asociativa extensible.

Para localizar el cajón que contiene el valor de la clave de búsqueda K_l, el sistema toma los primeros *i* bits más significativos de $h(K_l)$, busca la entrada de la tabla correspondiente a esa cadena de bits, y sigue el puntero del cajón de esa entrada de la tabla.

Para insertar un registro con un valor de la clave de búsqueda K_l se sigue el mismo procedimiento de búsqueda que antes, y se llega a un cajón—por ejemplo, *j*. Si hay sitio en ese cajón, se inserta en él el registro. Si, por el contrario, el cajón está lleno, hay que dividir el cajón y redistribuir los registros actuales, junto con el nuevo. Para dividir el cajón, primero hay que determinar, a partir del valor de la función de asociación, si hace falta incrementar el número de bits que hay que utilizar.

- Si $i = i_j$, entonces solamente apunta al cajón *j* una entrada de la tabla de direcciones de los cajones. Por tanto, es necesario incrementar el tamaño de la tabla de direcciones de los cajones para incluir los punteros a los dos cajones que resultan de la división del cajón *j*. Esto se consigue considerando otro bit más del valor de asociación. Se incrementa en uno el valor de *i*, lo que duplica el tamaño de la tabla de direcciones de los cajones. Cada entrada se sustituye por dos entradas, ambas con el mismo puntero que la entrada original. Ahora, dos entradas de la tabla de direcciones de cajones apuntan al cajón *j*. Se asigna un nuevo cajón (el cajón *z*) y se hace que la segunda entrada apunte al nuevo cajón. Se definen i_j e i_z como *i*. A continuación se vuelve a calcular la función de asociación para todos los registros del cajón *j* y, en función de los primeros *i* bits (recuérdese que se ha añadido uno a *i*), se mantienen en el cajón *j* o se colocan en el cajón recién creado.

  Ahora se vuelve a intentar la inserción del nuevo registro. Normalmente el intento tiene éxito. Sin embargo, si todos los registros del cajón *j*, así como el registro nuevo, tienen el mismo prefijo de asociación, será necesario volver a dividir el cajón, ya que tanto los registros del cajón *j* como el registro nuevo tienen asignado el mismo cajón. Si la función de asociación se eligió cuidadosamente, es poco probable que una simple inserción provoque que un cajón se divida más de una vez, a menos que haya un gran número de registros con la misma clave de búsqueda. Si todos los registros del cajón *j* tienen el mismo valor de la clave de búsqueda, la división no servirá de nada. En esos casos se utilizan cajones de desbordamiento para almacenar los registros, como en la asociación estática.

- Si $i > i_j$, entonces más de una entrada en la tabla de direcciones de los cajones apunta al cajón *j*. Por tanto, se puede dividir el cajón *j* sin aumentar el tamaño de la tabla de direcciones

C-217	Barcelona	750
C-101	Daimiel	500
C-110	Daimiel	600
C-215	Madrid	700
C-102	Pamplona	400
C-201	Pamplona	900
C-218	Pamplona	700
C-222	Reus	700
C-305	Ronda	350

Figura 12.25 Archivo *cuenta* de ejemplo.

nombre_sucursal	*h(nombre_sucursal)*
Barcelona	0010 1101 1111 1011 0010 1100 0011 0000
Daimiel	1010 0011 1010 0000 1100 0110 1001 1111
Madrid	1100 0111 1110 1101 1011 1111 0011 1010
Pamplona	1111 0001 0010 0100 1001 0011 0110 1101
Reus	0011 0101 1010 0110 1100 1001 1110 1011
Ronda	1101 1000 0011 1111 1001 1100 0000 0001

Figura 12.26 Función de asociación para *nombre_sucursal*.

de los cajones. Obsérvese que todas las entradas que apuntan al cajón *j* corresponden a prefijos de asociación que tienen el mismo valor en los i_j bits situados más a la izquierda. Se asigna un nuevo cajón (el cajón *z*) y se definen i_j e i_z con el valor que resulta de añadir uno al valor original de i_j. A continuación hay que ajustar las entradas de la tabla de direcciones de los cajones que anteriormente apuntaban al cajón *j* (obsérvese que, con el nuevo valor de i_j, no todas las entradas corresponden a prefijos de asociación que tienen el mismo valor en los i_j bits situados más a la izquierda). La primera mitad de las entradas se deja como estaba (apuntando al cajón *j*) y se hace que el resto de las entradas apunte al cajón recién creado (el cajón *z*). Luego, como en el caso anterior, se vuelve a calcular la función de asociación para todos los registros del cajón *j* y se asignan o bien al cajón *j* o bien al cajón *z* recién creado.

Luego se vuelve a intentar la inserción. En el improbable caso de que vuelva a fallar, se aplica uno de los dos casos, $i = i_j$ o $i > i_j$, según corresponda.

Obsérvese que en ambos casos solamente se necesita volver a calcular la función de asociación de los registros del cajón *j*.

Para borrar un registro con valor de la clave de búsqueda K_l se sigue el mismo procedimiento de búsqueda, que finaliza en un cajón—por ejemplo, *j*. Se borran tanto el registro del archivo como la clave de búsqueda del cajón. También se elimina el cajón si se queda vacío. Obsérvese que, en este momento, se pueden fusionar varios cajones y se puede reducir el tamaño de la tabla de direcciones de los cajones a la mitad. El procedimiento para decidir los cajones que se deben fusionar y el momento de hacerlo se deja al lector como ejercicio. Las condiciones bajo las que la tabla de direcciones de los cajones se puede reducir de tamaño también se dejan como ejercicio. A diferencia de la fusión de los cajones, el cambio de tamaño de la tabla de direcciones de los cajones es una operación bastante costosa si la tabla es de gran tamaño. Por tanto, sólo merece la pena reducir el tamaño de la tabla de direcciones de los cajones si el número de cajones se reduce considerablemente.

El ejemplo del archivo *cuenta* de la Figura 12.25 ilustra la operación de inserción. Los valores de asociación de 32 bits de *nombre_sucursal* se muestran en la Figura 12.26. Supóngase que, inicialmente, el archivo está vacío, como se muestra en la Figura 12.27. Los registros se insertan de uno en uno. Para mostrar todas las características de la asociación extensible en una estructura pequeña, se hará la suposición no realista de que cada cajón sólo puede contener dos registros.

Se va a insertar el registro (C-217, Barcelona, 750). La tabla de direcciones de los cajones contiene un puntero al único cajón existente y el sistema inserta el registro. A continuación se inserta el registro (C-101, Daimiel, 500). Este registro también se inserta en el único cajón de la estructura.

Cuando se intenta insertar el siguiente registro (C-110, Daimiel, 600), el cajón está lleno. Ya que $i = i_0$, es necesario incrementar el número de bits del valor de asociación que se utilizan. Ahora se utiliza un bit, lo que permite $2^1 = 2$ cajones. Este incremento en el número de bits necesarios necesita doblar el tamaño de la tabla de direcciones de cajones a dos entradas. El sistema divide el cajón, coloca en el nuevo aquellos registros cuya clave de búsqueda tiene un valor de asociación que comienza por 1 y deja el resto de los registros en el cajón original. En la Figura 12.28 se muestra el estado de la estructura después de la división.

A continuación se inserta (C-215, Madrid, 700). Como el primer bit de *h*(Madrid) es 1, hay que insertar ese registro en el cajón al que apunta la entrada "1" de la tabla de direcciones de los cajones. Una vez más, el cajón se encuentra lleno e $i = i_1$. Se incrementa a dos el número de bits del valor de asociación que se usan. Este incremento en el número de bits necesarios hace que se doble el tamaño de la tabla de direcciones de los cajones a cuatro entradas, como se muestra en la Figura 12.29. Como el cajón de la Figura 12.28 con el prefijo 0 del valor de asociación no se dividió, las dos entradas, 00 y 01, de la tabla de direcciones de los cajones apuntan a ese cajón.

Para cada registro del cajón de la Figura 12.28 con prefijo de asociación 1 (el cajón que se va a dividir) se examinan los dos primeros bits del valor de asociación para determinar el cajón de la nueva estructura que le corresponde.

A continuación se inserta el registro (C-102, Pamplona, 400), que se aloja en el mismo cajón que Madrid. La siguiente inserción, la de (C-201, Pamplona, 900), provoca un desbordamiento en un cajón, lo que provoca el incremento del número de bits y la duplicación del tamaño de la tabla de direcciones de los cajones. La inserción del tercer registro de Pamplona, (C-218, Pamplona, 700), produce otro desbordamiento. Sin embargo, este desbordamiento no se puede resolver incrementando el número de bits, ya que hay tres registros con el mismo valor de asociación exactamente. Por tanto se utiliza un cajón de desbordamiento, como se muestra en la Figura 12.30.

Se continúa de esta manera hasta que se hayan insertado todos los registros del archivo *cuenta* de la Figura 12.25. La estructura resultante se muestra en la Figura 12.31.

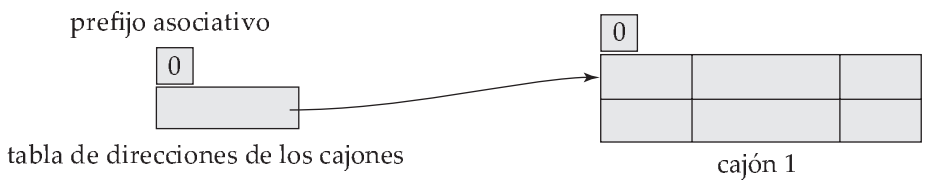


Figura 12.27 Estructura asociativa extensible inicial.

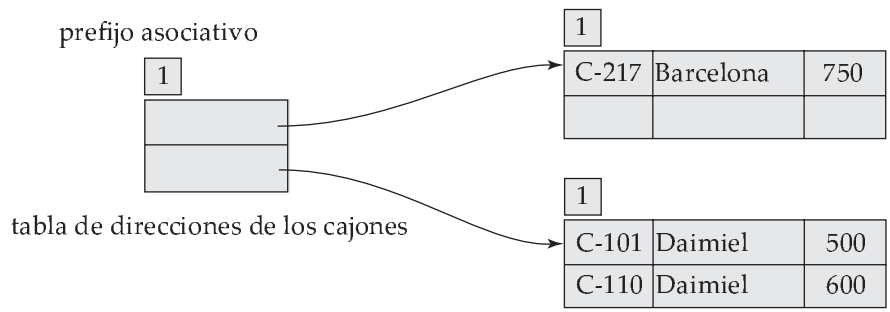


Figura 12.28 Estructura asociativa después de tres inserciones.

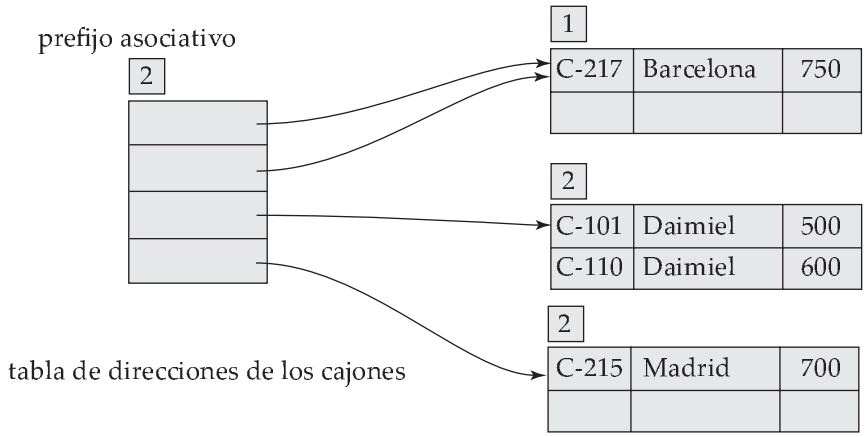


Figura 12.29 Estructura asociativa después de cuatro inserciones.

12.7.3 Comparación entre la asociación estática y la dinámica

A continuación se examinan las ventajas e inconvenientes de la asociación extensible respecto de la asociación dinámica. La ventaja principal de la asociación extensible es que el rendimiento no se degrada a medida que el archivo aumenta de tamaño. Además, el espacio adicional necesario es mínimo. Aunque la tabla de direcciones de los cajones provoca un gasto adicional, sólo contiene un puntero por cada valor de asociación para la longitud actual del prefijo. Por tanto, el tamaño de la tabla es pequeño. El principal ahorro de espacio de la asociación extensible respecto de otras formas de asociación es que no es necesario reservar cajones para un futuro aumento de tamaño; en vez de eso, los cajones se pueden asignar de manera dinámica.

Un inconveniente de la asociación extensible es que la búsqueda implica un nivel adicional de referencia, ya que se debe acceder a la tabla de direcciones de los cajones antes que a los propios cajones. Esta referencia adicional sólo tiene una mínima repercusión en el rendimiento. Aunque las estructuras asociativas que se estudiaron en el Apartado 12.6 no tienen este nivel adicional de referencia, pierden esa mínima ventaja de rendimiento cuando se llenan.

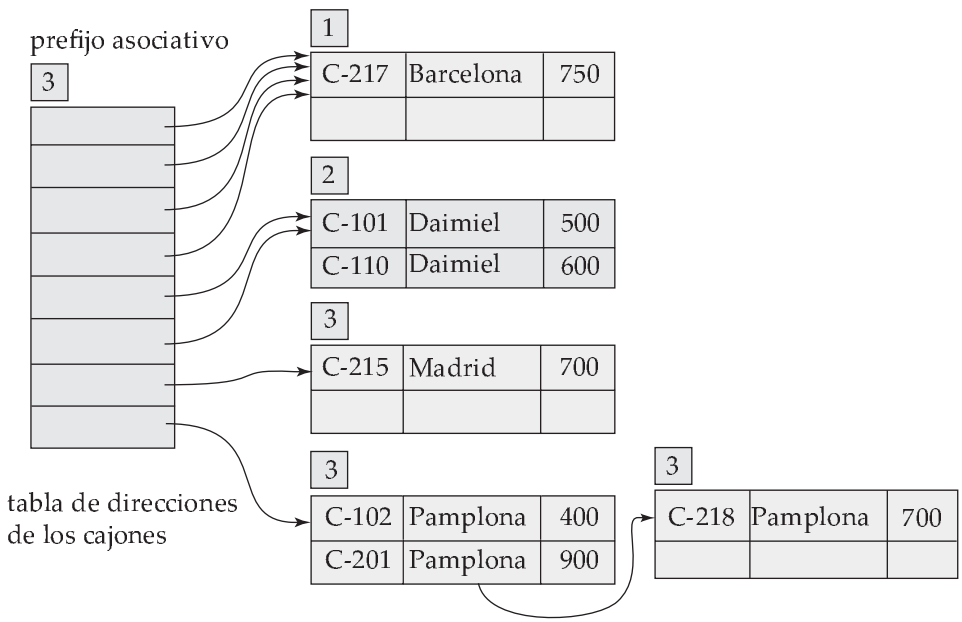


Figura 12.30 Estructura asociativa después de siete inserciones.

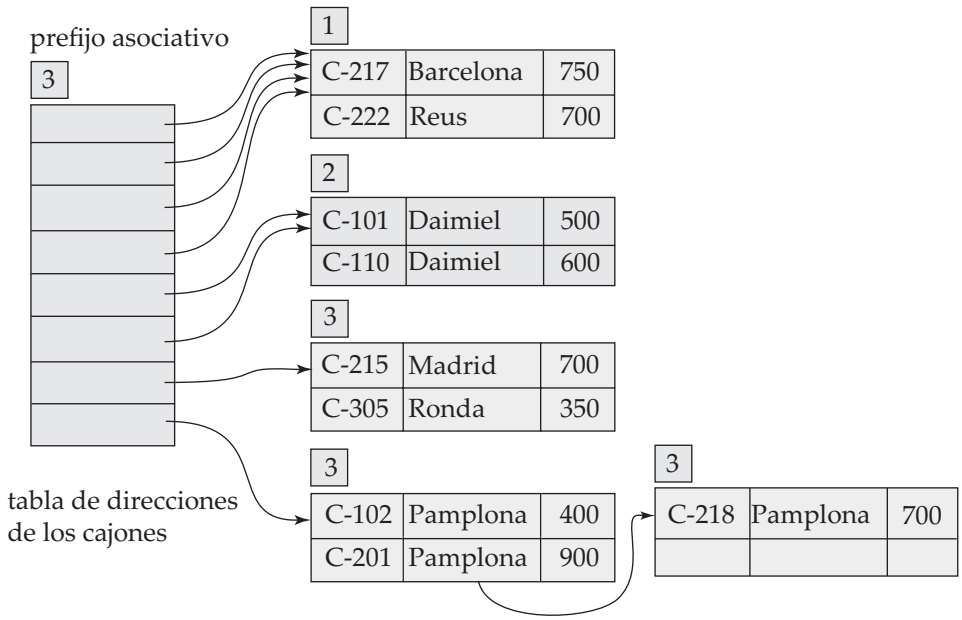


Figura 12.31 Estructura asociativa extensible para el archivo *cuenta*.

Por tanto, la asociación extensible se muestra como una técnica muy atractiva, siempre que se acepte la complejidad añadida de su implementación. En las notas bibliográficas se proporcionan descripciones más detalladas de la implementación de la asociación extensible.

Las notas bibliográficas también ofrecen referencias a otra forma de asociación dinámica denominada **asociación lineal**, que evita el nivel adicional de referencia asociado con la asociación extensible al posible coste de más cajones de desbordamiento.

12.8 Comparación de la indexación ordenada y la asociación

Se han examinado varios esquemas de indexación ordenada y varios esquemas de asociación. Se pueden organizar los archivos de registros como archivos ordenados, mediante una organización de índice secuencial u organizaciones de árbol B^+. Alternativamente, se pueden organizar los archivos mediante la asociación. Finalmente, los archivos se pueden organizar como montículos, en los que los registros no están ordenados de ninguna manera en especial.

Cada esquema tiene sus ventajas, dependiendo de la situación. Un fabricante de un sistema de bases de datos puede proporcionar muchos esquemas y dejar al diseñador de la base de datos la decisión final sobre los esquemas que se utilizarán. Sin embargo, este enfoque exige que el fabricante escriba más código, lo que aumenta tanto el coste del sistema como el espacio que ocupa. La mayor parte de los sistemas de bases de datos soportan árboles B^+ y pueden dar soporte también a alguna forma de organización de archivos asociativos o de índices asociativos.

Para hacer una buena elección de organización de archivos y de técnica de indexado, el fabricante o el diseñador de la base de datos debe tener en consideración los siguientes aspectos:

- ¿Es aceptable el coste de una reorganización periódica del índice o de la estructura asociativa?
- ¿Cuál es la frecuencia relativa de las inserciones y de los borrados?
- ¿Es deseable optimizar el tiempo medio de acceso a expensas de incrementar el peor tiempo de acceso posible?
- ¿Qué tipos de consultas se supone que van a formular los usuarios?

De estos puntos ya se han examinado los tres primeros, comenzando con la revisión de las ventajas relativas de las distintas técnicas de indexado, y nuevamente en la discusión de las técnicas de asociación.

El cuarto punto, el tipo de consultas esperado, resulta fundamental para la elección entre la indexación ordenada y la asociación.

Si la mayoría de las consultas son de la forma:

select $A_1, A_2, \ldots, A_n$
from *r*
where $A_i = c$

entonces, para procesarla, el sistema realiza una búsqueda en un índice ordenado o en una estructura asociativa del atributo A_i con el valor *c*. Para este tipo de consultas es preferible un esquema asociativo. Las búsquedas en índices ordenados requieren un tiempo proporcional al logaritmo del número de valores de A_i en *r*. Sin embargo, en las estructuras asociativas, el tiempo medio de búsqueda es una constante independiente del tamaño de la base de datos. La única ventaja de los índices respecto de las estructuras asociativas en este tipo de consultas es que el tiempo de búsqueda en el peor de los casos es proporcional al logaritmo del número de valores de A_i en *r*. Por el contrario, si se utiliza una estructura asociativa, el tiempo de búsqueda en el peor de los casos es proporcional al número de valores de A_i en *r*. Sin embargo, es poco probable en el caso de la asociación que se dé el peor caso de búsqueda posible (máximo tiempo de búsqueda) y, en este caso, es preferible emplear una estructura asociativa.

Las técnicas de índices ordenados son preferibles a las estructuras asociativas en los casos en los que la consulta especifica un rango de valores. Estas consultas tienen el siguiente aspecto:

select $A_1, A_2, ..., A_n$
from *r*
where $A_i \leq c_2$ **and** $A_i \geq c_1$

En otras palabras, la consulta anterior busca todos los registros con A_i valores comprendidos entre c_1 y c_2.

Considérese la manera de procesar esta consulta empleando un índice ordenado. En primer lugar se realiza una búsqueda del valor c_1. Una vez que se ha encontrado el cajón que contiene el valor c_1, se sigue el orden de la cadena de punteros del índice para leer el siguiente cajón y se continúa de esta manera hasta que se llega a c_2.

Si, en vez de un índice ordenado, se tiene una estructura asociativa, se puede llevar a cabo una búsqueda de c_1 y localizar el cajón correspondiente—pero, en general, no resulta fácil determinar el cajón que hay que examinar a continuación. La dificultad surge porque una buena función de asociación asigna valores a los cajones de manera aleatoria. Por tanto, no hay un concepto sencillo de "siguiente cajón según el orden establecido". La razón por la que no se puede encadenar un cajón detrás de otro según un cierto orden de A_i es que cada cajón tiene asignado muchos valores de la clave de búsqueda. Como los valores están diseminados aleatoriamente según la función de asociación, es probable que los valores del rango especificado estén esparcidos por muchos cajones o, tal vez, por todos. Por esa razón, hay que leer todos los cajones para encontrar las claves de búsqueda necesarias.

Normalmente el diseñador elige la indexación ordenada, a menos que se sepa de antemano que las consultas de rango van a ser poco frecuentes, en cuyo caso se escoge la asociación. Las organizaciones asociativas resultan especialmente útiles para los archivos temporales creados durante el procesamiento de las consultas, siempre que se realicen búsquedas basadas en valores de la clave pero no se vayan a realizar consultas de rango.

12.9 Índices de mapas de bits

Los índices de mapas de bits son un tipo de índices especializado diseñado para la consulta sencilla sobre varias claves, aunque cada índice de mapas de bits se construya para una única clave.

Para que se utilicen los índices de mapas de bits, los registros de la relación deben estar numerados secuencialmente comenzando, por ejemplo, por 0. Dado un número n debe ser fácil recuperar el registro con número n. Esto resulta especialmente fácil de conseguir si los registros son de tamaño fijo y están asignados a bloques consecutivos de un archivo. El número de registro se puede traducir fácilmente en un número de bloque y en un número que identifica el registro dentro de ese bloque.

Considérese una relación r con un atributo A que sólo puede tomar como valor un número pequeño (por ejemplo, entre 2 y 20). Por ejemplo, la relación *info_cliente* puede tener el atributo *sexo*, que sólo

puede tomar los valores m (masculino) o f (femenino). Otro ejemplo puede ser el atributo *nivel_ingresos*, en el que los ingresos se han dividido en cinco niveles: $L1$: \$0 − 9999, $L2$: \$10.000 − 19.999, $L3$: 20.000 − 39.999, $L4$: 40.000 − 74.999 y $L5$: $75,000 - \infty$. Aquí, los datos originales pueden tomar muchos valores, pero un analista de datos los ha dividido en un número pequeño de rangos para simplificar su análisis.

12.9.1 Estructura de los índices de mapas de bits

Un **mapa de bits** es simplemente un array de bits. En su forma más sencilla, un **índice de mapas de bits** sobre el atributo A de la relación r consiste en un mapa de bits para cada valor que pueda tomar A. Cada mapa de bits tiene tantos bits como el número de registros de la relación. El i-ésimo bit del mapa de bits para el valor v_j se define como 1 si el registro con número i tiene el valor v_j para el atributo A. El resto de los bits del mapa de bits se define como 0.

En este ejemplo hay un mapa de bits para el valor m y otro para f. El i-ésimo bit del mapa de bits para m se define como 1 si el valor *sexo* del registro con número i es m. El resto de bits del mapa de bits de m se definen como 0. Análogamente, el mapa de bits de f tiene el valor 1 para los bits correspondientes a los registros con el valor f para el atributo *sexo*; el resto de bits tienen el valor 0. La Figura 12.32 muestra un ejemplo de índices de mapa de bits para la relación *info_cliente*.

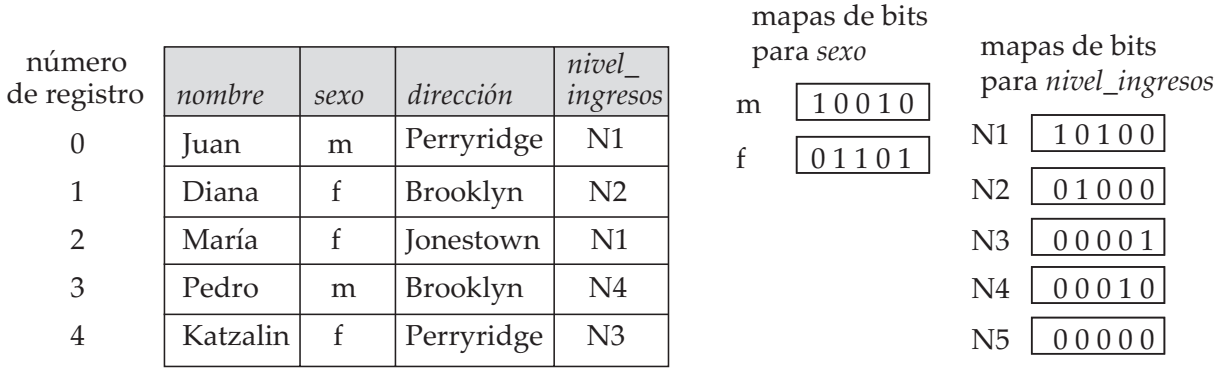
Ahora se considerará cuándo resultan útiles los mapas de bits. La manera más sencilla de recuperar todos los registros con el valor m (o f) sería simplemente leer todos los registros de la relación y seleccionar los de valor m (o f, respectivamente). El índice de mapas de bits no ayuda realmente a acelerar esa selección.

De hecho, los índices de mapas de bits resultan útiles para las selecciones sobre todo cuando hay selecciones bajo varias claves. Supóngase que se crea un índice de mapas de bits sobre el atributo *nivel_ingresos*, que ya se ha descrito antes, además del índice de mapas de bits para *sexo*.

Considérese ahora una consulta que seleccione mujeres con ingresos en el rango 10.000 − 19.999 €. Esta consulta se puede expresar como $\sigma_{sexo=\text{f} \wedge nivel_ingresos=L2}(r)$. Para evaluar esta selección se busca el valor f en el mapa de bits de *sexo* y el valor $L2$ en el de *nivel_ingresos* y se realiza la **intersección** (conjunción lógica) de los dos mapas de bits. En otras palabras, se calcula un nuevo mapa de bits en el que el bit i tenga el valor 1 si el i-ésimo bit de los dos mapas de bits es 1, y el valor 0 en caso contrario. En el ejemplo de la Figura 12.32, la intersección del mapa de bits de $sexo = \text{f}$ (01101) y el de $nivel_ingresos = L2$ (01000) da como resultado el mapa de bits 01000.

Como el primer atributo puede tomar dos valores y el segundo cinco, se puede esperar, en promedio, que sólo de 1 a 10 registros satisfagan la condición combinada de los dos atributos. Si hay más condiciones, es probable que la proporción de los registros que satisfacen todas las condiciones sea bastante pequeña. El sistema puede calcular así el resultado de la consulta buscando todos los bits con valor 1 del mapa de bits resultado de la intersección y recuperando los registros correspondientes. Si la proporción es grande, la exploración de la relación completa seguirá siendo la alternativa menos costosa.

Otro uso importante de los mapas de bits es el recuento del número de tuplas que satisfacen una selección dada. Esas consultas son importantes para el análisis de datos. Por ejemplo, si se desea de-



número de registro	*nombre*	*sexo*	*dirección*	*nivel_ingresos*
0	Juan	m	Perryridge	N1
1	Diana	f	Brooklyn	N2
2	María	f	Jonestown	N1
3	Pedro	m	Brooklyn	N4
4	Katzalin	f	Perryridge	N3

Figura 12.32 Índices de mapas de bits para la relación *info_cliente*.

terminar el número de mujeres que tienen un nivel de ingresos $L2$, se calcula la intersección de los dos mapas de bits y luego se cuenta el número de bits de valor 1 en el mapa de bits resultado de la intersección. Así se puede obtener el resultado deseado del índice de mapa de bits sin ni siquiera acceder a la relación.

Los índices de mapas de bits son generalmente bastante pequeños en comparación con el tamaño real de la relación. Los registros suelen tener entre decenas y centenares de bytes de longitud, mientras que un solo bit representa a un registro en el mapa de bits. Por tanto, el espacio ocupado por cada mapa de bits suele ser menos del uno por ciento del espacio ocupado por la relación. Por ejemplo, si el tamaño del registro de una relación dada es de 100 bytes, el espacio ocupado por cada mapa de bits sería la octava parte del uno por ciento del espacio ocupado por la relación. Si el atributo A de la relación sólo puede tomar uno valor de entre ocho, el índice de mapas de bits de ese atributo consiste en ocho mapas de bits que, juntos, sólo ocupan el uno por ciento del tamaño de la relación.

El borrado de registros crea huecos en la secuencia de registros, ya que el desplazamiento de registros (o de los números de registro) para rellenar los huecos resultaría excesivamente costoso. Para reconocer los registros borrados se puede almacenar un **mapa de bits de existencia** en el que el bit i sea 0 si el registro i no existe, y 1 en caso contrario. Se verá la necesidad de la existencia de los mapas de bits en el Apartado 12.9.2. La inserción de registros no debe afectar a la secuencia de numeración de los demás registros. Por tanto, se puede insertar tanto añadiendo registros al final del archivo como reemplazando los registros borrados.

12.9.2 Implementación eficiente de las operaciones de mapas de bits

Se puede calcular fácilmente la intersección de dos mapas de bits usando un bucle **for**: la iteración i-ésima del bucle calcula la conjunción (**and**) de los bits i-ésimos de los dos mapas de bits. Se puede acelerar considerablemente el cálculo de la intersección usando las instrucciones de bits **and** soportadas por la mayoría de los conjuntos de instrucciones de las computadoras. Cada *palabra* suele constar de 32 o 64 bits, en función de la arquitectura de la computadora. La instrucción de bits **and** toma dos palabras como entrada y devuelve una palabra en que cada bit es la conjunción lógica de los bits de igual posición de las palabras de entrada. Lo que es importante observar es que una sola instrucción de bits **and** puede calcular la intersección de 32 o de 64 bits *a la vez*.

Si una relación tuviese un millón de registros, cada mapa de bits contendría un millón de bits, o, lo que es equivalente, 128 kilobytes. Sólo se necesitan 31.250 instrucciones para calcular la intersección de dos mapas de bits de la relación, suponiendo un tamaño de palabra de 32 bits. Por tanto, el cálculo de intersecciones de mapas de bits es una operación extremadamente rápida.

Al igual que la intersección de mapas de bits resulta útil para calcular la conjunción de dos condiciones, la unión de mapas de bits resulta útil para calcular la disyunción de dos condiciones. El procedimiento para la unión de mapas de bits es exactamente igual que el de la intersección, salvo que se utiliza la instrucción de bits **or** en lugar de **and**.

La operación complemento se puede usar para calcular un predicado que incluya la negación de una condición, como **not** (*nivel-ingresos* = $L1$). El complemento de un mapa de bits se genera complementando cada uno de sus bits (el complemento de 1 es 0 y el complemento de 0 is 1). Puede parecer que **not** (*nivel_ingresos* = $L1$) se puede implementar simplemente calculando el complemento del mapa de bits del nivel de ingresos $L1$. Sin embargo, si se ha borrado algún registro, el mero cálculo del complemento del mapa de bits no resulta suficiente. Los bits correspondientes a esos registros serán 0 en el mapa de bits original, pero pasarán a ser 1 en el complemento, aunque el registro no exista. También surge un problema similar cuando el valor de un atributo es *nulo*. Por ejemplo, si el valor de *nivel_ingresos* es nulo, el bit será 0 en el mapa de bits original para el valor $L1$ y 1 en el complementado.

Para asegurarse de que los bits correspondientes a registros borrados se definen como 0 en el resultado, hay que intersecar el mapa de bits complementado con el mapa de bits de existencia para desactivar los bits de los registros borrados. Análogamente, para manejar los valores nulos, también se debe intersecar el mapa de bits complementado con el complemento del mapa de bits para el valor *nulo*.[1]

1. El tratamiento de predicados como **is unknown** puede causar aún más complicaciones, que requerirían en general el uso de un mapa de bits adicional para determinar los resultados de las operaciones que son desconocidos.

Se puede contar rápidamente el número de bits que valen 1 en el mapa de bits si se emplea una técnica inteligente. Se puede mantener un array de 256 entradas, donde la entrada i-ésima almacene el número de bits que valen 1 en la representación binaria de i. Hay que definir el recuento inicial como 0. Se toma cada byte del mapa de bits, se usa para indexar en el array y se añade el recuento almacenado al recuento total. El número de operaciones de suma sería la octava parte del número de tuplas y, por tanto el proceso de recuento es muy eficiente. Un gran array (que emplee $2^{16} = 65.536$ entradas), indexado por pares de bytes, dará incluso aceleraciones mayores, pero con un coste de almacenamiento mayor.

12.9.3 Mapas de bits y árboles B^+

Los mapas de bits se pueden combinar con los índices normales de árboles B^+ para las relaciones donde unos pocos valores de los atributos sean extremadamente frecuentes y también aparezcan otros valores, pero con mucha menor frecuencia. En las hojas de los índices de los árboles B^+, para cada valor se suele mantener una lista de todos los registros con ese valor para el atributo indexado. Cada elemento de la lista sería un identificador de registro que consta, al menos, de 32 bits, y normalmente más. Para cada valor que aparece en muchos registros se almacena un mapa de bits en lugar de una lista de registros.

Supónganse que un valor dado v_i aparece en la dieciseisava parte de los registros de una relación. Sea N el número de registros de la relación y supóngase que cada registro tiene un número de 64 bits que lo identifica. El mapa de bits sólo necesita sólo un bit por registro, o N en total. En cambio, la representación de lista necesita sesenta y cuatro bits por registro en el que aparezca el valor, o $64 * N/16 = 4N$ bits. Por tanto, es preferible el mapa de bits para representar la lista de registros del valor v_i. En el ejemplo (con un identificador de registros de 64 bits), si menos de uno de cada sesenta y cuatro registros tiene un valor dado, es preferible la representación de lista de registros para la identificación de los registros con ese valor, ya que emplea menos bits que la representación con mapas de bits. Si más de uno de cada sesenta y cuatro registros tiene ese valor dado, es preferible la representación de mapas de bits.

Por tanto, los mapas de bits se pueden emplear como mecanismo de almacenamiento comprimido en los nodos hoja de los árboles B^+ de los valores que aparecen muy frecuentemente.

12.10 Definición de índices en SQL

La norma SQL no proporciona al usuario o administrador de la base de datos ninguna manera de controlar qué índices se crean y se mantienen por el sistema de base de datos. Los índices no se necesitan para la corrección, ya que son estructuras de datos redundantes. Sin embargo, los índices son importantes para el procesamiento eficiente de las transacciones, incluyendo las transacciones de actualización y consulta. Los índices son también importantes para un cumplimiento eficiente de las ligaduras de integridad. Por ejemplo, las implementaciones típicas obligan a declarar una clave (Capítulo 4) mediante la creación de un índice con la clave declarada como la clave de búsqueda del índice.

En principio, un sistema de base de datos puede decidir automáticamente qué índices crear. Sin embargo, debido al coste en espacio de los índices, así como el efecto de los índices en el procesamiento de actualizaciones, no es fácil hacer una elección apropiada automáticamente sobre qué índices mantener. Por este motivo, la mayoría de las implementaciones de SQL proporcionan al programador control sobre la creación y eliminación de índices mediante comandos del lenguaje de definición de datos.

A continuación se ilustrará las sintaxis de estos comandos. Aunque la sintaxis que se muestra se usa ampliamente y está soportada en muchos sistemas de bases de datos, no es parte de la norma SQL:1999. Las normas SQL (hasta SQL:1999, al menos) no dan soporte al control del esquema físico de la base de datos y este libro se limita al esquema lógico de la base de datos.

Un índice se crea mediante el comando **create index**, que tiene la forma

create index <nombre-índice> **on** <nombre-relación> (<lista-atributos>)

lista-atributos es la lista de atributos de la relación que constituye la clave de búsqueda del índice

Para definir un índice llamado *índice_sucursal* de la relación *sucursal* con la clave de búsqueda *nombre_sucursal*, se escribe

create index *índice_sucursal* **on** *sucursal* (*nombre_sucursal*)

Si se desea declarar que la clave de búsqueda es una clave candidata hay que añadir el atributo **unique** a la definición del índice. Con esto, el comando

create unique index *índice_sucursal* **on** *sucursal* (*nombre_sucursal*)

declara *nombre_sucursal* como una clave candidata de *sucursal*. Si cuando se introduce el comando **create unique index**, *nombre_sucursal* no es una clave candidata, se mostrará un mensaje de error y el intento de crear un índice fallará. Por otro lado, si el intento de crear el índice ha tenido éxito, cualquier intento de insertar una tupla que viole la declaración de clave fallará. Hay que observar que el carácter **unique** es redundante si el sistema de bases de datos soporta la declaración **unique** de SQL estándar.

Muchos sistemas de bases de datos ofrecen un modo de especificar el tipo de índice que se va a utilizar (como los árboles B^+ o la asociación). Algunos sistemas de bases de datos permiten también que se declare uno de los índices de una relación como agrupado; el sistema almacena entonces la relación ordenada de acuerdo con la clave de búsqueda del índice agrupado.

Para hacer posible la eliminación (drop) de un índice es necesario especificar su nombre. El comando **drop index** tiene la forma:

drop index <*nombre_índice*>

12.11 Resumen

- Muchas consultas solamente hacen referencia a una pequeña proporción de los registros de un archivo. Para reducir el gasto adicional en la búsqueda de estos registros se pueden construir *índices* para los archivos almacenados en la base de datos.
- Los archivos secuenciales indexados son unos de los esquemas de índice más antiguos usados en los sistemas de bases de datos. Para permitir una rápida recuperación de los registros según el orden de la clave de búsqueda, los registros se almacenan consecutivamente y los que no siguen el orden se encadenan entre sí. Para permitir un acceso aleatorio, se usan estructuras índice.
- Existen dos tipos de índices que se pueden utilizar: los índices densos y los índices dispersos. Los índices densos contienen una entrada por cada valor de la clave de búsqueda, mientras que los índices dispersos contienen entradas sólo para algunos de esos valores.
- Si el orden de una clave de búsqueda se corresponde con el orden secuencial del archivo, un índice sobre la clave de búsqueda se conoce como *índice con agrupación*. Los otros índices son los *índices sin agrupación* o *secundarios*. Los índices secundarios mejoran el rendimiento de las consultas que utilizan otras claves de búsqueda distinta de la del índice con agrupación. Sin embargo, éstas implican un gasto adicional en la modificación de la base de datos.
- El inconveniente principal de la organización del archivo secuencial indexado es que el rendimiento disminuye según crece el archivo. Para superar esta deficiencia se puede usar un *índice de árbol* B^+.
- Un índice de árbol B^+ tiene la forma de un árbol *equilibrado*, en el cual cada camino de la raíz a las hojas del árbol tiene la misma longitud. La altura de un árbol B^+ es proporcional al logaritmo en base N del número de registros de la relación, donde cada nodo interno almacena N punteros; el valor de N está usualmente entre 50 y 100. Los árboles B^+ son más cortos que otras estructuras de árboles binarios equilibrados como los árboles AVL y, por tanto, necesitan menos accesos a disco para localizar los registros.
- Las búsquedas en un índice de árbol B^+ son directas y eficientes. Sin embargo, la inserción y el borrado son algo más complicados pero eficientes. El número de operaciones que se necesitan para la inserción y borrado en un árbol B^+ es proporcional al logaritmo en base N del número de registros de la relación, donde cada nodo interno almacena N punteros.
- Se pueden utilizar los árboles B^+ tanto para indexar un archivo con registros, como para organizar los registros de un archivo.

- Los índices de árbol B son similares a los índices de árbol B^+. La mayor ventaja de un árbol B es que el árbol B elimina el almacenamiento redundante de los valores de la clave de búsqueda. Los inconvenientes principales son la complejidad y el reducido grado de salida para un tamaño de nodo dado. En la práctica, los índices de árbol B^+ están universalmente mejor considerados que los índices de árbol B por los diseñadores de sistemas.
- Las organizaciones de archivos secuenciales necesitan una estructura de índice para localizar los datos. Los archivos con organizaciones basadas en asociación, en cambio, permiten encontrar la dirección de un elemento de datos directamente mediante el cálculo de una función con el valor de la clave de búsqueda del registro deseado. Ya que no se sabe a la hora de diseñar la manera precisa en la cual los valores de la clave de búsqueda se van a almacenar en el archivo, una buena función de asociación a elegir es la que distribuya los valores de la clave de búsqueda a los cajones de una manera uniforme y aleatoria.
- La *asociación estática* utiliza una función de asociación en la que el conjunto de direcciones de cajones está fijado. Estas funciones de asociación no se pueden adaptar fácilmente a las bases de datos que tengan un crecimiento significativo con el tiempo. Hay varias *técnicas de asociación dinámica* que permiten que la función de asociación cambie. Un ejemplo es la *asociación extensible*, que trata los cambios de tamaño de la base datos mediante la división y fusión de cajones según crezca o disminuya la base de datos.
- También se puede utilizar la asociación para crear índices secundarios; tales índices se llaman *índices asociativos*. Por motivos de notación se asume que las organizaciones de archivos asociativos tienen un índice asociativo implícito en la clave de búsqueda usada para la asociación.
- Los índices ordenados con árboles B^+ y con índices asociativos se pueden usar para la selección basada en condiciones de igualdad que involucren varios atributos. Cuando hay varios atributos en una condición de selección se pueden intersecar los identificadores de los registros recuperados con los diferentes índices.
- Los índices de mapas de bits proporcionan una representación muy compacta para la indexación de atributos con muy pocos valores distintos. Las operaciones de intersección son extremadamente rápidas en los mapas de bits, haciéndolos ideales para el soporte de consultas con varios atributos.

Términos de repaso

- Tipos de acceso.
- Tiempo de acceso.
- Tiempo de inserción.
- Tiempo de borrado.
- Espacio adicional.
- Índice ordenado.
- Índice con agrupación.
- Índice primario.
- Índice sin agrupación.
- Índice secundario.
- Archivo secuencial indexado.
- Registro/entrada del índice.
- Índice denso.
- Índice disperso.
- Índice multinivel.
- Clave compuesta.
- Exploración secuencial.
- Índice de árbol B^+.
- Árbol equilibrado.
- Organización de archivos con árboles B^+.
- Índice de árbol B.
- Asociación estática.
- Organización de archivos asociativos.
- Índice asociativo.
- Cajón.
- Función de asociación.
- Desbordamiento de cajones.
- Atasco.
- Asociación cerrada.
- Asociación dinámica.
- Asociación extensible.

- Acceso bajo varias claves.
- Índices sobre varias claves.
- Índice de mapas de bits.
- Operaciones de mapas de bits:
  - ☐ Intersección.
  - ☐ Unión.
  - ☐ Complemento.
  - ☐ Mapa de bits de existencia.

Ejercicios prácticos

12.1 Dado que los índices agilizan el procesamiento de consultas, ¿por qué no deberían de mantenerse en varias claves de búsqueda? Enumérense tantas razones como sea posible.

12.2 ¿Es posible en general tener dos índices con agrupación en la misma relación para dos claves de búsqueda diferentes? Razónese la respuesta.

12.3 Constrúyase un árbol B^+ con el siguiente conjunto de valores de la clave:

(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)

Supóngase que el árbol está inicialmente vacío y que se añaden los valores en orden ascendente. Constrúyanse árboles B^+ para los casos en los que el número de punteros que caben en un nodo son:

a. Cuatro
b. Seis
c. Ocho

12.4 Para cada árbol B^+ del Ejercicio práctico 12.3 muéstrese el aspecto del árbol después de cada una de las siguientes operaciones:

a. Insertar 9.
b. Insertar 10.
c. Insertar 8.
d. Borrar 23.
e. Borrar 19.

12.5 Considérese el esquema modificado de redistribución para árboles B^+ descrito en la página 414. ¿Cuál es la altura esperada del árbol en función de n?

12.6 Repítase el Ejercicio práctico 12.3 para un árbol B.

12.7 Supóngase que se está usando la asociación extensible en un archivo que contiene registros con los siguientes valores de la clave de búsqueda:

2, 3, 5, 7, 11, 17, 19, 23, 29, 31

Muéstrese la estructura asociativa extensible para este archivo si la función de asociación es $h(x) = x$ mod 8 y los cajones pueden contener hasta tres registros.

12.8 Muéstrese cómo cambia la estructura asociativa extensible del Ejercicio práctico 12.7 como resultado de realizar los siguientes pasos:

a. Borrar 11.
b. Borrar 31.
c. Insertar 1.
d. Insertar 15.

12.9 Dese un pseudocódigo para el borrado de entradas de una estructura asociativa extensible, incluyendo detalles del momento y forma de fusionar cajones. No se debe considerar la reducción del tamaño de la tabla de direcciones de cajones.

12.10 Sugiérase una forma eficaz de comprobar si la tabla de direcciones de cajones en una asociación extensible se puede reducir en tamaño almacenando un recuento extra con la tabla de direcciones

de cajones. Dense detalles de cómo se debería mantener el recuento cuando se dividen, fusionan o borran los cajones.

Nota: la reducción del tamaño de la tabla de direcciones de cajones es una operación costosa y las inserciones subsecuentes pueden causar que la tabla vuelva a crecer. Por tanto, es mejor no reducir el tamaño tan pronto como se pueda, sino solamente si el número de entradas de índice es pequeño en comparación con el tamaño de la tabla de direcciones de los cajones.

12.11 Considérese la relación *cuenta* mostrada en la Figura 12.25.

- **a.** Constrúyase un índice de mapa de bits sobre los atributos *nombre_sucursal* y *saldo*, dividiendo *saldo* en cuatro rangos: menores que 250, entre 250 y menor que 500, entre 500 y menor que 750, y 750 o mayor.
- **b.** Considérese una consulta que solicite todas las cuentas de Daimiel con un saldo de 500 o más. Descríbanse los pasos para responder a la consulta y muéstrense los mapas de bits finales e intermedios construidos para responder la consulta.

12.12 Supóngase que se tiene una relación con n_r tuplas sobre la que se va a construir un índice secundario de árbol B$^+$.

- **a.** Dese una fórmula para el coste de construir el índice de árbol B$^+$ insertando un registro cada vez. Supóngase que cada página contendrá en media f entradas, y que todos los niveles del árbol sobre la raíz están en memoria.
- **b.** Suponiendo un tiempo de acceso a disco de 10 milisegundos, ¿cuál es el coste de la construcción del índice para una relación con 10 millones de registros?
- **c.** Sugiérase una manera *ascendente* más eficiente de crear el índice, construyendo en primer lugar todo el nivel de hojas y luego los niveles superiores uno a uno. Supóngase que se tiene una función que puede ordenar de manera eficiente un conjunto de registros muy grande, aunque sea tan grande que no cabe en memoria. (Estos algoritmos de ordenación se describen más adelante, en el Apartado 13.4 y, suponiendo una cantidad razonable de memoria principal tienen un coste de alrededor de una operación de E/S por bloque.)

Ejercicios

12.13 ¿Cuándo es preferible utilizar un índice denso en vez de un índice disperso? Razónese la respuesta.

12.14 ¿Cuál es la diferencia entre un índice con agrupación y un índice secundario?

12.15 Para cada árbol B$^+$ del Ejercicio práctico 12.3 muéstrense los pasos involucrados en las siguientes consultas:

- **a.** Encontrar los registros con un valor de la clave de búsqueda de 11.
- **b.** Encontrar los registros con un valor de la clave de búsqueda entre 7 y 17, ambos inclusive.

12.16 La solución presentada en el Apartado 12.5.3 para tratar las claves de búsqueda duplicadas añadían un atributo adicional a la clave de búsqueda. ¿Qué efecto tiene este cambio en la altura del árbol B$^+$?

12.17 Explíquense las diferencias entre la asociación abierta y la cerrada. Coméntense los beneficios de cada técnica en aplicaciones de bases de datos.

12.18 ¿Cuáles son las causas del desbordamiento de cajones en un archivo con una organización asociativa? ¿Qué se puede hacer para reducir la aparición del desbordamiento de cajones?

12.19 ¿Por qué una estructura asociativa no es la mejor elección para una clave de búsqueda en la que son probables las consultas de rangos?

12.20 Supóngase la relación $R(A, B, C)$, con un índice de árbol B$^+$ con la clave de búsqueda (A, B).

- **a.** ¿Cuál es el máximo coste posible de buscar registros que satisfagan $10 < A < 50$ al usar este índice en términos del número de registros recuperados n_1 y la altura h del árbol?

b. ¿Cuál es el máximo coste posible de buscar registros que satisfagan $10 < A < 50 \wedge 5 < B < 10$ al usar este índice en términos del número de registros recuperados n_2, y de n_1 y de h, como se definieron antes?

c. ¿Bajo qué condiciones sobre n_1 y sobre n_2 el índice sería una forma eficiente de buscar los registros que satisfagan $10 < A < 50 \wedge 5 < B < 10$?

12.21 Supóngase que hay que crear un índice de árbol B^+ sobre un gran número de nombres, donde el tamaño máximo de un nombre puede ser grande (por ejemplo, 40 caracteres) y el tamaño medio también (10 caracteres). Explíquese cómo se puede usar la compresión de prefijo para maximizar el grado de salida medio de los nodos internos.

12.22 ¿Por qué podrían perder secuencialidad los nodos hoja de una organización de archivos con árboles B^+? Sugiérase cómo se podría reorganizar para restaurar la secuencialidad.

12.23 Supóngase una relación almacenada en una organización de archivo de árbol B+. Supóngase que los índices secundarios con identificadores de registro que son punteros a los registros del disco.

a. ¿Cuál sería el efecto sobre los índices secundarios si ocurre una división de página en la organización de archivo?

b. ¿Cuál sería el coste de la actualización de todos los registros afectados en un índice secundario?

c. ¿Cómo soluciona este problema el uso de una clave de búsqueda como un identificador lógico de registro?

d. ¿Cuál es el coste adicional debido al uso de estos identificadores lógicos?

12.24 Muéstrese la forma de calcular mapas de existencia a partir de otros mapas de bits. Asegúrese de que la técnica funciona incluso con valores nulos, usando un mapa de bits para el valor *nulo.*

12.25 ¿Cómo afecta el cifrado de datos a los esquemas de índices? En particular, ¿cómo afectaría a los esquemas que intentan almacenar los datos de manera ordenada?

12.26 Nuestra descripción de la asociación estática asume que una gran extensión contigua de bloques de disco se pueden ubicar en una tabla asociativa estática. Supóngase que se pueden asignar sólo C bloques contiguos. Sugiérase cómo implementar la tabla asociativa si puede ser mucho más grande que los C bloques. El acceso a bloque debería seguir siendo eficiente.

Notas bibliográficas

En Cormen et al. [1990] se pueden encontrar explicaciones acerca de las estructuras básicas utilizadas en la indexación y asociación. Los índices de árbol B se introdujeron por primera vez en Bayer [1972] y en Bayer y McCreight [1972]. Los árboles B^+ se tratan en Comer [1979], Bayer y Unterauer [1977] y Knuth [1973]. Las notas bibliográficas del Capítulo 16 proporcionan referencias a la investigación sobre los accesos concurrentes y las actualizaciones en los árboles B^+. Gray y Reuter [1993] proporciona una buena descripción de los resultados en la implementación de árboles B^+.

Se han propuesto varias estructuras alternativas de árboles y basadas en árboles. Los **tries** son unos árboles cuya estructuras está basada en los "dígitos" de las claves (por ejemplo, el índice de muescas de un diccionario, con una entrada para cada letra). Estos árboles podrían no estar equilibrados en el sentido que lo están los árboles B^+. Los tries se estudian en Ramesh et al. [1989], Orenstein [1982], Litwin [1981] y Fredkin [1960]. Otros trabajos relacionados son los árboles B digitales de Lomet [1981]. Knuth [1973] analiza un gran número de técnicas de asociación distintas. Existen varias técnicas de asociación dinámica. Fagin et al. [1979] introducen la asociación extensible. La asociación lineal se introduce en Litwin [1978] y Litwin [1980]; en Rathi et al. [1990] se presentó una comparación de rendimiento con la asociación extensible. Una alternativa propuesta en Ramakrishna y Larson [1989] permite la recuperación en un solo acceso a disco al precio de una gran sobrecarga en una pequeña fracción de las modificaciones de la base de datos. La asociación dividida es una extensión de la asociación para varios atributos, y se trata en Rivest [1976], Burkhard [1976] y Burkhard [1979].

Vitter [2001] proporciona una extensa visión general de las estructuras de datos en memoria externa y algoritmos para ellas.

Los índices de mapas de bits y las variantes denominadas **índices por capas de bits** e **índices de proyección** se describen en O'Neil y Quass [1997]. Se introdujeron por primera vez en el gestor de archivos Model 204 de IBM sobre la plataforma AS 400. Proporcionan grandes ganancias de velocidad en ciertos tipos de consultas y se encuentran implementadas actualmente en la mayoría de sistemas de bases de datos. Entre la investigación más reciente en índices de mapas de bits figura la de Wu y Buchmann [1998], Chan y Ioannidis [1998], Chan y Ioannidis [1999] y Johnson [1999].

CAPÍTULO 13

Procesamiento de consultas

El **procesamiento de consultas** hace referencia a una serie de actividades implicadas en la extracción de datos de una base de datos. Estas actividades incluyen la traducción de consultas expresadas en lenguajes de bases de datos de alto nivel en expresiones implementadas en el nivel físico del sistema, así como transformaciones de optimización de consultas y la evaluación real de las mismas.

13.1 Visión general

En la Figura 13.1 se ilustran los pasos involucrados en el procesamiento de una consulta. Los pasos básicos son:

1. Análisis y traducción.
2. Optimización.
3. Evaluación.

Antes de empezar el procesamiento de una consulta, el sistema debe traducirla a una forma utilizable. Un lenguaje como SQL es adecuado para el uso humano, pero es poco apropiado para una representación interna en el sistema de la consulta. Una representación interna más útil estaría basada en el álgebra relacional extendido.

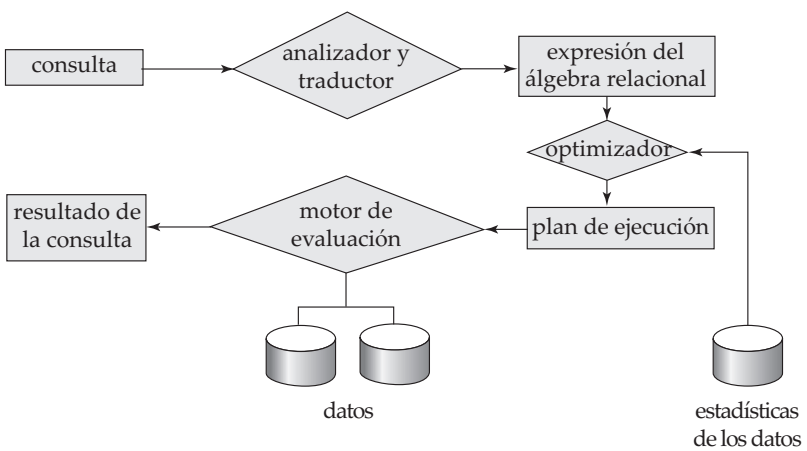


Figura 13.1 Pasos en el procesamiento de una consulta.

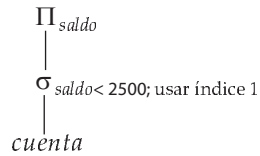


Figura 13.2 Plan de ejecución de una consulta.

Así, la primera acción que el sistema tiene que emprender para procesar una consulta es su traducción a su formato interno. Este proceso de traducción es similar al trabajo que realiza el analizador de un compilador. Durante la generación del formato interno de una consulta, el analizador comprueba la sintaxis de la consulta del usuario, verifica que los nombres de las relaciones que aparecen en ella sean nombres de relaciones en la base de datos, etc. Posteriormente se construye un árbol para el análisis de la consulta, que se transformará en una expresión del álgebra relacional. Si la consulta estuviera expresada en términos de una vista, la fase de traducción también sustituye todas las referencias a vistas por las expresiones del álgebra relacional que las definen[1]. El análisis de lenguajes se describe en la mayoría de los libros sobre compiladores (véanse las notas bibliográficas).

Dada una consulta hay generalmente distintos métodos para obtener la respuesta. Por ejemplo, ya se ha visto que en SQL se puede expresar una consulta de diferentes maneras. Cada consulta SQL puede traducirse a diferentes expresiones del álgebra relacional. Además de esto, la representación de una consulta en el álgebra relacional especifica de manera parcial cómo evaluar la consulta; existen normalmente varias maneras de evaluar expresiones del álgebra relacional. Como ejemplo, considérese la consulta:

select *saldo*
from *cuenta*
where *saldo* < 2500

Esta consulta se puede traducir en alguna de las siguientes expresiones del álgebra relacional:

- $\sigma_{saldo<2500}\ (\Pi_{saldo}\ (cuenta))$
- $\Pi_{saldo}\ (\sigma_{saldo<2500}\ (cuenta))$

Además, se puede ejecutar cada operación del álgebra relacional utilizando alguno de los diferentes algoritmos. Por ejemplo, para implementar la selección anterior se puede examinar cada tupla de *cuenta* para encontrar las tuplas cuyo saldo sea menor que 2500. Por otro lado, si se dispone de un índice de árbol B^+ en el atributo *saldo*, se puede utilizar este índice para localizar las tuplas.

Para especificar completamente cómo evaluar una consulta, no basta con proporcionar la expresión del álgebra relacional, además hay que anotar en ella las instrucciones que especifiquen cómo evaluar cada operación. Estas anotaciones podrían ser el algoritmo a usar para una operación específica o el índice o índices concretos a utilizar. Las operaciones del álgebra relacional anotadas con instrucciones sobre su evaluación reciben el nombre de **primitivas de evaluación**. La secuencia de operaciones primitivas que se pueden utilizar en la evaluación de una consulta establece un **plan de ejecución de la consulta** o un **plan de evaluación de la consulta**. En la Figura 13.2 se ilustra un plan de evaluación para nuestro ejemplo de consulta, en el que se especifica un índice concreto (denotado en la figura como "índice 1") para la operación selección. El **motor de ejecución de consultas** escoge un plan de evaluación, lo ejecuta y devuelve su respuesta a la consulta.

Los diferentes planes de evaluación de una misma consulta dada pueden tener costes distintos. No se puede esperar que los usuarios escriban las consultas de manera que sugieran el plan de evaluación más eficiente. En su lugar, es responsabilidad del sistema construir un plan de evaluación de la consulta

1. Para vistas materializadas, la expresión que define la vista ha sido ya evaluada y almacenada. Por tanto, se puede usar la relación almacenada en lugar de reemplazar los usos de la vista por la expresión que define la vista. Las vistas recursivas se tratan de manera diferente mediante un procedimiento de búsqueda de punto fijo, según se vio en los Apartados 4.7 y 5.4.6.

que minimice el coste de la evaluación de la consulta; esta tarea se denomina *optimización de consultas*. El Capítulo 14 describe en detalle la optimización de consultas.

Una vez elegido el plan de la consulta, ésta se evalúa con este plan y se muestra su resultado.

La secuencia de pasos que se han descrito para procesar una consulta son representativos; no todas las bases de datos los siguen exactamente. Por ejemplo, en lugar de utilizar la representación del álgebra relacional, varias bases de datos usan una representación del árbol de análisis con anotaciones basada en la estructura de la consulta SQL. Sin embargo, los conceptos que se describen aquí constituyen la base del procesamiento de consultas en las bases de datos.

Para optimizar una consulta, el optimizador de consultas debe conocer el coste de cada operación. Aunque el coste exacto es difícil de calcular, dado que depende de muchos parámetros como la memoria real disponible, es posible obtener una estimación aproximada del coste de ejecución para cada operación.

En este capítulo se estudia la forma de evaluar operaciones individuales en un plan de consulta y cómo estimar su coste; se vuelve a la optimización de consultas en el Capítulo 14. En el Apartado 13.2 se describe cómo se mide el coste de una consulta. Desde el Apartado 13.3 hasta el 13.6 se estudia la evaluación de operaciones individuales del álgebra relacional. Varias operaciones se pueden agrupar en un **cauce**, en el que cada una de las ellas empieza trabajando sobre sus tuplas de entrada del mismo modo que si fuesen generadas por otra operación. En el Apartado 13.7 se examina cómo coordinar la ejecución de varias operaciones de un plan de evaluación de consultas, en particular, cómo usar las operaciones encauzadas para evitar escribir resultados intermedios en disco.

13.2 Medidas del coste de una consulta

El coste de la evaluación de una consulta se puede expresar en términos de diferentes recursos, incluyendo los accesos a disco, el tiempo de CPU en ejecutar una consulta y, en sistemas de bases de datos distribuidos o paralelos, el coste de la comunicación (que se estudiará más tarde en los Capítulos 21 y 22). El tiempo de respuesta para un plan de evaluación de una consulta (esto es, el tiempo de reloj que se necesita para ejecutar el plan), si se supone que no hay otra actividad ejecutándose en el sistema, podría tener en cuenta todos estos costes y utilizarlos como una buena medida del coste del plan.

Sin embargo, en grandes sistemas de bases de datos, el coste de acceso a los datos en disco es normalmente el más importante, ya que los accesos a disco son más lentos comparados con las operaciones en memoria. Además, la velocidad de la CPU aumenta mucho más rápidamente que las velocidades de los discos. Así, lo más probable es que el tiempo empleado en operaciones del disco siga influyendo en el tiempo total de ejecución de una consulta. Las estimaciones del tiempo de CPU son más difíciles de hacer porque dependen de detalles de bajo nivel del código de ejecución. Aunque los optimizadores reales de consultas consideran los costes de CPU, aquí se ignoran y sólo se considera el coste de los accesos a disco para medir el coste del plan de evaluación de una consulta.

Se usarán el *número de transferencias de bloques* de disco y el *número de búsquedas de disco* para medir el coste del acceso a datos en disco. Si al subsistema de disco le lleva una media de t_T segundos para transferir un bloque de datos, con un tiempo medio de acceso a bloque (tiempo de búsqueda en el disco más latencia rotacional) de t_S segundos, entoces una operación que transfiera b bloques y ejecute S búsquedas tardaría $b * t_T + S * t_S$ segundos. Los valores de t_T y de t_S se deben ajustar al disco usado, pero los valores normales de los discos de altas prestaciones actuales serían $t_S = 4$ milisegundos y $t_T = 0.1$ milisegundos, asumiendo un tamaño de bloque de 4 kilobytes y una velocidad de transferencia de 40 megabytes por segundo[2].

Se puede refinar aún más la estimación de coste distinguiendo entre las lecturas y escrituras de bloques, dado que normalmente se tarda el doble de tiempo en escribir un bloque que en leerlo de disco (debido a que los sistemas de disco leen los sectores después de escribirlos para comprobar que la escritura fue correcta). Para simplificar se ignorará este detalle y se propone al lector que desarrolle estimaciones de coste más precisas para distintas operaciones.

Las estimaciones de coste que se proporcionan no incluyen el coste de escribir el resultado final de una operación en disco. Esto se tendrá en cuenta cuando sea preciso. Los costes de todos los algoritmos

2. Algunos sistemas de bases de datos ejecutan búsquedas y transferencias de bloque de prueba para estimar los costes medios de búsqueda y transferencia de bloque, como parte del proceso de instalación del software.

que se consideran aquí dependen del tamaño de la memoria intermedia en la memoria principal. En el mejor caso, todos los datos se pueden leer en las memorias intermedias y no es necesario acceder de nuevo a disco. En el peor caso se asume que la memoria intermedia puede contener sólo unos pocos bloques de datos (aproximadamente un bloque por relación). Al presentar las estimaciones de coste se asume generalmente el peor caso.

Además, aunque se asuma que los datos se deben leer inicialmente de disco, es posible que el bloque al que se acceda pueda estar en la memoria intermedia. De nuevo y para simplificar se ignorará este efecto; como resultado el coste del acceso a disco real durante la ejecución de un plan puede ser menor que el coste estimado.

13.3 Operación selección

En el procesamiento de consultas, el **explorador de archivo** es el operador de nivel más bajo para acceder a los datos. Los exploradores de archivo son algoritmos de búsqueda que localizan y recuperan los registros que cumplen una condición de selección. En los sistemas relacionales, el explorador de archivo permite leer una relación completa en los casos en que la relación se almacena en un único archivo dedicado.

13.3.1 Algoritmos básicos

Considérese una operación selección en una relación cuyas tuplas se almacenan juntas en un archivo. A continuación se muestran dos algoritmos exploradores que implementan la operación selección:

- **A1** (**búsqueda lineal**). En una búsqueda lineal se explora cada bloque del archivo y se comprueban todos los registros para determinar si satisfacen o no la condición de selección. Para una selección sobre un atributo clave, el sistema puede terminar la exploración si encuentra el registro requerido sin necesidad de examinar los otros registros de la relación.

  El coste de la búsqueda lineal, en términos de operaciones E/S, es una búsqueda más b_r transferencias de bloque, donde b_r denota el número de bloques del archivo o, de forma equivalente, el coste temporal es $t_S + b_r * t_T$

  En una selección sobre un atributo clave el sistema puede terminar la exploración si se encuentra el registro requerido, sin examinar los otros registros de la relación. Las selecciones sobre atributos clave presentan un coste medio de $b_r/2$, pero en el peor caso aún tiene un coste de b_r transferencias de bloque más una búsqueda.

  Aunque podría ser más lento que otros algoritmos para la implementación de la selección, el algoritmo de búsqueda lineal se puede aplicar a cualquier archivo, sin importar su ordenación, la presencia de índices o la naturaleza de la operación selección. El resto de algoritmos que se estudiarán no son aplicables en todos los casos, pero cuando lo son, son más rápidos que la búsqueda lineal.

- **A2** (**búsqueda binaria**). Si el archivo está ordenado según un atributo y la condición de la selección es una comparación de igualdad en ese atributo, se puede utilizar una búsqueda binaria para localizar los registros que satisfacen la selección. El sistema realiza la búsqueda binaria de los bloques del archivo.

  En el peor caso, el número de bloques que deben ser examinados para encontrar los registros requeridos es $\lceil \log_2(b_r) \rceil$, donde b_r denota el número de bloques del archivo. Cada uno de estos accesos a bloque requiere una búsqueda además de una transferencia de bloque, y por tanto el coste temporal es $\lceil \log_2(b_r) \rceil * (t_T + t_S)$.

  Si la selección es sobre un atributo que no sea clave, es posible que más de un bloque contenga los registros requeridos, y el coste de la lectura de los bloques extra se debe añadir a la estimación del coste. Se puede calcular este número mediante la estimación del tamaño del resultado de la selección (que se explica en el Apartado 14.3) y dividiéndolo entre el número de registros que se almacenan por bloque de la relación. Se asume que estos bloques se almacenan contiguos por lo que sólo se añade un coste de transferencia t_T por cada bloque adicional.

13.3.2 Selecciones con índices

Las estructuras índice se denominan **caminos de acceso**, ya que proporcionan un camino a través del cual se pueden localizar y acceder a los datos. En el Capítulo 12 se señaló la eficiencia de leer los registros del archivo en un orden próximo al orden físico. Recuérdese que un *índice primario* (también conocido como *índice con agrupación*) es un índice que permite leer los registros de un archivo en el mismo orden que el orden físico del archivo. Un índice que no es primario se llama *índice secundario*.

Los algoritmos de búsqueda que utilizan un índice reciben el nombre de **exploraciones del índice**. Los índices ordenados, como los árboles B^+, también permiten acceder a las tuplas según cierto orden que es útil para la implementación de las consultas de rangos. Aunque los índices pueden proporcionar un acceso rápido, directo y ordenado, su uso implica un gasto adicional en los accesos a los bloques que contienen el índice. Utilizaremos el predicado de selección como guía en la elección del índice a usar en el procesamiento de la consulta. Los algoritmos de búsqueda que usan un índice son:

- **A3** (**índice primario, igualdad basada en la clave**). Para una condición de igualdad en un atributo clave con un índice primario se puede utilizar el índice para recuperar el único registro que satisface la correspondiente condición de igualdad.

  Si se usa un árbol B^+, el coste de la operación en términos de operaciones E/S es igual a la altura del árbol más una operación E/S para recuperar el registro[3]; cada una de las cuales requiere una búsqueda y una transferencia de bloque. Por tanto, el coste es $(h_i + 1) * (t_T + t_S)$, donde h_i denota la altura del índice[4].

- **A4** (**índice primario, igualdad basada en un atributo no clave**). Se pueden recuperar varios registros mediante el uso de un índice primario cuando la condición de selección especifica una comparación de igualdad en un atributo A que no sea clave. La única diferencia del caso anterior es que puede ser necesario recuperar varios registros. Sin embargo, estos registros estarían almacenados consecutivamente en el archivo ya que el archivo se ordena según la clave de búsqueda.

  El coste de la operación depende de la altura del árbol más el número de bloques que contengan registros con la clave de búsqueda especificada. Se necesita una búsqueda para cada nivel del árbol. Además se necesita una búsqueda para obtener el primer bloque que contenga el registro deseado; el resto de bloques se almacenan consecutivamente y no requieren más búsquedas. En concreto el coste es $h_i * (t_T + t_S) + t_S + b * t_T$, donde h_i denota la altura del índice y b denota el número de bloques que contienen bloques con la clave de búsqueda especificada.

- **A5** (**índice secundario, igualdad**). Las selecciones con una condición de igualdad pueden utilizar un índice secundario. Esta estrategia puede recuperar un único registro si la condición de igualdad es sobre una clave; puede que se recuperen varios registros si el campo índice no es clave.

  En el primer caso sólo se obtiene un registro, y el coste es igual a la altura del árbol más una operación E/S para recuperar el registro. Cada operación E/S requiere una búsqueda y una transferencia de bloque. El coste temporal en este caso es el mismo que el del índice primario (caso A3).

  En el segundo caso, cada registro puede residir en un bloque diferente, que puede provocar una operación E/S por cada registro recuperado, y cada operación E/S requiere una búsqueda y una transferencia de bloque. El coste podría llegar a ser incluso peor que el de la búsqueda lineal si se obtiene un gran número de registros. El coste temporal es en este caso $(h_i + n) * (t_S + t_T)$, donde n es el número de registros recuperados[5].

---

3. Si se usa un árbol B^+, no se requiere E/S adicional porque los registros se almacenan en las hojas del árbol. Se deberían hacer ajustes similares a algunos de los algoritmos descritos a continuación en este apartado cuando se usen árboles B^+.
4. Los optimizadores reales asumen generalmente que la raíz del árbol está en la memoria intermedia porque se accede frecuentemente a ella. Algunos optimizadores incluso asumen que todos menos las hojas se encuentran en memoria porque se accede a ellos con relativa frecuencia, y normalmente el uno por ciento de los nodos no son hojas. La fórmula del coste se debe modificar adecuadamente.
5. Si la memoria intermedia es grande, el bloque que contenga el registro puede estar ya en memoria principal. Es posible construir una estimación del coste *medio* o *esperado* de la selección teniendo en cuenta la probabilidad de que el bloque se encuentre ya en memoria. Para grandes memorias intermedias la estimación será mucho menor que la estimación del peor caso.

Como se describió en el Apartado 12.5.5, cuando los registros se almacenan en una organización de árbol B^+ u otras organizaciones que puedan requerir la reubicación de los registros, los índices secundarios generalmente no almacenan punteros a los registros[6]. En su lugar, estos índices almacenan los valores de los atributos usados como la clave de búsqueda en una organización de árbol B^+. El acceso a un registro mediante un índice secundario es por tanto más caro: primero se busca en el índice secundario para encontrar los valores de la clave de búsqueda del índice primario y después se busca en el índice primario para encontrar los registros. La fórmula del coste descrita para índices secundarios se tendrá que modificar adecuadamente si se usan estos índices.

13.3.3 Selecciones con condiciones de comparación

Considérese una selección de la forma $\sigma_{A \leq v}(r)$. Se pueden implementar utilizando búsqueda lineal o binaria, o con índices de alguna de las siguientes maneras:

- **A6 (índice primario, comparación)**. Se puede utilizar un índice ordenado primario (por ejemplo, un índice primario de árbol B^+) cuando la condición de selección sea una comparación. Para condiciones con comparaciones de la forma $A > v$ o $A \geq v$ se puede usar el índice primario sobre A para guiar la recuperación de las tuplas de la manera siguiente. Para el caso de $A \geq v$ se busca el valor de v en el índice para encontrar la primera tupla del archivo que tenga un valor de $A = v$. Un explorador de archivo comenzando en esa tupla y hasta el final del archivo devuelve todas las tuplas que satisfacen la condición. Para $A > v$, el explorador de archivo comienza con la primera tupla tal que $A > v$. La estimación de coste en este caso es idéntica a la del caso A4.

  Para comparaciones de la forma $A < v$ o $A \leq v$ no es necesario buscar en el índice. Para el caso de $A < v$ se utiliza un simple explorador partiendo del inicio del archivo y continuando hasta (pero sin incluirlo) la primera tupla con el atributo $A = v$. El caso $A \leq v$ es similar, excepto que el explorador continúa hasta (pero sin incluir) la primera tupla con el atributo $A > v$. En ninguno de los dos casos el índice es de utilidad alguna.

- **A7 (índice secundario, comparación)**. Se puede utilizar un índice secundario ordenado para guiar la recuperación bajo condiciones de comparación que contengan $<, \leq, \geq$, o $>$. Los bloques del índice del nivel más bajo se exploran o bien desde el valor más pequeño hasta v (para $<$ y $\leq$) o bien desde v hasta el valor mayor (para $>$ y $\geq$).

  El índice secundario proporciona punteros a los registros, pero para obtener los reales es necesario extraerlos usando los punteros. Este paso puede requerir una operación E/S por cada registro extraído, dado que los consecutivos pueden estar en diferentes bloques de disco; como antes, cada operación E/S requiere una búsqueda y una transferencia de bloque. Si el número de registros extraídos es grande, el uso del índice secundario puede ser incluso más caro que la búsqueda lineal. Por tanto, el índice secundario sólo se debería usar si se seleccionan muy pocos registros.

13.3.4 Implementación de selecciones complejas

Hasta ahora sólo se han considerado condiciones de selección simples de la forma $AopB$, donde op es una operación igualdad o de comparación. Se revisarán a continuación predicados de selección más complejos.

- **Conjunción**. Una *selección conjuntiva* es una selección de la forma

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(r)$$

- **Disyunción**. Una *selección disyuntiva* es una selección de la forma

$$\sigma_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n}(r)$$

6. Recuérdese que si se usan organizaciones de árboles B^+ para almacenar relaciones, los registros se pueden trasladar entre bloques cuando los nodos hoja se dividen o fusionan, y cuando se redistribuyen los registros.

Una condición disyuntiva se cumple mediante la unión de todos los registros que cumplen alguna de las condiciones θ_i.

- **Negación**. El resultado de una selección $\sigma_{\neg\theta}(r)$ es el conjunto de tuplas de r para las que la condición θ es falsa. A falta de valores nulos, este conjunto es simplemente el conjunto de tuplas que no están en $\sigma_\theta(r)$.

Se puede implementar una operación selección con una conjunción o una disyunción de condiciones sencillas utilizando alguno de los siguientes algoritmos:

- **A8** (**selección conjuntiva utilizando un indice**). Inicialmente hay que determinar si para un atributo hay disponible algún camino de acceso en alguna de las condiciones simples. Si lo hay, cualquiera de los algoritmos de selección A2 hasta A7 puede recuperar los registros que cumplan esa condición. Se completa la operación mediante la comprobación en la memoria intermedia de que cada registro recuperado cumpla o no el resto de condiciones simples.

  Para reducir el coste, se elige un θ_i y uno de los algoritmos entre A1 y A7 para el que la combinación resulte en el menor coste de $\sigma_{\theta_i}(r)$. El coste del algoritmo A8 está determinado por el coste del algoritmo elegido.

- **A9** (**selección conjuntiva utilizando un índice compuesto**). Puede que se disponga de un *índice compuesto* (es decir, un índice sobre varios atributos) apropiado para algunas selecciones conjuntivas. Si la selección especifica una condición de igualdad en dos o más atributos y existe un índice compuesto en estos campos con atributos combinados, entonces se podría buscar en el índice directamente. El tipo de índice determina cuál de los algoritmos A3, A4 o A5 se utilizará.

- **A10** (**selección conjuntiva mediante la intersección de identificadores**). Otra alternativa para implementar la operación selección conjuntiva implica la utilización de punteros a registros o identificadores de registros. Este algoritmo necesita índices con punteros a registros en los campos involucrados por cada condición individual. De este modo se explora cada índice en busca de punteros cuyas tuplas cumplan alguna condición individual. La intersección de todos los punteros recuperados forma el conjunto de punteros a tuplas que satisfacen la condición conjuntiva. Luego se usa el conjunto de punteros para recuperar los registros reales. Si no hubiera índices disponibles para algunas condiciones concretas entonces habría que comprobar el resto de condiciones de los registros recuperados.

  El coste del algoritmo **A10** es la suma de los costes de las cada una de las exploraciones del índice más el coste de recuperar los registros en la intersección de las listas recuperadas de punteros. Este coste se puede reducir ordenando la lista de punteros y recuperando registros en orden. Por tanto, (1) todos los punteros a los registros de un bloque van juntos, y así todos los registros seleccionados en el bloque se pueden recuperar usando una única operación E/S, y (2) los bloques se leen ordenados, minimizando el movimiento del brazo del disco. El Apartado 13.4 describe algunos algoritmos de ordenación.

- **A11** (**selección disyuntiva mediante la unión de identificadores**). Si se dispone de caminos de acceso en todas las condiciones de la selección disyuntiva, se explora cada índice en busca de punteros cuyas tuplas cumplan una condición individual. La unión de todos los punteros recuperados proporciona el conjunto de punteros a todas las tuplas que cumplen la condición disyuntiva. Después se utilizan estos punteros para recuperar los registros reales.

  Sin embargo, aunque sólo una de las condiciones no tenga un camino de acceso, se tiene que realizar una búsqueda lineal en la relación para encontrar todas las tuplas que cumplan la condición. Por tanto, aunque sólo exista una de estas condiciones en la disyunción, el método de acceso más eficiente es una exploración lineal, que comprueba durante la exploración la condición disyuntiva en cada tupla.

La implementación de selecciones con condiciones negativas se deja como ejercicio (Ejercicio práctico 13.6).

13.4 Ordenación

La ordenación de los datos juega un papel importante en los sistemas de bases de datos por dos razones. Primero, las consultas SQL pueden solicitar que los resultados sean ordenados. Segundo, e igualmente importante para el procesamiento de consultas, varias de las operaciones relacionales, como las reuniones, se pueden implementar eficientemente si las relaciones de entrada están ordenadas. Por este motivo se revisa la ordenación antes que la operación reunión en el Apartado 13.5.

La ordenación se puede conseguir mediante la construcción de un índice en la clave de ordenación y utilizando luego ese índice para leer la relación de manera ordenada. No obstante, este proceso ordena la relación sólo *lógicamente* a través de un índice, en lugar de *físicamente*. Por tanto, la lectura de las tuplas de manera ordenada podría implicar un acceso a disco (búsqueda más transferencia de bloque) para cada tupla, lo que puede ser muy costoso dado que el número de registros puede ser mucho mayor que el número de bloques. Por esta razón sería deseable ordenar las tuplas físicamente.

El problema de la ordenación se ha estudiado ampliamente para los casos en que la relación cabe completamente en memoria principal, y para el caso en el que la relación es mayor que la memoria. En el primer caso se utilizan técnicas de ordenación clásicas como la ordenación rápida (*quick-sort*). Aquí se estudia cómo tratar el segundo caso.

La ordenación de relaciones que no caben en memoria se denomina **ordenación externa**. La técnica más utilizada para la ordenación externa es normalmente el algoritmo de **ordenación–mezcla externa**. A continuación se describe el algoritmo de ordenación–mezcla externa. Sea M el número de marcos de página en la memoria intermedia de la memoria principal (el número de bloques de disco cuyos contenidos se pueden alojar en la memoria intermedia de la memoria principal).

1. En la primera etapa, se crean varias **secuencias** ordenadas; cada **secuencia** está ordenada, pero sólo contiene parte de los registros de la relación.

```
i = 0;
repeat
    leer M bloques, o bien de la relación, o bien del resto de la relación,
        según el que tenga menor número de bloques;
    ordenar la parte de la relación que está en la memoria;
    escribir los datos ordenados al archivo de secuencias S_i;
    i = i + 1;
until el final de la relación
```

2. En la segunda etapa, las secuencias se *mezclan*. Supóngase que, por ahora, el número total de secuencias N es menor que M, así que se puede asignar un marco de página para cada secuencia y reservar espacio para guardar una página con el resultado. La etapa de mezcla se lleva a cabo de la siguiente manera:

```
leer un bloque de cada uno de los N archivos S_i y
    guardarlos en una página de la memoria intermedia en memoria;
repeat
    elegir la primera tupla (según el orden) de entre todas las páginas
        de la memoria intermedia;
    escribir la tupla y suprimirla de la página de la memoria intermedia;
    if la página de la memoria intermedia de alguna secuencia S_i está vacía
      and not fin-de-archivo(S_i)
        then leer el siguiente bloque de S_i y guardarlo en la página de la memoria intermedia;
until todas las páginas de la memoria intermedia estén vacías
```

El resultado de la etapa de mezcla es la relación ya ordenada. El archivo de salida se almacena en una memoria intermedia para reducir el número de operaciones de escritura en el disco. La operación anterior de mezcla es una generalización de la mezcla de dos vías utilizado por el algoritmo normal de ordenación–mezcla en memoria; éste mezcla N secuencias, por eso se llama **mezcla de N vías**.

En general, si la relación es mucho mayor que la memoria, se podrían generar M o más secuencias en la primera etapa y no sería posible asignar un marco de página para cada secuencia durante la etapa de mezcla. En este caso, se realiza la operación de mezcla en varios ciclos. Como hay suficiente memoria para $M - 1$ páginas de la memoria intermedia de entrada, cada mezcla puede tomar $M - 1$ secuencias como entrada.

El *ciclo* inicial se realiza como sigue. Se mezclan las $M - 1$ secuencias primeras (según se ha descrito ya en el punto 2) para obtener un única secuencia en el siguiente ciclo. Luego se mezclan de manera similar las siguientes $M-1$ secuencias, continuando así hasta que todas las secuencias iniciales se hayan procesado. En este punto, el número de secuencias se ha reducido por un factor de $M - 1$. Si el número reducido de secuencias es todavía mayor o igual que M, se realiza otro ciclo con las secuencias creadas en el primer ciclo como entrada. Cada ciclo reduce el número de secuencias por un factor de $M - 1$. Estos ciclos se repiten tantas veces como sea necesario, hasta que el número de secuencias sea menor que M; momento en el que un último ciclo genera el resultado ordenado.

En la Figura 13.3 se ilustran los pasos de la ordenación–mezcla externa en una relación ficticia. Por motivos didácticos supóngase que solamente cabe una tupla en cada bloque ($f_r = 1$) y que la memoria puede contener como mucho tres marcos de página. Durante la etapa de mezcla se utilizan dos marcos de página como entrada y uno para la salida.

El coste de acceso a disco de la ordenación—mezcla externa se calcula de la siguiente manera: sea b_r el número de bloques que contienen registros de la relación r. En la primera etapa, se lee y se copia de nuevo cada bloque de la relación, lo que resulta en un total de $2b_r$ transferencias de bloques. El número inicial de secuencias es $\lceil b_r/M \rceil$. Puesto que el número de secuencias decrece en un factor de $M - 1$ en cada ciclo de la mezcla, el número total de ciclos requeridos viene dado por la expresión $\lceil \log_{M-1}(b_r/M) \rceil$. Cada uno de estos ciclos lee y copia todos los bloques de la relación una vez, con dos excepciones. En primer lugar, el ciclo final puede producir la ordenación como resultado sin escribir el resultado en disco. En segundo lugar, puede que haya secuencias que ni lean ni copien durante un ciclo (por ejemplo, si hay M secuencias que mezclar en un ciclo, se leen y se mezclan $M - 1$, y no se accede a una de las secuencias durante ese ciclo). Si se ignora el ahorro (relativamente pequeño) debido a este

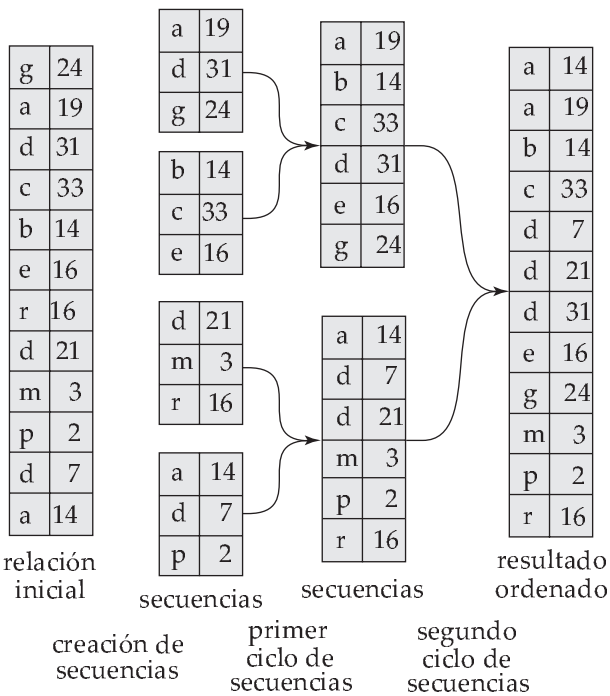


Figura 13.3 Ordenación externa utilizando ordenación–mezcla.

último efecto, el número total de transferencias de bloques para la ordenación externa de la relación es

$$b_r(2\lceil \log_{M-1}(b_r/M)\rceil + 1)$$

Aplicando esta ecuación al ejemplo de la Figura 13.3 se obtiene un total de $12*(4+1) = 60$ transferencias de bloques, como se puede comprobar en la figura. Obsérvese que este número no incluye el coste de escribir el resultado final.

También es necesario añadir los costes de la búsqueda en disco. La generación de ciclos requiere búsquedas para la lectura de datos para cada una de las secuencias, así como para escribir las secuencias. Durante la fase de mezcla, si se leen simultáneamente b_b bloques de cada secuencia (es decir, se asignan b_b bloques de memoria intermedia a cada secuencia), entonces cada paso de mezcla requeriría cerca de $\lceil b_r/b_b \rceil$ búsquedas para la lectura de los datos[7]. Aunque la salida se escriba secuencialmente, si está en el mismo disco que las secuencias de entrada, la cabeza se puede haber desplazado entre las escrituras de los bloques consecutivos. Así que habría que añadir un total de $2\lceil b_r/b_b \rceil$ búsquedas por cada paso de mezcla, excepto el paso final (ya que se asume que el resultado final no se escribe a disco). El número total de búsquedas es, por tanto:

$$2\lceil b_r/M \rceil + \lceil b_r/b_b \rceil (2\lceil \log_{M-1}(b_r/M)\rceil - 1)$$

Aplicando esta ecuación al ejemplo de la Figura 13.3 se obtiene un total de $8 + 12 * (2 * 2 - 1) = 44$ búsquedas si se establece el número de bloques por secuencia b_b como 1.

13.5 Operación reunión

En este apartado se estudian varios algoritmos para calcular la reunión de relaciones y para analizar sus costes asociados.

Se utiliza la palabra **equirreunión** para hacer referencia a las reuniones de la forma $r \bowtie_{r.A=s.B} s$, donde A y B son atributos o conjuntos de atributos de las relaciones r y s, respectivamente.

Utilizaremos como ejemplo la expresión

$$\textit{impositor} \bowtie \textit{cliente}$$

la cual utiliza los mismos esquemas de relación de los Capítulos 2 y 3. Se da por supuesta la siguiente información de catálogo acerca de las dos relaciones:

- Número de registros de *cliente*: $n_{cliente} = 10.000$.
- Número de bloques de *cliente*: $b_{cliente} = 400$.
- Número de registros de *impositor*: $n_{impositor} = 5.000$.
- Número de bloques de *impositor*: $b_{impositor} = 100$.

13.5.1 Reunión en bucle anidado

En la Figura 13.4 se muestra un algoritmo sencillo para calcular la reunión zeta, $r \bowtie_\theta s$, de dos relaciones r y s. Este algoritmo se llama de **reunión en bucle anidado**, ya que básicamente consiste en un par de bucles **for** anidados. La relación r se denomina la **relación externa** y s la **relación interna** de la reunión, puesto que el bucle de r incluye al bucle de s. El algoritmo utiliza la notación $t_r \cdot t_s$, donde t_r y t_s son tuplas; $t_r \cdot t_s$ denota la tupla construida mediante la concatenación de los valores de los atributos de las tuplas t_r y t_s.

Al igual que el algoritmo de búsqueda lineal en archivos de la selección, el algoritmo de reunión en bucle anidado tampoco necesita índices y se puede utilizar sin importar la condición de la reunión. La manera de extender el algoritmo para calcular la reunión natural es directa, puesto que la reunión natural se puede expresar como una reunión zeta seguida de la eliminación de los atributos repetidos mediante una proyección. El único cambio que se necesita es un paso adicional para borrar los atributos repetidos de la tupla $t_r \cdot t_s$ antes de añadirla al resultado.

7. Para ser más precisos, dado que cada secuencia se lee por separado y se pueden obtener menos de b_b bloques al leer el final de una secuencia, es posible que se necesite una búsqueda adicional por cada secuencia. Para simplificar se ignorará este detalle.

```
for each tupla t_r in r do begin
    for each tupla t_s in s do begin
        comprobar que el par (t_r, t_s) satisface la condición θ de la reunión
        si la cumple, añadir t_r · t_s al resultado.
    end
end
```

Figura 13.4 Reunión en bucle anidado.

```
for each bloque B_r of r do begin
    for each bloque B_s of s do begin
        for each tupla t_r in B_r do begin
            for each tupla t_s in B_s do begin
                comprobar que el par (t_r, t_s) satisface la condición de la reunión
                si la cumple, añadir t_r · t_s al resultado.
            end
        end
    end
end
```

Figura 13.5 Reunión en bucle anidado por bloques.

El algoritmo de reunión en bucle anidado es costoso, ya que examina cada pareja de tuplas en las dos relaciones. Considérese el coste del algoritmo de reunión en bucle anidado. El número de pares de tuplas a considerar es $n_r * n_s$, donde n_r denota el número de tuplas en r, y n_s denota el número de tuplas en s. Para registro de r se tiene que realizar una exploración completa de s. En el peor caso, la memoria intermedia solamente puede contener un bloque de cada relación, necesitándose un total de $n_r * b_s + b_r$ transferencias de bloques, donde b_r y b_s denotan el número de bloques que contienen tuplas de r y de s, respectivamente. Sólo se necesita una búsqueda por cada exploración de la relación interna s, dado que se lee secuencialmente, y un total de b_r búsquedas para leer r, por lo que resulta un total de $n_r + b_r$ búsquedas. En el mejor caso hay suficiente espacio para que las dos relaciones quepan en memoria, así que cada bloque se tendrá que leer solamente una vez; en consecuencia, sólo se necesitarán $b_r + b_s$ transferencias de bloques, además de 2 búsquedas.

Si una de las relaciones cabe en memoria por completo, es útil usar esa relación como la relación más interna, ya que solamente será necesario leer una vez la relación del bucle más interno. Por lo tanto, si s es lo suficientemente pequeña para caber en la memoria principal, esta estrategia necesita solamente un total de $b_r + b_s$ transferencias de bloques y dos búsquedas (el mismo coste que en el caso en que las dos relaciones quepan en memoria).

Considérese ahora el caso de la reunión natural de *impositor* y *cliente*. Supóngase que, por el momento, no hay ningún índice en cualquiera de las relaciones y que no se desea crear ninguno. Se pueden utilizar los bucles anidados para calcular la reunión; supóngase que *impositor* es la relación externa y que *cliente* es la relación interna de la reunión. Se tendrán que examinar $5.000 * 10.000 = 50 * 10^6$ pares de tuplas. En el peor de los casos el número de transferencias de bloques será de $5.000 * 400 + 100 = 2.000.100$, más $5.000 + 100 = 5.100$ búsquedas. En la mejor de las situaciones, sin embargo, se tienen que leer ambas relaciones solamente una vez y realizar el cálculo. Este cálculo necesita a lo sumo $100 + 400 = 500$ transferencias de bloques más dos búsquedas (una mejora significativa respecto de la peor situación posible). Si se hubiera utilizado *cliente* como la relación del bucle externo e *impositor* para el bucle interno, el coste en el peor de los casos de la última estrategia habría sido menor: $10.000 * 100 + 400 = 1.000.400$ transferencias de bloques más 10.400 búsquedas. El número de transferencias de bloques es significativamente menor y, aunque el número de búsquedas es superior, el coste total se reduce, suponiendo que $t_S = 4$ milisegundos y que $t_T = 0,1$ milisegundos.

13.5.2 Reunión en bucle anidado por bloques

Si la memoria intermedia es demasiado pequeña para contener las dos relaciones por completo en memoria, todavía se puede lograr un mayor ahorro en los accesos a los bloques si se procesan las relaciones por bloques en lugar de por tuplas. En la Figura 13.5 se muestra la **reunión en bucle anidado por bloques**, que es una variante de la reunión en bucle anidado donde se empareja cada bloque de la relación interna con cada bloque de la relación externa. En cada par de bloques se empareja cada tupla de un bloque con cada tupla del otro bloque para generar todos los pares de tuplas. Al igual que antes se añaden al resultado todas las parejas de tuplas que satisfacen la condición de la reunión.

La diferencia principal en coste entre la reunión en bucle anidado por bloques y la reunión en bucle anidado básica es que, en el peor de los casos, cada bloque de la relación interna s se lee solamente una vez por cada *bloque* de la relación externa, en lugar de una vez por cada *tupla* de la relación externa. De este modo, en el peor de los casos, habrá un total de $b_r * b_s + b_r$ transferencias de bloques, donde b_r y b_s denotan respectivamente el número de bloques que contienen registros de r y de s, respectivamente. Cada exploración de la relación interna requiere una búsqueda y la exploración de la relación externa una búsqueda por bloque, dando un total de $2 * b_r$ búsquedas. Evidentemente, será más eficiente utilizar la relación más pequeña como la relación externa, en caso de que ninguna de ellas quepa en memoria. En el mejor de los casos, cuando la relación interna quepa en memoria, habrá que realizar $b_r + b_s$ transferencias de bloques y sólo dos búsquedas (en este caso se escogería la relación de menor tamaño como relación interna).

En el ejemplo del cálculo de *impositor* $\bowtie$ *cliente* se usará ahora el algoritmo de reunión en bucle anidado por bloques. En el peor de los casos hay que leer una vez cada bloque de *cliente* por cada bloque de *impositor*. Así, en el peor caso, son necesarias un total de $100 * 400 + 100 = 40.100$ transferencias de bloques más $2 * 100 = 200$ búsquedas. Este coste supone una mejora importante frente a las $5.000 * 400 + 100 = 2.000.100$ transferencias de bloques más 5.100 búsquedas necesarias en el peor de los casos para la reunión en bucle anidado básica. El coste en el mejor caso sigue siendo el mismo (es decir, $100 + 400 = 500$ transferencias de bloques y dos búsquedas).

El rendimiento de los procedimientos de bucle anidado y bucle anidado por bloques se puede mejorar aún más:

- Si los atributos de la reunión en una reunión natural o en una equirreunión forman una clave de la relación interna, entonces el bucle interno puede finalizar tan pronto como se encuentre la primera correspondencia.
- En el algoritmo en bucle anidado por bloques, en lugar de utilizar bloques de disco como la unidad de bloqueo de la relación externa, se puede utilizar el mayor tamaño que quepa en memoria, mientras quede suficiente espacio para las memorias intermedias de la relación interna y la salida. En otras palabras, si la memoria tiene M bloques, se leen $M - 2$ bloques de la relación externa de una vez, y cuando se lee cada bloque de la relación interna se reúne con los $M - 2$ bloques de la relación externa. Este cambio reduce el número de exploraciones de la relación interna de b_r a $\lceil b_r/(M-2) \rceil$, donde b_r es el número de bloques de la relación externa. El coste total es $\lceil b_r/(M-2) \rceil * b_s + b_r$ transferencias de bloques y $2\lceil b_r/(M-2) \rceil$ búsquedas.
- Se puede explorar el bucle interno alternativamente hacia adelante y hacia atrás. Este método de búsqueda ordena las peticiones de bloques de disco de tal manera que los datos restantes en la memoria intermedia de la búsqueda anterior se reutilizan, reduciendo de este modo el número de accesos a disco necesarios.
- Si se dispone de un índice en un atributo de la reunión del bucle interno se pueden sustituir las búsquedas en archivos por búsquedas más eficientes en el índice. Esta optimización se describe en el Apartado 13.5.3.

13.5.3 Reunión en bucle anidado indexada

En una reunión en bucle anidado (Figura 13.4), si se dispone de un índice sobre el atributo de la reunión del bucle interno, se pueden sustituir las exploraciones de archivo por búsquedas en el índice. Para cada

tupla t_r de la relación externa *r*, se utiliza el índice para buscar tuplas en s que cumplan la condición de reunión con la tupla t_r.

Este método de reunión se denomina **reunión en bucle anidado indexada**; se puede utilizar cuando existen índices y cuando se crean índices temporales con el único propósito de evaluar la reunión.

La búsqueda de tuplas en s que cumplan las condiciones de la reunión con una tupla dada t_r es esencialmente una selección en s. Por ejemplo, considérese *impositor* $\bowtie$ *cliente*. Supóngase que se tiene una tupla de *impositor* con *nombre_cliente* "Martín". Entonces, las tuplas relevantes de s son aquellas que satisfacen la selección "*nombre_cliente* = Martín."

El coste de la reunión en bucle anidado indexada se puede calcular como se indica a continuación. Para cada tupla en la relación externa r se realiza una búsqueda en el índice para s recuperando las tuplas apropiadas. En el peor de los casos sólo hay espacio en la memoria intermedia para una página de r y una página del índice. Por tanto, son necesarios b_r operaciones de E/S para leer la relación r, donde b_r denota el número de bloques que contienen registros de r; cada E/S requiere una búsqueda y una transferencia de bloque ya que la cabeza del disco se puede haber trasladado entre cada E/S. Para cada tupla de r, se realiza una búsqueda en el índice para s. Por tanto, el coste temporal de la reunión se puede calcular como $b_r(t_T + t_S) + n_r * c$, donde n_r es el número de registros de la relación r y c es el coste de una única selección en s utilizando la condición de la reunión. Ya se vio en el Apartado 13.3 cómo estimar el coste del algoritmo de una única selección (posiblemente utilizando índices) cuyo cálculo proporciona el valor de c.

La fórmula del coste indica que, si hay índices disponibles en ambas relaciones r y s, normalmente es más eficiente usar como relación más externa aquélla que tenga menos tuplas.

Por ejemplo, considérese una reunión en bucle anidado indexada de *impositor* $\bowtie$ *cliente*, con *impositor* como relación externa. Supóngase también que *cliente* tiene un índice de árbol B^+ primario en el atributo de la reunión *nombre_cliente*, que contiene 20 entradas en promedio por cada nodo del índice. Dado que *cliente* tiene 10.000 tuplas, la altura del árbol es 4, y será necesario un acceso más para encontrar el dato real. Como $n_{impositor}$ es 5.000, el coste total es $100 + 5000 * 5 = 25.100$ accesos a disco, cada uno de los cuales requiere una búsqueda y una transferencia de bloque. En cambio, como se indicó anteriormente, se necesitan 40.100 transferencias de bloque más 200 búsquedas para una reunión en bucle anidado por bloques. Aunque se ha reducido el número de transferencias de bloques, el coste de búsqueda se ha incrementado, aumentando el coste total dado que una búsqueda es considerablemente más costosa que una transferencia de bloque. Sin embargo, si se tuviese una selección sobre la relación *impositor* que redujera significativamente el número de filas, la reunión en bucle anidado sería significativamente más rápida que una reunión en bucle anidado por bloques.

13.5.4 Reunión por mezcla

El algoritmo de **reunión por mezcla** (también llamado algoritmo de **reunión por ordenación-mezcla**) se puede utilizar para calcular reuniones naturales y equirreuniones. Sean $r(R)$ y $s(S)$ las relaciones que vamos a utilizar para realizar la reunión natural, y sean $R \cap S$ sus atributos en común. Supóngase que ambas relaciones están ordenadas según los atributos de $R \cap S$. Por tanto, su reunión se puede calcular mediante un proceso muy parecido a la etapa de mezcla del algoritmo de ordenación–mezcla.

El algoritmo de reunión por mezcla se muestra en la Figura 13.6. En este algoritmo, *AtribsReunión* se refiere a los atributos de $R \cap S$ y, a su vez, $t_r \bowtie t_s$, donde t_r y t_s son las tuplas que tienen los mismos valores en *AtribsReunión*, y denota la concatenación de los atributos de las tuplas, seguido de una proyección para no incluir los atributos repetidos. El algoritmo de reunión por mezcla asocia un puntero con cada relación. Al comienzo, estos punteros apuntan a la primera tupla de sus respectivas relaciones. Según avanza el algoritmo, el puntero se mueve a través de la relación. De este modo se leen en S_s un grupo de tuplas de una relación con el mismo valor en los atributos de la reunión. El algoritmo de la Figura 13.6 *necesita* que cada conjunto de tuplas S_s quepa en memoria principal; más adelante, en este apartado, se examinarán extensiones del algoritmo que evitan este supuesto. Después, las tuplas correspondientes de la otra relación (si las hay) se leen y se procesan según se están leyendo.

En la Figura 13.7 se muestran dos relaciones que están ordenadas en su atributo de la reunión $a1$. Es instructivo examinar los pasos del algoritmo de reunión por mezcla con las relaciones que se muestran en la figura.

Dado que las relaciones están ordenadas, las tuplas con el mismo valor en los atributos de la reunión aparecerán consecutivamente. De este modo solamente es necesario leer ordenadamente cada tupla una vez y, como resultado, leer también cada bloque solamente una vez. Puesto que sólo se recorre un ciclo en ambos archivos, el método de reunión por mezcla resulta eficiente; el número de transferencias de bloques es igual a la suma de los bloques en los dos archivos, $b_r + b_s$.

Asumiendo que se asignen b_b bloques de memoria intermedia para cada relación, el número de búsquedas requeridas sería $\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil$. Puesto que las búsquedas son mucho más costosas que las transferencias de bloques, tiene sentido asignar varios bloques de la memoria intermedia para cada relación, siempre que haya memoria disponible. Por ejemplo con $t_T = 0.1$ milisegundos por bloque de 4 megabytes y $t_S = 4$ milisegundos, el tamaño de la memoria intermedia es de 400 bloques (o $1,6$ megabytes), el tiempo de búsqueda sería 4 milisegundos por cada 40 milisegundos de tiempo de transferencia, en otras palabras, el tiempo de búsqueda sería sólo el 10 por ciento del tiempo de transferencia.

Si alguna de las relaciones de entrada r o s no está ordenada según los atributos de la reunión, se pueden ordenar primero y luego utilizar el algoritmo de reunión por mezcla. El algoritmo de reunión por mezcla también se puede extender fácilmente desde las reuniones naturales al caso más general de las equirreuniones.

```
pr := dirección de la primera tupla de r;
ps := dirección de la primera tupla de s;
while (ps ≠ null and pr ≠ null) do
    begin
        t_s := tupla a la que apunta ps;
        S_s := {t_s};
        hacer que ps apunte a la siguiente tupla de s;
        hecho := falso;
        while (not hecho and ps ≠ null) do
            begin
                t_s' := tupla a la que apunta ps;
                if (t_s'[AtribsReunión] = t_s[AtribsReunión])
                    then begin
                            S_s := S_s ∪ {t_s'};
                            hacer que ps apunte a la siguiente tupla de s;
                         end
                    else hecho := cierto;
            end
        t_r := tupla a la que apunta pr;
        while (pr ≠ null and t_r[AtribsReunión] < t_s[AtribsReunión]) do
            begin
                hacer que pr apunte a la siguiente tupla de r;
                t_r := tupla a la que apunta pr;
            end
        while (pr ≠ null and t_r[AtribsReunión] = t_s[AtribsReunión]) do
            begin
                for each t_s in S_s do
                    begin
                        añadir t_s ⋈ t_r al resultado;
                    end
                hacer que pr apunte a la siguiente tupla de r;
                t_r := tupla a la que apunta pr;
            end
    end.
```

Figura 13.6 Reunión por mezcla.

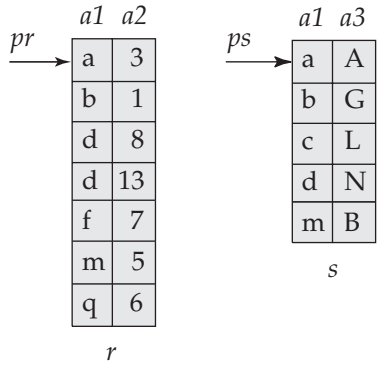


Figura 13.7 Relaciones ordenadas para la reunión por mezcla.

Como se mencionó anteriormente, el algoritmo de reunión por mezcla de la Figura 13.6 requiere que el conjunto S_s de todas las tuplas con el mismo valor de los atributos de la reunión deben caber en memoria principal. Este requisito se suele cumplir, incluso cuando la relación s es grande. Si no se puede cumplir, se debería realizar una reunión en bucle anidado por bloques entre S_s y las tuplas de r con los mismos valores de los atributos de reunión. Por tanto, se incrementa el coste total de la reunión por mezcla.

También es posible realizar una variación de la operación reunión por mezcla en tuplas desordenadas si existen índices secundarios en los dos atributos de la reunión. Así, se examinan los registros a través de los índices recuperándolos de manera ordenada. Sin embargo, esta variación presenta un importante inconveniente, puesto que los registros podrían estar diseminados a través de los bloques del archivo. Por tanto, cada acceso a una tupla podría implicar acceder a un bloque del disco, y esto es muy costoso.

Para evitar este coste se podría utilizar una técnica de reunión por mezcla híbrida que combinase índices con reunión por mezcla. Supóngase que una de las relaciones se encuentra ordenada y la otra desordenada, pero con un índice secundario de árbol B^+ en los atributos de la reunión. El **algoritmo híbrido de reunión por mezcla** fusiona la relación ordenada con las entradas hoja del índice secundario de árbol B^+. El archivo resultante contiene tuplas de la relación ordenada y direcciones de las tuplas de la relación desordenada. Después se ordena el archivo resultante según las direcciones de las tuplas de la relación desordenada, permitiendo así una recuperación eficiente de las correspondientes tuplas, según el orden físico de almacenamiento, para completar la reunión. Se deja como ejercicio la extensión de esta técnica para tratar el caso de dos relaciones desordenadas.

Supóngase que se aplica el esquema de reunión por mezcla al ejemplo de *impositor* $\bowtie$ *cliente*. En este caso, el atributo de la reunión es *nombre_cliente*. Supóngase además que las dos relaciones se encuentren ya ordenadas según el atributo de la reunión *nombre_cliente*. En este caso, la reunión por mezcla emplea un total de $400 + 100 = 500$ transferencias de bloques. Si se asume que en el peor caso sólo se asigna un bloque de memoria intermedia para cada relación de entrada (es decir, $b_b = 1$), se necesitarían un total de $400 + 100 = 500$ búsquedas; en realidad se puede hacer que b_b sea mucho mayor ya que se necesitan bloques de la memoria intermedia sólo para dos relaciones, y el coste de búsqueda sería significativamente menor.

Supóngase que las relaciones no están ordenadas y que el tamaño de memoria en el peor caso es de sólo tres bloques. El coste en este caso se calcularía:

1. Usando la fórmula desarrollada en el Apartado 13.4, la ordenación de la relación *cliente* hacen falta $\lceil \log_{3-1}(400/3) \rceil = 8$ pasos de mezcla. La relación de esta relación necesitaría $400 * (2\lceil \log_{3-1}(400/3) \rceil + 1)$, o 6.800 transferencias de bloques, con otras 400 transferencias para escribir el resultado. El número de búsquedas necesarias es $2*\lceil 400/3 \rceil + 400*(2*8-1)$, 6.268 búsquedas para ordenar y 400 búsquedas para escribir el resultado, con un total de 6.668 búsquedas dado que sólo hay disponible un bloque de memoria intermedia para cada ciclo.

2. Análogamente, la ordenación de *impositor* necesita $\lceil \log_{3-1}(100/3) \rceil = 6$ pasos de mezcla y $100 * (2\lceil \log_{3-1}(100/3) \rceil + 1)$, 1.300 transferencias de bloques, con otras 100 transferencias para escribir

el resultado. El número de búsquedas necesarias es $2*\lceil 100/3\rceil + 100*(2*6-1) = 1.164$ búsquedas para ordenar, y 100 búsquedas para escribir el resultado, con un total de 1.264 búsquedas.

3. Finalmente, la mezcla de ambas relaciones necesita $400 + 100 = 500$ transferencias de bloques y 500 búsquedas.

Por tanto, el coste total es de 9.100 transferencias de bloques más 8.932 búsquedas si las relaciones no están ordenadas y el tamaño de memoria es sólo de 3 bloques.

Con un tamaño de memoria de 25 bloques y sin tener ordenadas las relaciones, el coste de la ordenación seguida de la reunión por mezcla sería:

1. La ordenación de *cliente* se puede hacer en un único paso de mezcla y requiere un total de $400 * (2\lceil \log_{24}(400/25)\rceil + 1) = 1.200$ transferencias de bloques. De forma similar, la ordenación de *impositor* emplea 300 transferencias de bloques. Escribir la salida ordenada a disco requiere $400 + 100 = 500$ transferencias de bloques, y el paso de mezcla necesita 500 transferencias de bloques para volver a leer los datos. Al sumar estos costes se calcula un total de 2.500 transferencias de bloques.

2. Si se asume que sólo se asigna un bloque de memoria intermedia para cada secuencia, el número de búsquedas requeridas en este caso es $2 * \lceil 400/25\rceil + 400 + 400 = 832$ búsquedas para ordenar *cliente* y escribir la salida ordenada a disco y, análogamente, $2 * \lceil 100/25\rceil + 100 + 100 = 208$ para *impositor*, más $400 + 100$ búsquedas para la lectura de los datos ordenados en el paso de reunión por mezcla. Al sumar estos costes se calcula un total de 1.640 búsquedas.

   Se puede reducir significativamente el número de búsquedas asignando más bloques de memoria intermedia en cada ciclo. Por ejemplo, si se asignan cinco bloques y para la salida de la de la mezcla de las cuatro secuencias de *impositor*, el coste se reduce de 208 a $2*\lceil 100/25\rceil + \lceil 100/5\rceil + \lceil 100/5\rceil = 48$ búsquedas. Si en el paso de reunión por mezcla se asignan 12 bloques para ambas relaciones, el número de búsquedas para este paso se reduce de 500 a $\lceil 400/12\rceil + \lceil 100/12\rceil = 43$ búsquedas. Por tanto, el número total de búsquedas es 251.

Por tanto, el coste total es de 2.500 transferencias de bloques más 251 búsquedas si las relaciones no están ordenadas y el tamaño de memoria es de 25 bloques.

13.5.5 Reunión por asociación

Al igual que el algoritmo de reunión por mezcla, el algoritmo de reunión por asociación se puede utilizar para implementar reuniones naturales y equirreuniones. En el algoritmo de reunión por asociación se utiliza una función de asociación h para dividir las tuplas de ambas relaciones. La idea fundamental es dividir las tuplas de cada relación en conjuntos con el mismo valor de la función de asociación en los atributos de la reunión.

Se da por supuesto que

- h es una función de asociación que asigna a los *AtribsReunión* los valores $\{0, 1, \ldots, n_h\}$, donde los *AtribsReunión* denotan los atributos comunes de r y s utilizados en la reunión natural.
- H_{r_0}, $H_{r_1}, \ldots, H_{r_{n_h}}$ denotan las particiones de las tuplas de r, inicialmente todas vacías. Cada tupla $t_r \in r$ se pone en la partición H_{r_i}, donde $i = h(t_r[\textit{AtribsReunión}])$.
- H_{s_0}, H_{s_1}, ..., $H_{s_{n_h}}$ denotan las particiones de las tuplas de s, inicialmente todas vacías. Cada tupla $t_s \in s$ se pone en la partición H_{s_i}, donde $i = h(t_s[\textit{AtribsReunión}])$.

La función de asociación h debería de tener las "buenas" propiedades de aleatoriedad y uniformidad que se estudiaron en el Capítulo 12. La división de las relaciones se muestra de manera gráfica en la Figura 13.8.

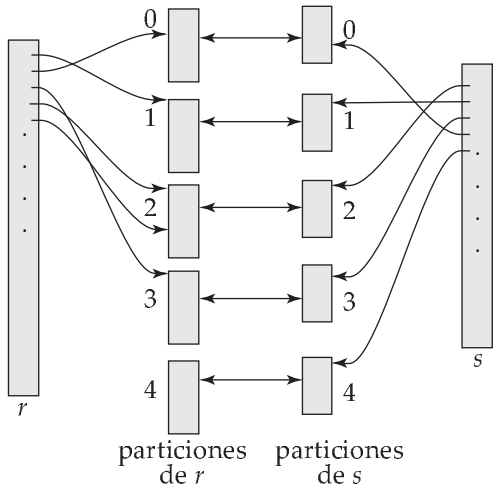


Figura 13.8 División por asociación de relaciones.

13.5.5.1 Fundamentos

La idea en que se basa el algoritmo de reunión por asociación es la siguiente. Supóngase que una tupla de r y una tupla de s satisfacen la condición de la reunión; por tanto, tendrán el mismo valor en los atributos de la reunión. Si el valor se asocia con algún valor i, la tupla de r tiene que estar en H_{r_i} y la tupla de s en H_{s_i}. De este modo solamente es necesario comparar las tuplas de r en H_{r_i} con las tuplas de s en H_{s_i}; no es necesario compararlas con las tuplas de s de ninguna otra partición.

Por ejemplo, si *i* es una tupla de *impositor*, *c* una tupla de *cliente* y h una función de asociación en los atributos *nombre_cliente* de las tuplas, solamente hay que comprobar *i* y *c* si $h(c) = h(i)$. Si $h(c) \neq h(d)$, entonces *c* e *i* poseen valores distintos de *nombre_cliente*. Sin embargo, si $h(c) = h(i)$ hay que comprobar que *c* e *i* tengan los mismos valores en los atributos de la reunión, ya que es posible que *c* e *i* tengan valores diferentes de *nombre_cliente*, con el mismo valor de la función de asociación.

En la Figura 13.9 se muestran los detalles del algoritmo de **reunión por asociación** para calcular la reunión natural de las relaciones r y s. Como en el algoritmo de reunión por mezcla, $t_r \bowtie t_s$ denota la concatenación de los atributos de las tuplas de t_r y t_s, seguido de la proyección para eliminar los atributos repetidos. Después de la división de las relaciones, el resto del código de reunión por asociación realiza una reunión en bucle anidado indexada separada en cada una de los partición pares i, con $i = 0, \ldots, n_h$. Para lograr esto, primero se **construye** un índice asociativo en cada H_{s_i} y luego se **prueba** (es decir, se busca en H_{s_i}) con las tuplas de H_{r_i}. La relación s es la **entrada de construcción** y r es la **entrada de prueba**.

El índice asociativo en H_{s_i} se crea en la memoria, así que no es necesario acceder al disco para recuperar las tuplas. La función de asociación utilizada para construir este índice asociativo debe ser distinta de la función de asociación h utilizada anteriormente, pero aún se aplica exclusivamente a los atributos de la reunión. A lo largo de la reunión en bucle anidado indexada, el sistema utiliza este índice asociativo para recuperar los registros que concuerden con los registro de la entrada de prueba.

Las etapas de construcción y prueba necesitan solamente un único ciclo a través de las entradas de construcción y prueba. La extensión del algoritmo de reunión por asociación para calcular equirreuniones más generales es directa.

Se tiene que elegir el valor de n_h lo bastante grande como para que, para cada i, las tuplas de la partición H_{s_i} de la relación de construcción, junto con el índice asociativo de la partición, quepan en memoria. Pero no tienen por qué caber en memoria las particiones de la relación que se prueban. Sería obviamente mejor utilizar la relación de entrada más pequeña como la relación de construcción. Si el tamaño de la relación de construcción es de b_s bloques, entonces para que cada una de las n_h particiones tengan un tamaño menor o igual que M, n_h debe ser al menos $\lceil b_s/M \rceil$. Con más exactitud, hay que tener en cuenta también el espacio adicional ocupado por el índice asociativo de la partición, incrementando

```
/* División de s */
for each tupla t_s in s do begin
   i := h(t_s[AtribsReunión]);
   H_{s_i} := H_{s_i} ∪ {t_s};
end
/* División de r */
for each tupla t_r in r do begin
   i := h(t_r[AtribsReunión]);
   H_{r_i} := H_{r_i} ∪ {t_r};
end
/* Realizar la reunión de cada partición */
for i := 0 to n_h do begin
   leer H_{s_i} y construir un índice asociativo en memoria en él
   for each tupla t_r in H_{r_i} do begin
      explorar el índice asociativo en H_{s_i} para localizar todas las tuplas t_s
         tales que t_s[AtribsReunión] = t_r[AtribsReunión]
      for each tupla t_s que concuerde in H_{s_i} do begin
         añadir t_r ⋈ t_s al resultado
      end
   end
end
```

Figura 13.9 Reunión por asociación.

n_h según corresponda. Por simplificar, en estos análisis se ignoran muchas veces los requisitos de espacio del índice asociativo.

13.5.5.2 División recursiva

Si el valor de n_h es mayor o igual que el número de marcos de página de la memoria, la división de las relaciones no se puede hacer en un solo ciclo, puesto que no habría suficientes páginas para memorias intermedias. En lugar de eso, la división se tiene que hacer mediante la repetición de ciclos. En un ciclo se puede dividir la entrada en tantas particiones como marcos de página haya disponibles para utilizarlos como memorias intermedias de salida. Cada cajón generado por un ciclo se lee de manera separada y se divide de nuevo en el siguiente ciclo para crear particiones más pequeñas. La función de asociación utilizada en un ciclo es, por supuesto, diferente de la que se ha utilizado en el ciclo anterior. Se repite esta división de la entrada hasta que cada partición de la entrada de construcción quepa en memoria. Esta división se denomina **división recursiva**.

Una relación no necesita de la división recursiva si $M > n_h + 1$ o, lo que es equivalente, $M > (b_s/M) + 1$, lo cual se simplifica (de manera aproximada) a $M > \sqrt{b_s}$. Por ejemplo, considerando un tamaño de la memoria de 12 megabytes, dividido en bloques de 4 megabytes, la relación contendría un total de 3K (3.072) bloques. Se puede utilizar una memoria de este tamaño para dividir relaciones de 3K $*$ 3K bloques, que son 36 gigabytes. Del mismo modo, una relación del tamaño de 1 gigabyte necesita $\sqrt{256\text{K}}$ bloques, o 2 megabytes, para evitar la división recursiva.

13.5.5.3 Gestión de desbordamientos

Se produce el **desbordamiento de una tabla de asociación** en la partición i de la relación de construcción s, si el índice asociativo de H_{s_i} es mayor que la memoria principal. El desbordamiento de la tabla de asociación puede ocurrir si existen muchas tuplas en la relación de construcción con los mismos valores en los atributos de la reunión, o si la función de asociación no tiene las propiedades de aleatoriedad y uniformidad. En cualquier caso, algunas de las particiones tendrán más tuplas que la media, mientras que otras tendrán menos; se dice entonces que la división está **sesgada**.

Se puede controlar parcialmente el sesgo mediante el incremento del número de particiones, de tal manera que el tamaño esperado de cada partición (incluído el índice asociativo en la partición) sea algo

menor que el tamaño de la memoria. Por consiguiente, el número de particiones se incrementa en un pequeño valor llamado **factor de escape**, que normalmente es del orden del 20 por ciento del número de particiones calculadas, como se describe en el Apartado 13.5.5.

Aunque se sea conservador con los tamaños de las particiones utilizando un factor de escape, todavía pueden ocurrir desbordamientos. Los desbordamientos de la tabla de asociación se pueden tratar mediante *resolución del desbordamiento* o *evitación del desbordamiento*. La **resolución del desbordamiento** se realiza como sigue. Si H_{s_i}, para cualquier i, resulta ser demasiado grande, se divide de nuevo en particiones más pequeñas utilizando una función de asociación distinta. Del mismo modo, también se divide H_{r_i} utilizando la nueva función de asociación, y solamente es necesario reunir las tuplas en las particiones concordantes.

En cambio, la **evitación del desbordamiento** realiza la división más cuidadosamente, de tal manera que el desbordamiento nunca se produce en la fase de construcción. Para evitar el desbordamiento se implementa la división de la relación de construcción s en muchas particiones pequeñas inicialmente, para luego combinar algunas de estas particiones de tal manera que cada partición combinada quepa en memoria. Además, la relación de prueba r se tiene que combinar de la misma manera que se combinan las particiones de s, sin importar los tamaños de H_{r_i}.

Las técnicas de resolución y evitación podrían fallar en algunas particiones, si un gran número de las tuplas en s tuvieran el mismo valor en los atributos de la reunión. En ese caso, en lugar de crear un índice asociativo en memoria y utilizar una reunión en bucle anidado para reunir las particiones, se pueden utilizar otras técnicas de reunión, tales como la reunión en bucle anidado por bloques, en esas particiones.

13.5.5.4 Coste de la reunión por asociación

Se considera ahora el coste de una reunión por asociación. El análisis supone que no hay desbordamiento de la tabla de asociación. Primero se considera el caso en el que no es necesaria una división recursiva.

- La división de las dos relaciones r y s reclama una lectura completa de ambas así como su posterior escritura. Este operación necesita $2(b_r + b_s)$ transferencias de bloques, donde b_r y b_s denotan respectivamente el número de bloques que contienen registros de las relaciones r y s. Las fases de construcción y prueba leen cada una de las particiones una vez, empleando $b_r + b_s$ transferencias adicionales. El número de bloques ocupados por las particiones podría ser ligeramente mayor que $b_r + b_s$ debido a que los bloques están parcialmente ocupados. El acceso a estos bloques llenos en parte puede añadir un gasto adicional de $2n_h$ a lo sumo, ya que cada una de las n_h particiones podría tener un bloque lleno parcialmente que haya que escribir y leer de nuevo. Así, el coste estimado para una reunión por asociación es

  $$3(b_r + b_s) + 4n_h$$

  transferencias de bloques. La sobrecarga $4n_h$ es muy pequeña comparada con $b_r + b_s$ y se puede ignorar.

- Si se da por supuesto que se asignan b_b bloques para las memorias intermedias de entrada y salida, la división necesita $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)$ búsquedas. Las fases de construcción y prueba requieren sólo una búsqueda para cada una de las n_h particiones de cada relación, ya que cada partición se puede leer secuencialmente. Por tanto, la reunión por asociación necesita $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil) + 2n_h$ búsquedas.

Considérese ahora el caso en el que es necesario realizar la división recursiva. Cada ciclo reduce el tamaño de cada una de las particiones por un factor esperado de $M - 1$; y los ciclos se repiten hasta que cada partición tenga como mucho un tamaño de M bloques. Por tanto, el número esperado de ciclos necesarios para dividir s es $\lceil \log_{M-1}(b_s) - 1 \rceil$.

- Como todos los bloques de s se leen y se escriben en cada ciclo, el número total de transferencias de bloques para dividir s es $2b_s\lceil \log_{M-1}(b_s) - 1 \rceil$. Como el número de ciclos para dividir r es el

mismo que el número de ciclos para dividir s, por tanto la estimación del coste de la reunión es

$$2(b_r + b_s)\lceil \log_{M-1}(b_s) - 1 \rceil + b_r + b_s$$

transferencias de bloques.

- De nuevo, dando por supuesto que se asignen b_b bloques en memoria intermedia para cada partición, e ignorando el número relativamente pequeño de búsquedas durante las fases de construcción y prueba, la reunión por asociación con división recursiva necesita

$$2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)\lceil \log_{M-1}(b_s) - 1 \rceil$$

búsquedas.

Considérese, por ejemplo, la reunión *cliente* $\bowtie$ *impositor*. Con un tamaño de memoria de 20 bloques, se puede dividir *impositor* en cinco partes, cada una de 20 bloques, con un tamaño tal que caben en memoria. Así, sólo es necesario un ciclo para la división. De la misma manera la relación *cliente* se divide en cinco particiones, cada una de 80 bloques. Si se ignora el coste de escribir los bloque parcialmente llenos, el coste es $3(100 + 400) = 1.500$ transferencias de bloques. Hay suficiente memoria para asignar tres bloques a la entrada y cada uno de los cinco bloques de salida durante la división, lo que resulta en $2(\lceil 100/3 \rceil + \lceil 400/3 \rceil) = 336$ búsquedas.

Es posible mejorar la reunión por asociación si el tamaño de la memoria principal es grande. Cuando la entrada de construcción se puede guardar por completo en memoria principal hay que establecer n_h a 0; de este modo el algoritmo de reunión por asociación se ejecuta rápidamente, sin dividir las relaciones en archivos temporales y sin importar el tamaño de la entrada de prueba. El coste estimado desciende a $b_r + b_s$ transferencias de bloques y dos búsquedas.

13.5.5.5 Reunión por asociación híbrida

El algoritmo de **reunión por asociación híbrida** realiza otra optimización; es útil cuando los tamaños de la memoria son relativamente grandes pero no cabe toda la relación de construcción en memoria. La fase de división del algoritmo de reunión por asociación necesita un bloque de memoria como memoria intermedia para cada partición que se cree, más un bloque de memoria como memoria intermedia de entrada. Por tanto, se necesitan $n_h + 1$ bloques de memoria para dividir las dos relaciones. Si la memoria es mayor que $n_h + 1$, se puede utilizar el resto de la memoria ($M - n_h - 1$ bloques) para guardar en memorias intermedias la primera partición de la entrada de construcción (esto es, H_{s_0}), así que no es necesario escribirla ni leerla de nuevo. Más aún, la función de asociación se diseña de tal manera que el índice asociativo en H_{s_0} quepa en $M - n_h - 1$ bloques, así que, al final de la división de s, H_{s_0} está en memoria por completo y se puede construir un índice asociativo en H_{s_0}.

Cuando r se divide de nuevo, las tuplas en H_{r_0} no se escriben en disco; en su lugar, según se van generando, el sistema las utiliza para examinar el índice asociativo residente en memoria de H_{s_0} y para generar las tuplas de salida de la reunión. Después de utilizarlas para la prueba, se descartan las tuplas, así que la partición H_{r_0} no ocupa ningún espacio en memoria. De este modo se ahorra un acceso de lectura y otro de escritura para cada bloque de H_{r_0} y H_{s_0}. Las tuplas en otras particiones se escriben de la manera usual para reunirlas más tarde. El ahorro de la reunión por asociación híbrida puede ser importante si la entrada de construcción es ligeramente mayor que la memoria.

Si el tamaño de la relación de construcción es b_s, n_h es aproximadamente igual a b_s/M. Así, la reunión por asociación híbrida es más útil si $M >> b_s/M$ o si $M >> \sqrt{b_s}$, donde la notación $>>$ significa *mucho mayor que.* Por ejemplo, supóngase que el tamaño del bloque es de 4 kilobytes y que el tamaño de la relación de construcción es de 1 gigabyte. Entonces, el algoritmo híbrido de reunión por asociación es útil si el tamaño de la memoria es claramente mayor que 2 megabytes; las memorias de 100 megabytes o más son comunes en las computadoras de hoy en día.

Considérese de nuevo la reunión *cliente* $\bowtie$ *impositor*. Con una memoria de 25 bloques de tamaño, se puede dividir *impositor* en cinco particiones, cada una de 20 bloques, y con la primera de las particiones de la relación de construcción almacenada en memoria. Ocupa 20 bloques de memoria; un bloque se utiliza para la entrada y cuatro bloques más para guardar en memorias intermedias cada partición. De manera similar se divide la relación *cliente* en cinco particiones cada una de tamaño 80, la primera de

las cuales de utiliza precisamente para comprobar, en lugar de escribirse y leerse de nuevo. Ignorando el coste de escribir los bloques parcialmente llenos, el coste es de $3(80 + 320) + 20 + 80 = 1.300$ bloques transferidos, en lugar de las 1.500 transferencias de bloques sin la optimización por asociación híbrida. Sin embargo, en este caso el tamaño de la memoria intermedia para la entrada y para cada partición escrita a disco desciende a un bloque, aumentando el número de búsquedas a alrededor del número de accesos a bloques, y aumentando también el coste total. Por tanto, en este caso es mejor usar una reunión por asociación con una memoria intermedia mayor de b_b en lugar de usar la reunión por asociación híbrida. Sin embargo, con tamaños mucho mayores de memoria, el aumento de b_b más allá de un punto determinado disminuye el rendimiento, y la memoria restante se puede usar para implementar la reunión por asociación híbrida.

13.5.6 Reuniones complejas

Las reuniones en bucle anidado y en bucle anidado por bloques son útiles sean cuales sean las condiciones de la reunión. Las otras técnicas de reunión son más eficientes que las reuniones en bucle anidado y sus variantes, aunque sólo se pueden utilizar con condiciones simples, tales como las reuniones naturales o las equirreuniones. Se pueden implementar reuniones con condiciones de la reunión más complejas, tales como conjunciones y disyunciones, utilizando las técnicas eficientes de la reunión mediante la aplicación de las técnicas desarrolladas en el Apartado 13.3.4 para el manejo de selecciones complejas.

Considérese la siguiente reunión con una condición conjuntiva:

$$r \bowtie_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n} s$$

Se pueden aplicar una o más de las técnicas de reunión descritas anteriormente en cada condición individual $r \bowtie_{\theta_1} s$, $r \bowtie_{\theta_2} s$, $r \bowtie_{\theta_3} s$, y así sucesivamente. El resultado total de la reunión se determina calculando primero el resultado de una de estas reuniones simples $r \bowtie_{\theta_i} s$; cada par de tuplas del resultado intermedio se compone de una tupla de r y otra de s. El resultado total de la reunión consiste en las tuplas del resultado intermedio que satisfacen el resto de condiciones

$$\theta_1 \wedge \cdots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \cdots \wedge \theta_n$$

Estas condiciones se pueden ir comprobando según se generan las tuplas de $r \bowtie_{\theta_i} s$.

Una reunión cuya condición es una disyunción se puede calcular como se indica a continuación. Considérese

$$r \bowtie_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n} s$$

La reunión se puede calcular como la unión de los registros de las reuniones $r \bowtie_{\theta_i} s$:

$$(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \cdots \cup (r \bowtie_{\theta_n} s)$$

Los algoritmos para calcular la unión de relaciones se describen en el Apartado 13.6.

13.6 Otras operaciones

Otras operaciones relacionales y operaciones relacionales extendidas (como eliminación de duplicados, proyección, operaciones sobre conjuntos, reunión externa y agregación) se pueden implementar según se describe en los Apartados 13.6.1 al 13.6.5.

13.6.1 Eliminación de duplicados

Se puede implementar fácilmente la eliminación de duplicados utilizando la ordenación. Las tuplas idénticas aparecerán consecutivas durante la ordenación, pudiéndose eliminar todas las copias menos una. Con la ordenación–mezcla externa, se pueden eliminar los duplicados mientras se crea una secuencia antes de que ésta se escriba en el disco, reduciendo así el número de bloques transferidos. El resto de duplicados se pueden suprimir durante la etapa de reunión/mezcla, así que el resultado final está libre de repeticiones. El coste estimado en el peor de los casos para la eliminación de duplicados es el mismo que el coste estimado en el peor caso para la ordenación de una relación.

Se puede implementar también la eliminación de duplicados utilizando la asociación de una manera similar al algoritmo de reunión por asociación. Primero, se divide la relación basándose en una función

de asociación en la tupla entera. Luego, se lee cada partición, y se construye un índice asociativo en memoria. Mientras se construye el índice asociativo se inserta una tupla solamente si no estaba ya presente. En otro caso se descarta. Después de que todas las tuplas de la relación se hayan procesado, las tuplas en el índice asociativo se escriben al resultado. El coste estimado es el mismo que el coste de procesar (división y lectura de cada partición) la relación de construcción en una reunión por asociación.

Debido al coste relativamente alto de la eliminación de duplicados, SQL exige una petición explícita del usuario para suprimir los duplicados; en caso contrario, se conservan.

13.6.2 Proyección

La proyección se puede implementar fácilmente realizando la proyección de cada tupla, pudiendo originar una relación con registros repetidos y suprimiendo después los registros duplicados. La eliminación de duplicados se puede llevar a cabo según se ha descrito en el Apartado 13.6.1. Si los atributos de la lista de proyección incluyen una clave de la relación, no se producirán duplicados; por tanto no será necesario eliminarlos. La proyección generalizada (tratada en el Apartado 2.4.1) se puede implementar de la misma manera que las proyecciones.

13.6.3 Operaciones sobre conjuntos

Las operaciones *unión*, *intersección* y *diferencia de conjuntos* se pueden implementar ordenando primero ambas relaciones y examinando después cada relación para producir el resultado. En $r \cup s$, cuando una exploración concurrente en ambas relaciones descubre la misma tupla en los dos archivos, solamente se conserva una de las tuplas. Por otra parte, el resultado de $r \cap s$ contendrá únicamente aquellas tuplas que aparezcan en ambas relaciones. De la misma manera se implementa la *diferencia de conjuntos*, $r - s$, guardando aquellas tuplas de r que estén ausentes de s.

Para todas estas operaciones solamente se necesita una exploración en cada relación de entrada, así el coste es $b_r + b_s$ transferencias de bloques si las relaciones están ordenadas de igual forma. Suponiendo que en el peor caso sólo hay un bloque de memoria intermedia para cada relación, se necesitarían un total de $b_r + b_s$ búsquedas además de las $b_r + b_s$ transferencias de bloques. El número de búsquedas se puede reducir asignando bloques adicionales de memoria intermedia.

Si las relaciones no están ordenadas inicialmente, hay que incluir el coste de la ordenación. Se puede utilizar cualquier otro orden en la evaluación de la operación de conjuntos, siempre que las dos entradas tengan la misma ordenación.

La asociación proporciona otra manera de implementar estas operaciones sobre conjuntos. El primer paso en cada caso es dividir las dos relaciones utilizando la misma función de asociación y de este modo crear las particiones $H_{r_0}, H_{r_1}, \ldots, H_{r_{n_h}}$ y $H_{s_0}, H_{s_1}, \ldots, H_{s_{n_h}}$. En función de cada operación, el sistema da a continuación estos pasos en cada partición $i = 0, 1, \ldots, n_h$:

- $r \cup s$
  1. Construir un índice asociativo en memoria sobre H_{r_i}.
  2. Añadir las tuplas de H_{s_i} al índice asociativo solamente si no estaban ya presentes.
  3. Añadir las tuplas del índice asociativo al resultado.

- $r \cap s$
  1. Construir un índice asociativo en memoria en H_{r_i}.
  2. Para cada tupla en H_{s_i} probar el índice asociativo y pasar la tupla al resultado únicamente si ya estaba presente en el índice.

- $r - s$
  1. Construir un índice asociativo en memoria en H_{r_i}.
  2. Para cada tupla de H_{s_i} probar el índice asociativo y, si la tupla está presente en el índice, suprimirla del índice asociativo.
  3. Añadir las tuplas restantes del índice asociativo al resultado.

13.6.4 Reunión externa

Recordemos las *operaciones de reunión externa* que se describieron en el Apartado 2.4.3. Por ejemplo, la reunión externa por la izquierda *cliente* $\exists\bowtie$ *impositor* contiene la reunión de *cliente* y de *impositor* y además, para cada tupla t de *cliente* que no concuerde con alguna tupla en *impositor* (es decir, donde no esté *nombre_cliente* en *impositor*), se añade la siguiente tupla t_1 al resultado. Para todos los atributos del esquema de *cliente* la tupla t_1 tiene los mismos valores que la tupla t. El resto de los atributos (del esquema de *impositor*) de la tupla t_1 contienen el valor nulo.

Se pueden implementar las operaciones de reunión externa empleando una de las dos estrategias siguientes:

1. Calcular la reunión correspondiente, y luego añadir más tuplas al resultado de la reunión hasta obtener la reunión externa resultado. Considérese la operación de reunión externa por la izquierda y dos relaciones: $r(R)$ y $s(S)$. Para evaluar $r \exists\bowtie_\theta s$, se calcula primero $r \bowtie_\theta s$ y se guarda este resultado como relación temporal q_1. A continuación se calcula $r - \Pi_R(q_1)$, que produce las tuplas de r que no participaron en la reunión. Se puede utilizar cualquier algoritmo para calcular las reuniones. Luego se rellenan cada una de estas tuplas con valores nulos en los atributos de s y se añaden a q_1 para obtener el resultado de la reunión externa.

   La operación reunión externa por la derecha $r \bowtie\exists_\theta s$ es equivalente a $s \exists\bowtie_\theta r$ y, por tanto, se puede implementar de manera simétrica a la reunión externa por la izquierda. También se puede implementar la operación reunión externa completa $r \exists\bowtie\exists_\theta s$ calculando la reunión $r \bowtie s$ y añadiendo luego las tuplas adicionales de las operaciones reunión externa por la izquierda y por la derecha, al igual que antes.

2. Modificar los algoritmos de la reunión. Así, es fácil extender los algoritmos de reunión en bucle anidado para calcular la reunión externa por la izquierda. Las tuplas de la relación externa que no concuerdan con ninguna tupla de la relación interna se escriben en la salida después de haber sido completadas con valores nulos. Sin embargo, es difícil extender la reunión en bucle anidado para calcular la reunión externa completa.

   Las reuniones externas naturales y las reuniones externas con una condición de equirreunión se pueden calcular mediante extensiones de los algoritmos de reunión por mezcla y reunión por asociación. La reunión por mezcla se puede extender para realizar la reunión externa completa como se muestra a continuación. Cuando se está produciendo la mezcla de las dos relaciones, las tuplas de la relación que no encajan con ninguna tupla de la otra relación se pueden completar con valores nulos y escribirse en la salida. Del mismo modo se puede extender la reunión por mezcla para calcular las reuniones externas por la izquierda y por la derecha mediante la copia de las tuplas que no concuerden (rellenadas con valores nulos) desde solamente una de las relaciones. Puesto que las relaciones están ordenadas, es fácil detectar si una tupla coincide o no con alguna de las tuplas de la otra relación. Por ejemplo, cuando se hace una reunión por mezcla de *cliente* e *impositor*, las tuplas se leen según el orden de *nombre_cliente* y es fácil comprobar para cada tupla si hay alguna tupla coincidente.

   El coste estimado para implementar reuniones externas utilizando el algoritmo de reunión por mezcla es el mismo que para la correspondiente reunión. La única diferencia está en el tamaño del resultado y, por tanto, en los bloques transferidos para copiarlos que no se han tenido en cuenta en las estimaciones anteriores del coste.

   La extensión del algoritmo de reunión por asociación para calcular reuniones externas se deja como ejercicio (Ejercicio 13.13).

13.6.5 Agregación

Considérese el operador de agregación $\mathcal{G}$ estudiado en el Apartado 2.4.2. Por ejemplo, la operación:

$$_{nombre_sucursal}\mathcal{G}_{\mathbf{sum}(saldo)}(cuenta)$$

agrupa tuplas de *cuenta* por sucursal y calcula el saldo total de todas las cuentas de cada sucursal.

La operación agregación se puede implementar de una manera parecida a la eliminación de duplicados. Se utiliza la ordenación o la asociación, al igual que se hizo para la supresión de duplicados, pero

basándose ahora en la agrupación de atributos (*nombre_sucursal* en el ejemplo anterior). Sin embargo, en lugar de eliminar las tuplas con el mismo valor en los atributos de la agrupación, se reúnen en grupos y se aplican las operaciones agregación en cada grupo para obtener el resultado.

El coste estimado para la implementación de la operación agregación es el mismo coste de la eliminación de duplicados para las funciones de agregación como **min**, **max**, **sum**, **count** y **avg**.

En lugar de reunir todas las tuplas en grupos y aplicar entonces las funciones de agregación, se pueden implementar las funciones de agregación **sum**, **min**, **max**, **count** y **avg** sobre la marcha según se construyen los grupos. Para el caso de **sum**, **min** y **max**, cuando se encuentran dos tuplas del mismo grupo el sistema las sustituye por una sola tupla que contenga **sum**, **min** o **max**, respectivamente, de las columnas que se están agregando. Para la operación **count**, se mantiene una cuenta incremental para cada grupo con tuplas descubiertas. Por último, la operación **avg** se implementa calculando la suma y la cuenta de valores sobre la marcha, para dividir finalmente la suma entre la cuenta para obtener la media.

Si todas las tuplas del resultado caben en memoria, las implementaciones basadas en ordenación y las basadas en asociación no necesitan escribir ninguna tupla en disco. Según se leen las tuplas se pueden insertar en un estructura ordenada de árbol o en un índice asociativo. Así, cuando se utilizan las técnicas de agregación sobre la marcha, solamente es necesario almacenar una tupla para cada uno de los grupos. Por tanto, la estructura ordenada de árbol o el índice asociativo caben en memoria y se puede procesar la agregación con sólo b_r transferencias de bloques (y una búsqueda), en lugar de las $3b_r$ transferencias (y en el peor caso de hasta $2b_r$ búsquedas) que se necesitarían en el otro caso.

13.7 Evaluación de expresiones

Hasta aquí se ha estudiado cómo llevar a cabo operaciones relacionales individuales. Ahora se considera cómo evaluar una expresión que contiene varias operaciones. La manera evidente de evaluar una expresión es simplemente evaluar una operación a la vez en un orden apropiado. El resultado de cada evaluación se **materializa** en una relación temporal para su inmediata utilización. Un inconveniente de esta aproximación es la necesidad de construir relaciones temporales, que (a menos que sean pequeñas) se tienen que escribir en disco. Un enfoque alternativo es evaluar varias operaciones de manera simultánea en un **cauce**, con los resultados de una operación pasados a la siguiente, sin la necesidad de almacenar relaciones temporales.

En los Apartados 13.7.1 y 13.7.2 se consideran ambos enfoques, *materialización* y *encauzamiento*. Se verá que el coste de estos enfoques puede diferir substancialmente, pero también que existen casos donde sólo la materialización es posible.

13.7.1 Materialización

Intuitivamente es fácil de entender cómo evaluar una expresión observando una representación gráfica de la expresión en un **árbol de operadores**. Considérese la expresión

$$\Pi_{nombre_cliente}\ (\sigma_{saldo<2500}\ (cuenta) \bowtie\ cliente)$$

que se muestra en la Figura 13.10.

Si se aplica el enfoque de la materialización, se comienza por las operaciones de la expresión de nivel más bajo (al fondo del árbol). En el ejemplo solamente hay una de estas operaciones: la operación selección en *cuenta*. Las entradas de las operaciones de nivel más bajo son las relaciones de la base de datos. Se ejecutan estas operaciones utilizando los algoritmos ya estudiados, almacenando sus resultados en relaciones temporales. Luego se utilizan estas relaciones temporales para ejecutar las operaciones del siguiente nivel en el árbol, cuyas entradas son ahora o bien relaciones temporales o bien relaciones almacenadas en la base de datos. En este ejemplo, las entradas de la reunión son la relación *cliente* y la relación temporal producida por la selección en *cuenta.* Ahora se puede evaluar la reunión creando otra relación temporal.

Repitiendo este proceso se calcularía finalmente la operación en la raíz del árbol, obteniendo el resultado final de la expresión. En el ejemplo se consigue el resultado final mediante la ejecución de la operación proyección de la raíz utilizando como entrada la relación temporal creada por la reunión.

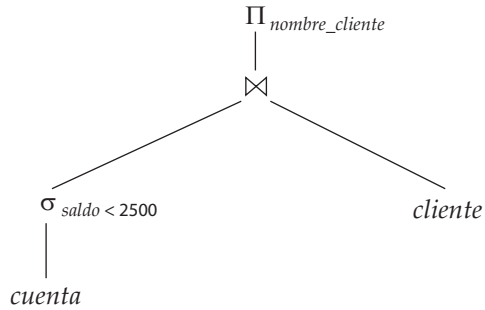


Figura 13.10 Representación gráfica de una expresión.

Una evaluación como la descrita se llama **evaluación materializada**, puesto que los resultados de cada operación intermedia se crean (materializan) para utilizarse a continuación en la evaluación de las operaciones del siguiente nivel.

El coste de una evaluación materializada no es simplemente la suma de los costes de las operaciones involucradas. Cuando se calcularon los costes estimados de los algoritmos se ignoró el coste de escribir el resultado de la operación en disco. Para calcular el coste de evaluar una expresión como la que se ha hecho hay que añadir los costes de todas las operaciones, incluyendo el coste de escribir los resultados intermedios en disco. Supóngase que los registros del resultado se acumulan en una memoria intermedia y que cuando ésta se llena, los registros se escriben en el disco. El número de bloques escritos, b_r, se puede estimar en n_r/f_r, donde n_r es el número aproximado de tuplas de la relación resultado r y f_r es el *factor de bloqueo* de la relación resultado, es decir, el número de registros de r que caben en un bloque. Además del tiempo de transferencia es posible que se necesiten algunas búsquedas, ya que la cabeza del disco se puede haber desplazado entre escrituras sucesivas. Se puede estimar el número de búsquedas como $\lceil b_r/b_b \rceil$, donde b_b es el tamaño en bloques de la memoria intermedia para la salida.

La **memoria intermedia doble** (usando dos memorias intermedias, una donde progresa la ejecución del algoritmo mientras que la otra se está copiando) permite que el algoritmo se ejecute más rápidamente mediante la ejecución en paralelo de acciones en la CPU con acciones de E/S. El número de búsquedas se puede reducir asignando bloques adicionales en la memoria intermedia de salida y escribiendo varios bloques a la vez.

13.7.2 Encauzamiento

Se puede mejorar la eficiencia de la evaluación de consultas mediante la reducción del número de archivos temporales que se producen. Se lleva a cabo esta reducción con la combinación de varias operaciones relacionales en un *encauzamiento* de operaciones, en el que se pasan los resultados de una operación a la siguiente operación del encauzamiento. Esta evaluación, como se ha descrito, se denomina **evaluación encauzada**. La combinación de operaciones en un encauzamiento elimina el coste de leer y escribir relaciones temporales.

Por ejemplo, considérese la reunión de un par de relaciones seguida de una proyección ($\Pi_{a1,a2}(r \bowtie s)$). Si se aplicara la materialización, la evaluación implicaría la creación de una relación temporal para guardar el resultado de la reunión y la posterior lectura del resultado para realizar la proyección. Estas operaciones se pueden combinar como sigue. Cuando la operación reunión genera una tupla del resultado, se pasa inmediatamente esa tupla al operador de proyección para su procesamiento. Mediante la combinación de la reunión y de la proyección, se evita la creación de resultados intermedios, creando en su lugar el resultado final directamente.

13.7.2.1 Implementación del encauzamiento

Se puede implementar el encauzamiento construyendo una única y compleja operación que combine las operaciones que constituyen el encauzamiento. Aunque este enfoque podría ser factible en muchas situaciones, es deseable en general reusar el código en operaciones individuales en la construcción del encauzamiento. Por lo tanto, cada operación del encauzamiento se modela como un proceso aislado o

una hebra en el sistema, que toma un flujo de tuplas de sus entradas encauzadas y produce un flujo de tuplas como salida. Para cada pareja de operaciones adyacentes en el encauzamiento se crea una memoria intermedia para guardar las tuplas que se envían de una operación a la siguiente.

En el ejemplo de la Figura 13.10, las tres operaciones se pueden situar en un encauzamiento, en el que los resultados de la selección se pasan a la reunión según se generan. Por su parte, los resultados de la reunión se envían a la proyección según se van generando. Así, los requisitos de memoria son bajos, ya que los resultados de una operación no se almacenan por mucho tiempo. Sin embargo, como resultado del encauzamiento, las entradas de las operaciones no están disponibles todas a la vez para su procesamiento.

Los encauzamientos se pueden ejecutar de alguno de los siguientes modos:

1. Bajo demanda.
2. Desde los productores.

En un **encauzamiento bajo demanda**, el sistema reitera peticiones de tuplas desde la operación de la cima del encauzamiento. Cada vez que una operación recibe una petición de tuplas, calcula la siguiente tupla (o tuplas) a devolver y la envía. Si las entradas de la operación no están encauzadas, la(s) siguiente(s) tupla(s) a devolver se calcula(n) de las relaciones de entrada mientras se lleva cuenta de lo que se ha remitido hasta el momento. Si alguna de sus entradas está encauzada, la operación también hace peticiones de tuplas desde sus entradas encauzadas. Así, utilizando las tuplas recibidas en sus entradas encauzadas, la operación calcula sus tuplas de salida y las envía hasta su padre.

En un **encauzamiento por los productores**, los operadores no esperan a que se produzcan peticiones para producir tuplas, en su lugar generan las tuplas **impacientemente**. Cada operación del fondo del encauzamiento genera continuamente tuplas de salida y las ponen en su memoria intermedia de salida hasta que se llena. Una operación en cualquier otro nivel del encauzamiento obtiene sus tuplas de entrada de un nivel inferior del encauzamiento hasta llenar su memoria intermedia de salida. Una vez que la operación ha utilizado una tupla de su entrada encauzada, la elimina de ella. En cualquier caso, una vez que la memoria intermedia de salida esté llena, la operación esperará hasta que su operación padre elimine las tuplas de la memoria intermedia para hacer más espacio a nuevas tuplas. En este momento, la operación genera más tuplas hasta que se llene la memoria intermedia de nuevo. Este proceso se repite por una operación hasta que se hayan generado todas las tuplas de salida.

El sistema necesita cambiar de una operación a otra solamente cuando se llena una memoria intermedia de salida o cuando una memoria intermedia de entrada está vacía y se necesitan más tuplas para generar las tuplas de salida. En un sistema de procesamiento paralelo, las operaciones del encauzamiento se pueden ejecutar concurrentemente en distintos procesadores (véase el Capítulo 21).

Se puede imaginar el encauzamiento por los productores como una **inserción** de datos de abajo hacia arriba en el árbol de operaciones, mientras que el encauzamiento bajo demanda se puede pensar como una **extracción** de datos desde la cima del árbol de operaciones. Mientras que las tuplas se generan *impacientemente* en el encauzamiento por los productores, se generan de forma **perezosa**, bajo demanda, en el encauzamiento bajo demanda.

Cada operación en un encauzamiento bajo demanda se puede implementar como un **iterador**, que proporciona las siguientes funciones: *abrir*(), *siguiente*() y *cerrar*(). Después de una llamada a *abrir*(), cada llamada a *siguiente*() devuelve la siguiente tupla de salida de la operación. La implementación de la operación realiza por turno llamadas a *abrir*() y a *siguiente*() desde sus entradas, cuando necesita tuplas de entrada. La función *cerrar*() comunica al iterador que no necesita más tuplas. De este modo, el iterador mantiene el **estado** de su ejecución entre las llamadas, de tal manera que las sucesivas peticiones *siguiente*() reciban sucesivas tuplas resultado.

Por ejemplo, en un iterador que implemente la operación selección usando la búsqueda lineal, la operación *abrir*() inicia una exploración de archivo y el estado del iterador registra el punto en el que el archivo se ha explorado. Cuando se llama a la función *siguiente*() la exploración del archivo continúa a continuación del punto anterior; cuando se encuentre la siguiente tupla que cumpla la selección al explorar el archivo, se devuelve la tupla después de almacenar el punto donde se encontró en el estado del iterador. Una operación *abrir*() del iterador de reunión por mezcla abriría sus entradas y, si aún no están ordenadas, también las ordenaría. En las llamadas a *siguiente*() se devolvería el siguiente par

de tuplas coincidentes. La información de estado consistiría en el grado en que se ha explorado cada entrada.

Los detalles de la implementación de iteradores se dejan para completar en el Ejercicio práctico 13.7. El encauzamiento bajo demanda se utiliza normalmente más que el encauzamiento por los productores, ya que es más fácil de implementar.

13.7.2.2 Algoritmos de evaluación para el encauzamiento

Considérese una operación reunión cuya entrada del lado izquierdo está encauzada. Puesto que está encauzada, no toda la entrada está disponible al mismo tiempo para el procesamiento de la operación reunión. Al no estar disponible, se limita la elección del algoritmo de reunión a emplear. Por ejemplo, no se puede usar la reunión por mezcla si las entradas no están ordenadas, puesto que no es posible ordenar la relación hasta que todas las tuplas estén disponibles (así, en efecto, se convierte el encauzamiento en materialización). Sin embargo, se puede utilizar la reunión en bucle anidado indexada: según se reciben las tuplas por el lado izquierdo de la reunión se pueden utilizar para indexar el lado derecho de la relación y para generar las tuplas resultado de la reunión. Este ejemplo ilustra que las elecciones respecto del algoritmo a utilizar para una operación y las elecciones respecto del encauzamiento no son independientes.

Las restricciones en los algoritmos de evaluación que se pueden utilizar es un factor determinante del encauzamiento. Como resultado, a pesar de las aparentes ventajas del encauzamiento, hay casos donde la materialización alcanza un coste total menor. Supóngase que se desea realizar la reunión de r y de s, y que la entrada r está encauzada. Si se utiliza una reunión en bucle anidado indexada para llevar a cabo el encauzamiento, podría ser necesario un acceso a disco para cada tupla de la relación de entrada encauzada. El coste de esta técnica es $n_r * \mathrm{AA}_i * (t_S + t_T)$, donde AA_i es la altura del índice de s. Con la materialización, el coste de copiar r sería $b_r * t_T$. Con una técnica de reunión, como la reunión por asociación, podría ser posible realizar la reunión con un coste aproximado de $3(b_r + b_s)$ transferencias de bloques más $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)$ búsquedas (asumiendo suficiente memoria para evitar la división recursiva). Para n_r grandes, la materialización sería más económica.

El uso eficiente del encauzamiento necesita la utilización de algoritmos de evaluación que puedan generar tuplas de salida según se están recibiendo tuplas por las entradas de la operación. Se pueden distinguir dos casos:

1. Solamente una de las entradas de la reunión está encauzada.
2. Las dos entradas de la reunión están encauzadas.

Si únicamente una de las entradas de la reunión está encauzada, la reunión en bucle anidado indexada es la elección más natural. Si las tuplas de entrada encauzadas están ordenadas según los atributos de la reunión y la condición de la reunión es una equirreunión, también se puede emplear la reunión por mezcla. La reunión por asociación híbrida se puede utilizar con la entrada encauzada como la relación de prueba. Sin embargo, las tuplas que no están en la primera partición se enviarán a la salida solamente después de que la relación de entrada encauzada se reciba por completo. La reunión por asociación híbrida es útil si las entradas no encauzadas caben completamente en memoria, o si al menos la mayoría de las entradas caben en memoria.

Si ambas entradas están encauzadas, la elección de los algoritmos de reunión está más limitada. Si ambas entradas están ordenadas en los atributos de la reunión y la condición de la reunión es una equirreunión, entonces se puede usar la reunión por mezcla. Otra técnica alternativa es la **reunión encauzada**, que se muestra en la Figura 13.11. El algoritmo supone que las tuplas de entrada de ambas relaciones, r y s, están encauzadas. Las tuplas disponibles de ambas relaciones se dejan listas para su procesamiento en una cola simple. Así mismo, se generan unas entradas de la cola especiales, llamadas Fin_r y Fin_s, que sirven como marcas de fin de archivo y que se insertan en la cola después de que se hayan generado todas las tuplas de r y de s (respectivamente). Para una evaluación eficaz se deberían construir los índices apropiados en las relaciones r y s. Según se añaden tuplas a r y a s se deben mantener los índices actualizados.

```
hecho_r := falso;
hecho_s := falso;
r := ∅;
s := ∅;
resultado := ∅;
while not hecho_r or not hecho_s do
    begin
        if la cola está vacía, then esperar hasta que la cola no esté vacía;
        t := entrada de la cima de la cola;
        if t = Fin_r then hecho_r := cierto
          else if t = Fin_s then hecho_s := cierto
              else if t es de la entrada r
                  then
                      begin
                          r := r ∪ {t};
                          resultado := resultado ∪ ({t} ⋈ s);
                      end
                  else /* t es de la entrada s */
                      begin
                          s := s ∪ {t};
                          resultado := resultado ∪ (r ⋈ {t});
                      end
    end
```

Figura 13.11 Algoritmo de reunión encauzada.

13.8 Resumen

- La primera acción que el sistema debe realizar en una consulta es traducirla en su formato interno, que (para sistemas de bases de datos relacionales) está basado normalmente en el álgebra relacional. En el proceso de generación del formato interno de la consulta, el analizador comprueba la sintaxis, verifica que los nombres de relación que figuran en la consulta son nombres de relaciones de la base de datos, etc. Si la consulta se expresó en términos de una vista, el analizador sustituye todas las referencias al nombre de la vista con su expresión del álgebra relacional para calcularla.

- Dada una consulta, generalmente hay diversos métodos para calcular la respuesta. Es responsabilidad del sistema transformar la consulta proporcionada por el usuario en otra consulta equivalente que se pueda ejecutar de un modo más eficiente. El Capítulo 14 estudia la optimización de consultas.

- Se pueden procesar consultas que impliquen selecciones sencillas mediante una búsqueda lineal, una búsqueda binaria o utilizando índices. Así mismo, se pueden manejar selecciones más complejas mediante uniones e intersecciones de los resultados de selecciones simples.

- Se pueden ordenar relaciones que sean más grandes que la memoria utilizando el algoritmo de ordenación–mezcla externa.

- Las consultas que impliquen una reunión natural se pueden procesar de varias maneras, dependiendo de la disponibilidad de índices y del tipo de almacenamiento físico utilizado para las relaciones.
  - Si la reunión resultante es casi tan grande como el producto cartesiano de las dos relaciones, una estrategia de *reunión en bucle anidado por bloques* podría ser ventajosa.
  - Si hay índices disponibles, se puede utilizar *la reunión en bucle anidado indexada*.

  - Si las relaciones están ordenadas, sería deseable una *reunión por mezcla*. Además, podría ser útil ordenar una relación antes que calcular una reunión (para permitir el uso de una estrategia de reunión por mezcla).
  - El algoritmo de *reunión por asociación* divide la relación en varias particiones, de tal manera que cada partición de una de las relaciones quepa en memoria. La división se lleva a cabo con una función de asociación en los atributos de la reunión, de tal modo que los pares de particiones correspondientes se puedan reunir independientemente.
- La eliminación de duplicados, la proyección, las operaciones de conjuntos (unión, intersección y diferencia) se pueden realizar mediante ordenación o asociación.
- Las operaciones de reunión externa se pueden implementar como extensiones simples de los algoritmos de reunión.
- Las técnicas de asociación y ordenación son duales, en el sentido en que muchas operaciones como la eliminación de duplicados, agregación, reuniones y reuniones externas que se pueden implementar mediante asociación también se pueden implementar por ordenación, y viceversa; es decir, cualquier operación que se pueda implementar con ordenación también puede serlo por asociación.
- Una expresión se puede evaluar mediante materialización, donde el sistema calcula el resultado de cada subexpresión y lo almacena en disco, y después lo usa para calcular el resultado de la expresión padre.
- El encauzamiento ayuda a evitar la escritura en disco de los resultados de muchas subexpresiones usando los resultados de la expresión padre incluso si se estuviesen generando.

Términos de repaso

- Procesamiento de consultas.
- Evaluación de primitivas.
- Plan de ejecución de consultas.
- Plan de evaluación de consultas.
- Motor de ejecución de consultas.
- Medidas del coste de las consultas.
- E/S secuencial.
- E/S paralela.
- Exploración de archivos.
- Búsqueda lineal.
- Búsqueda binaria.
- Selecciones usando índices.
- Rutas de acceso.
- Exploración de índices.
- Selección conjuntiva.
- Selección disyuntiva.
- Índice compuesto.
- Intersección de identificadores.
- Ordenación externa.
- Ordenación–mezcla externa.
- Secuencias.
- Mezcla de n vías.
- Equirreunión.
- Reunión en bucle anidado.
- Reunión en bucle anidado por bloques.
- Reunión en bucle anidado indexada.
- Reunión por mezcla.
- Reunión por ordenación–mezcla.
- Reunión por mezcla híbrida.
- Reunión por asociación.
  - Construir.
  - Prueba.
  - Entrada de construcción.
  - Entrada de prueba.
  - División recursiva.
  - Desbordamiento de la tabla de asociación.
  - Sesgo.
  - Factor de escape.
  - Resolución del desbordamiento.
  - Evitación del desbordamiento.
- Reunión por asociación híbrida.
- Árbol de operadores.
- Evaluación materializada.
- Memoria intermedia doble.
- Evaluación encauzada.

- ☐ Cauce bajo demanda (perezoso, extracción).
- ☐ Cauce por productor (impaciente, inserción).
- ☐ Iterador.

- Reunión encauzada.

Ejercicios prácticos

13.1 Considérese la siguiente consulta SQL para la base de datos bancaria:

select *T.nombre_sucursal*
from *sucursal T*, *sucursal S*
where *T.activos* > *S.activos* **and** *S.ciudad_sucursal* = "Arganzuela"

Escríbase una expresión del álgebra relacional equivalente a la dada que sea más eficiente. Justifíquese la elección.

13.2 Supóngase (para simplificar este ejercicio) que solamente cabe una tupla en un bloque y que la memoria puede contener como máximo tres marcos de página. Muéstrense las secuencias creadas en cada ciclo del algoritmo de ordenación—mezcla cuando se aplica para ordenar el primer atributo de las siguientes tuplas: (canguro, 17), (ualabí, 21), (emu, 1), (wombat, 13), (ornitorrinco, 3), (león, 8), (jabalí, 4), (cebra, 11), (koala, 6), (hiena, 9), (cálao, 2), (babuino, 12).

13.3 Dadas las relaciones $r_1(A, B, C)$ y $r_2(C, D, E)$ con las siguientes propiedades: r_1 tiene 20.000 tuplas, r_2 tiene 45.000 tuplas, en cada bloque caben, como máximo, 25 tuplas de r_1 o 30 tuplas de r_2. Estímese el número de transferencias de bloques y de búsquedas necesarias utilizando las siguientes estrategias para la reunión $r_1 \bowtie r_2$:

a. Reunión en bucle anidado.
b. Reunión en bucle anidado por bloques.
c. Reunión por mezcla.
d. Reunión por asociación.

13.4 El algoritmo de reunión en bucle anidado indexada descrito en el Apartado 13.5.3 puede ser ineficiente si el índice fuera secundario y hubiese varias tuplas con el mismo valor en los atributos de la reunión. ¿Por qué es ineficiente? Descríbase una forma de reducir el coste de recuperar las tuplas de la relación más interna utilizando ordenación. ¿Bajo qué condiciones sería este algoritmo más eficiente que la reunión por mezcla híbrida?

13.5 Sean r y s dos relaciones sin índices que no están ordenadas. Suponiendo una memoria infinita ¿cuál es la manera más económica (en términos de operaciones de E/S) para calcular $r \bowtie s$? ¿Cuánta memoria se necesita en este algoritmo?

13.6 Supóngase que hay un índice de árbol B^+ disponible en *ciudad_sucursal* de la relación *sucursal* y que no hay más índices. ¿Cuál sería el mejor modo de manejar las siguientes selecciones con negaciones?

a. $\sigma_{\neg(ciudad_sucursal < \text{"Arganzuela"})}(sucursal)$
b. $\sigma_{\neg(ciudad_sucursal = \text{"Arganzuela"})}(sucursal)$
c. $\sigma_{\neg(ciudad_sucursal < \text{"Arganzuela"} \vee activos < 5000)}(sucursal)$

13.7 Escríbase el pseudocódigo para un iterador que implemente la reunión en bucle anidado indexada, donde la relación externa esté encauzada. Defínanse las funciones iteradoras estándar *open* (), *next* () y *close* (). Muéstrese la información de estado del iterador que se debe guardar entre las llamadas.

13.8 Diséñense algoritmos basado en ordenación y asociación para el cálculo de la operación división.

13.9 ¿Cuál es el efecto sobre el coste de la mezcla de secuencias si el número de bloques de memoria intermedia se incrementa en cada ciclo mientras se mantiene la memoria total disponible para la memoria intermedia de las secuencias?

Ejercicios

13.10 ¿Por qué no hay que obligar a los usuarios a que elijan explícitamente una estrategia de procesamiento de la consulta? ¿Hay casos en los que *es* deseable que los usuarios sepan el coste de las distintas estrategias posibles? Razónese la respuesta.

13.11 Diséñese una variante del algoritmo híbrido de reunión por mezcla para el caso en el que las dos relaciones no están ordenadas según el orden físico de almacenamiento, pero ambas tienen un índice secundario ordenado en los atributos de la reunión.

13.12 Estímese el número de accesos a bloques necesitados por la solución del Ejercicio 13.11 para $r_1 \bowtie r_2$, donde r_1 y r_2 son como las relaciones definidas en el Ejercicio 13.3.

13.13 El algoritmo de reunión por asociación descrito en el Apartado 13.5.5 calcula la reunión natural de dos relaciones. Descríbase cómo extender el algoritmo de reunión por asociación para calcular la reunión externa por la izquierda, la reunión externa por la derecha y la reunión externa completa. *Sugerencia*: se puede mantener información adicional con cada tupla en el índice asociativo para detectar si alguna tupla en la relación de prueba concuerda con alguna tupla del índice asociativo. Compruébese el algoritmo con las relaciones *cliente* e *impositor*.

13.14 Escríbase pseudocódigo para un iterador que implemente una versión del algoritmo ordenación–mezcla donde el resultado de la mezcla final se encauce a los consumidores. El pseudocódigo debe definir las funciones estándar del iterador *abrir* (), *siguiente* () y *cerrar* (). Muéstrese la información de estado que el iterador debe mantener entre cada llamada.

13.15 Se usa encauzamiento para evitar escribir resultados intermedios a disco. Supóngase que se necesita ordenar una relación r usando ordenación–mezcla y reuniendo por mezcla el resultado con una relación s previamente ordenada.

- **a.** Descríbase cómo se puede encauzar la salida de la ordenación de r con la reunión por mezcla sin escribir a disco.
- **b.** La misma idea es aplicable incluso si ambas entradas a la reunión por mezcla son las salidas de las operaciones de ordenación–mezcla. Sin embargo, la memoria disponible tiene que compartirse entre las dos operaciones de mezcla (el propio algoritmo de reunión por mezcla necesita muy poca memoria). ¿Cuál es el efecto de tener que compartir la memoria sobre el coste de cada operación de ordenación–mezcla?

13.16 Supóngase que hay que calcular ${}_A\mathcal{G}_{sum(C)}(r)$ y ${}_{A,B}\mathcal{G}_{sum(C)}(r)$. Descríbase cómo calcular ambas juntas usando una única ordenación de r.

Notas bibliográficas

Todos los procesadores de consultas deben analizar instrucciones del lenguaje de consulta y deben traducirlas a su formato interno. El análisis de los lenguajes de consultas difiere poco del análisis de los lenguajes de programación tradicionales. La mayoría de los libros sobre compiladores, como Aho et al. [1986] y Tremblay y Sorenson [1985], tratan las principales técnicas de análisis y presentan la optimización desde el punto de vista de los lenguajes de programación.

Graefe [1993] presenta una excelente revisión de las técnicas de evaluación de consultas.

Knuth [1973] presenta un excelente descripción de algoritmos de ordenación externa, incluyendo una optimización denominada *selección de reemplazamiento* que puede originar secuencias iniciales que son (en media) el doble del tamaño de la memoria. Estudios más recientes en Nyberg et al. [1995] han demostrado que debido al mal comportamiento de la caché del procesador, la selección de reemplazamiento se comporta peor que quicksort en memoria para la generación de secuencias, eliminando las ventajas de la generación de secuencias más grandes. Nyberg et al. [1995] presenta una algoritmo eficiente de ordenación externa que considera los efectos de la caché del procesador. Los algoritmos de evaluación de consultas que consideran los efectos de la caché se han estudiado ampliamente; véase, por ejemplo, Harizopoulos y Ailamaki [2004].

De acuerdo con estudios del rendimiento realizados a mediados de los años setenta del siglo veinte, los sistemas de bases de datos de esa época utilizaban solamente reunión en bucle anidado y reunión por mezcla. Estos estudios, que estuvieron relacionados con el desarrollo de System R, determinaron que tanto la reunión en bucle anidado como la reunión por mezcla casi siempre proporcionaban el método de reunión óptimo (Blasgen y Eswaran [1976]; por tanto, estos son los dos únicos algoritmos de reunión implementados en System R. Sin embargo, el estudio de System R no incluyó el análisis de los algoritmos de reunión por asociación. Actualmente, estos algoritmos se consideran muy eficientes y se usan mucho.

Los algoritmos de reunión por asociación se desarrollaron inicialmente para sistemas de bases de datos paralelos. La técnica de reunión por asociación se describe en Kitsuregawa et al. [1983] y en Shapiro [1986] se describen extensiones incluyendo la reunión por asociación híbrida. Resultados más recientes de Zeller y Gray [1990] y de Davison y Graefe [1994] describen técnicas de reunión por asociación que se pueden adaptar a la memoria disponible, que es importante en sistemas donde se pueden ejecutar a la vez varias consultas. Graefe et al. [1998] describe el uso en SQL Server de Microsoft de las reuniones por asociación y los *equipos asociados*, que permiten el encauzamiento de las reuniones por asociación usando la misma división para todas las reuniones por asociación en una secuencia encauzada.

CAPÍTULO 14

Optimización de consultas

La **optimización de consultas** es el proceso de selección del plan de evaluación de las consultas más eficiente de entre las muchas estrategias generalmente disponibles para el procesamiento de una consulta dada, especialmente si la consulta es compleja. No se espera que los usuarios escriban las consultas de modo que puedan procesarse de manera eficiente. Por el contrario, se espera que el sistema cree un plan de evaluación que minimice el coste de la evaluación de las consultas. Ahí es donde entra en acción la optimización de consultas.

Un aspecto de la optimización de las consultas tiene lugar en el nivel del álgebra relacional, donde el sistema intenta hallar una expresión que sea equivalente a la expresión dada, pero de ejecución más eficiente. Otro aspecto es la elección de una estrategia detallada para el procesamiento de la consulta, como puede ser la selección del algoritmo que se usará para ejecutar una operación, la selección de los índices concretos que se van a emplear, etc.

La diferencia en coste (en términos de tiempo de evaluación) entre una estrategia buena y una mala suele ser sustancial, y puede resultar de varios órdenes de magnitud. Por tanto, merece la pena que el sistema invierta una cantidad importante de tiempo en la selección de una buena estrategia para el procesamiento de la consulta, aunque esa consulta sólo se ejecute una vez.

14.1 Visión general

Considérese la expresión del álgebra relacional para la consulta "Hallar los nombres de todos los clientes que tengan una cuenta en cualquier sucursal ubicada en Arganzuela".

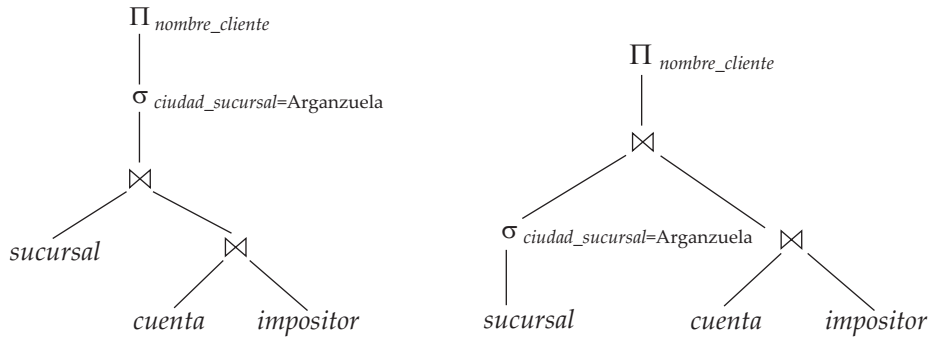
$$\Pi_{nombre_cliente}\ (\sigma_{ciudad_sucursal = \text{“Arganzuela”}}\ (sucursal \bowtie\ (cuenta \bowtie\ impositor)))$$

Esta expresión crea una relación intermedia de gran tamaño, $sucursal \bowtie\ cuenta \bowtie\ impositor$. Sin embargo, sólo resultan de interés unas pocas tuplas de esta relación (las correspondientes a las sucursales ubicadas en Arganzuela), y sólo en uno de los seis atributos de la relación. Dado que sólo son relevantes las tuplas de la relación *sucursal* que corresponden a las sucursales ubicadas en Arganzuela, no hace falta considerar las tuplas que no cumplen $ciudad_sucursal =$ "Arganzuela." Al reducir el número de tuplas de la relación *sucursal* a las que es necesario tener acceso, se reduce el tamaño del resultado intermedio. La consulta queda ahora representada por la siguiente expresión del álgebra relacional:

$$\Pi_{nombre_cliente}\ (\ (\sigma_{ciudad_sucursal = \text{“Arganzuela”}}\ (sucursal)) \bowtie\ (cuenta \bowtie\ impositor))$$

la cual es equivalente a la expresión algebraica original, pero genera relaciones intermedias de menor tamaño. La Figura 14.1 muestra la expresión inicial y la transformada.

Dada una expresión del álgebra relacional, es labor del optimizador de consultas diseñar un plan de evaluación para ellas que calcule el mismo resultado que la expresión dada, y de la forma menos costosa (o, como mínimo, no mucho más que la menos costosa).



(a) Árbol inicial de la expresión (b) Árbol transformado de la expresión

Figura 14.1 Expresiones equivalentes.

Para hallar el plan de evaluación de consultas menos costoso el optimizador necesita generar planes alternativos que produzcan el mismo resultado que la expresión dada y escoger el de menor coste. La generación de planes de evaluación de consultas implica tres etapas: (1) la generación de expresiones que sean equivalentes lógicamente a la expresión dada, (2) la estimación del coste de cada plan de evaluación y (3) la anotación de las expresiones alternativas resultantes para generar planes distintos de ejecución. Las etapas (1) y (3) se encuentran entrelazadas en el optimizador de consultas; algunas expresiones se generan y se anotan, luego se generan y se anotan otras expresiones, etc. La etapa (2) se realiza en segundo plano recopilando información estadística sobre las relaciones, tales como sus tamaños y la profundidad de los índices, para hacer una buena estimación del coste de un plan.

Para implementar la primera etapa el optimizador de consultas debe generar expresiones equivalentes a la expresión dada. Esto se lleva a cabo mediante las *reglas de equivalencia*, que especifican el modo de transformar una expresión en otra lógicamente equivalente. Estas reglas se describen en el Apartado 14.2.

En el Apartado 14.3 se describe el modo de estimar las estadísticas de los resultados de cada operación en los planes de consultas. El empleo de estas estadísticas con las fórmulas de costes del Capítulo 13 permite estimar los costes de cada operación. Los costes individuales se combinan para determinar el coste estimado de la evaluación de la expresión de álgebra relacional dada, como se indicó previamente en el Apartado 13.7.

En el Apartado 14.4 se describe el modo de escoger un plan de evaluación de consultas. Se puede escoger uno basado en el coste estimado de los planes. Dado que el coste es una estimación, el plan seleccionado no es necesariamente el menos costoso; no obstante, siempre y cuando las estimaciones sean buenas, es probable que el plan sea el menos costoso, o no mucho más costoso. Esta optimización, denominada **optimización basada en costes**, se describe en el Apartado 14.4.2.

Finalmente, las vistas materializadas ayudan a acelerar el procesamiento de ciertas consultas. En el Apartado 14.5 se estudia el modo de "mantener" las vistas materializadas (es decir, mantenerlas actualizadas) y la manera de llevar a cabo la optimización de consultas con las vistas materializadas.

14.2 Transformación de expresiones relacionales

Las consultas se pueden expresar de varias maneras diferentes, con costes de evaluación diferentes. En este apartado, en lugar de tomar la expresión relacional original, se consideran expresiones alternativas equivalentes.

Se dice que dos expresiones del álgebra relacional son **equivalentes** si, en cada ejemplar legal de la base de datos, las dos expresiones generan el mismo conjunto de tuplas (recuérdese que un ejemplar legal de la base de datos es la que satisface todas las restricciones de integridad especificadas en el esquema de la base de datos). Obsérvese que el orden de las tuplas resulta irrelevante; puede que las dos expresiones generen las tuplas en órdenes diferentes, pero se considerarán equivalentes siempre que el conjunto de tuplas sea el mismo.

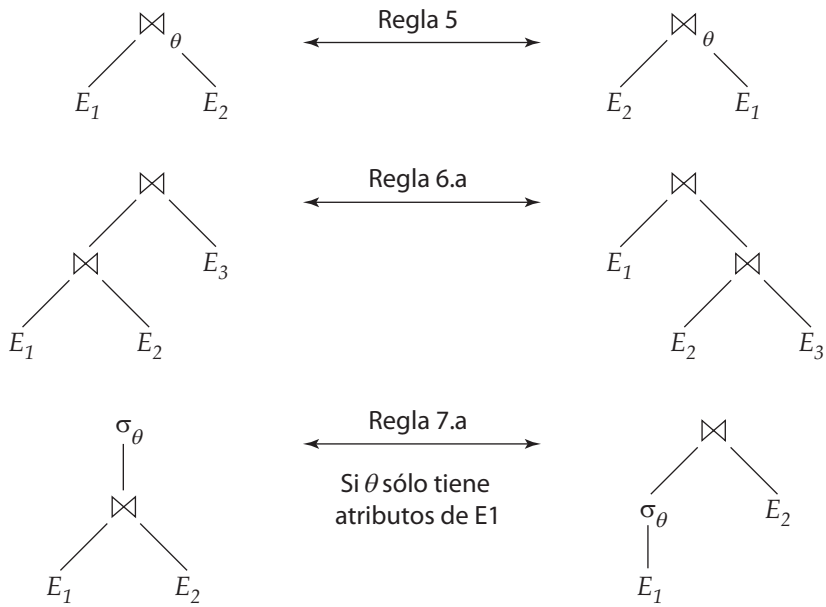


Figura 14.2 Representación gráfica de las equivalencias.

En SQL las entradas y las salidas son multiconjuntos de tuplas, y se usa la versión para multiconjuntos del álgebra relacional para evaluar las consultas de SQL. Se dice que dos expresiones de la versión *para multiconjuntos* del álgebra relacional son equivalentes si en cada base de datos legal las dos expresiones generan el mismo multiconjunto de tuplas. El estudio de este capítulo se basa en el álgebra relacional. Las extensiones a la versión para multiconjuntos del álgebra relacional se dejan al lector como ejercicios.

14.2.1 Reglas de equivalencia

Una **regla de equivalencia** establece que dos expresiones son equivalentes. Se puede sustituir la primera por la segunda forma, o viceversa, ya que las dos expresiones generan el mismo resultado en cualquier base de datos válida. El optimizador usa las reglas de equivalencia para transformar las expresiones en otras lógicamente equivalentes.

A continuación se enumeran varias reglas generales de equivalencia para las expresiones del álgebra relacional. Algunas de las equivalencias relacionadas aparecen en la Figura 14.2. Se usan $\theta, \theta_1, \theta_2$, etc., para denotar los predicados, L_1, L_2, L_3, etc., para denotar las listas de atributos y E, E_1, E_2, etc. para denotar las expresiones del álgebra relacional. El nombre de relación r no es más que un caso especial de expresión del álgebra relacional y puede usarse siempre que aparezca E.

1. Las operaciones de selección conjuntivas pueden dividirse en una secuencia de selecciones individuales. Esta transformación se denomina cascada de σ.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Las operaciones de selección son **conmutativas**.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Sólo son necesarias las últimas operaciones de una secuencia de operaciones de proyección, las demás pueden omitirse. Esta transformación también puede denominarse cascada de Π.

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) = \Pi_{L_1}(E)$$

4. Las selecciones pueden combinarse con los productos cartesianos y con las reuniones zeta.
   a. $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$. Esta expresión es precisamente la definición de la reunión zeta.
   b. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

5. Las operaciones de reunión zeta son conmutativas.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

Realmente, el orden de los atributos es diferente en el término de la derecha y en el de la izquierda, por lo que la equivalencia no se cumple si se tiene en cuenta el orden de los atributos. Se puede añadir una operación proyección a uno de los lados de la equivalencia para reordenar los atributos de la manera adecuada, pero para mayor sencillez se omite la proyección y se ignora el orden de los atributos en la mayor parte de los ejemplos.

Recuérdese que el operador de reunión natural es simplemente un caso especial del operador de reunión zeta; por tanto, las reuniones naturales también son conmutativas.

6. **a.** Las operaciones de reunión natural son **asociativas**.

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

b. Las reuniones zeta son asociativas en el sentido siguiente:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

donde θ_2 implica solamente atributos de E_2 y de E_3. Cualquiera de estas condiciones puede estar vacía; por tanto, se deduce que la operación producto cartesiano ($\times$) también es asociativa. La conmutatividad y la asociatividad de las operaciones de reunión son importantes para la reordenación de las reuniones en la optimización de las consultas.

7. La operación selección se distribuye por la operación reunión zeta bajo las dos condiciones siguientes:

a. Cuando todos los atributos de la condición de selección θ_0 implican únicamente los atributos de una de las expresiones (por ejemplo, E_1) que se están reuniendo.

$$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$$

b. Cuando la condición de selección θ_1 implica únicamente los atributos de E_1 y θ_2 implica únicamente los atributos de E_2.

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$$

8. La operación proyección se distribuye entre la operación reunión zeta bajo las condiciones siguientes.

a. Sean L_1 y L_2 atributos de E_1 y de E_2, respectivamente. Supóngase que la condición de reunión θ implica únicamente los atributos de $L_1 \cup L_2$. Entonces,

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

b. Considérese una reunión $E_1 \bowtie_\theta E_2$. Sean L_1 y L_2 conjuntos de atributos de E_1 y de E_2, respectivamente. Sean L_3 los atributos de E_1 que están implicados en la condición de reunión θ, pero que no están en $L_1 \cup L_2$, y sean L_4 los atributos de E_2 que están implicados en la condición de reunión θ, pero que no están en $L_1 \cup L_2$. Entonces,

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

9. Las operaciones de conjuntos unión e intersección son conmutativas.

$$E_1 \cup E_2 = E_2 \cup E_1$$

$$E_1 \cap E_2 = E_2 \cap E_1$$

La diferencia de conjuntos no es conmutativa.

10. La unión y la intersección de conjuntos son asociativas.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$

$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. La operación selección se distribuye por las operaciones de unión, intersección y diferencia de conjuntos.

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - \sigma_P(E_2)$$

De manera parecida, la equivalencia anterior, con $-$ sustituido por $\cup$ o por $\cap$, también es válida. Además,

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - E_2$$

La equivalencia anterior, con $-$ sustituido por $\cap$, también es válida, pero no se cumple si $-$ se sustituye por $\cup$.

12. La operación proyección es distributiva con respecto a la operación unión.

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

Ésta es sólo una lista parcial de las equivalencias. En los ejercicios se estudian más equivalencias que implican a los operadores relacionales extendidos, como la reunión externa y la agregación.

14.2.2 Ejemplos de transformaciones

Ahora se mostrará el empleo de las reglas de equivalencia. Se usará el ejemplo del banco con los esquemas de relaciones:

Esquema_sucursal = (*nombre_sucursal, ciudad_sucursal, activos*)
Esquema_cuenta = (*número_cuenta, nombre_sucursal, saldo*)
Esquema_impositor = (*nombre_cliente, número_cuenta*)

Las relaciones *sucursal*, *cuenta* e *impositor* son ejemplos de estos esquemas.

En el ejemplo del Apartado 14.1 la expresión

$$\Pi_{nombre_cliente}(\sigma_{ciudad_sucursal = \text{“Arganzuela”}}(sucursal \bowtie (cuenta \bowtie impositor)))$$

se transformaba en la expresión siguiente,

$$\Pi_{nombre_cliente}((\sigma_{ciudad_sucursal = \text{“Arganzuela”}}(sucursal)) \bowtie (cuenta \bowtie impositor))$$

la cual es equivalente a la expresión algebraica original, pero generando relaciones intermedias de menor tamaño. Esta transformación se puede llevar a cabo empleando la regla 7.a. Recuérdese que la regla sólo dice que las dos expresiones son equivalentes; no dice que una sea mejor que la otra.

Se pueden usar varias reglas de equivalencia, una tras otra, sobre una consulta o sobre partes de una consulta. Como ejemplo, supóngase que se modifica la consulta original para restringir la atención a los clientes que tienen un saldo superior a 1000 €. La nueva consulta del álgebra relacional es

$$\Pi_{nombre_cliente}\ (\sigma_{ciudad_sucursal = \text{“Arganzuela”} \wedge saldo > 1000} (sucursal \bowtie (cuenta \bowtie impositor)))$$

No se puede aplicar el predicado de la selección directamente a la relación *sucursal*, ya que éste implica atributos tanto de la relación *sucursal* como de la relación *cuenta*. No obstante, se puede aplicar antes la regla 6.a (asociatividad de la reunión natural) para transformar la reunión *sucursal* $\bowtie$ (*cuenta* $\bowtie$ *impositor*) en (*sucursal* $\bowtie$ *cuenta*) $\bowtie$ *impositor*:

$$\Pi_{nombre_cliente}\ (\sigma_{ciudad_sucursal = \text{“Arganzuela”} \wedge saldo > 1000} ((sucursal \bowtie cuenta) \bowtie impositor))$$

Luego, empleando la regla 7.a, se puede reescribir la consulta como

$$\Pi_{nombre_cliente}\ ((\sigma_{ciudad_sucursal = \text{“Arganzuela”} \wedge saldo > 1000} (sucursal \bowtie cuenta)) \bowtie impositor)$$

Considérese ahora la subexpresión de selección de esta expresión. Empleando la regla 1 se puede dividir la selección en dos, para obtener la subexpresión siguiente:

$$\sigma_{ciudad_sucursal = \text{“Arganzuela”}}\ (\sigma_{saldo > 1000}\ (sucursal \bowtie cuenta))$$

Las dos expresiones anteriores seleccionan tuplas con *ciudad_sucursal* = "Arganzuela" y *saldo* > 1000. Sin embargo, la última forma de la expresión ofrece una nueva oportunidad de aplicar la regla 7.a ("llevar a cabo primero las selecciones"), que da lugar a la subexpresión

$$\sigma_{ciudad_sucursal = \text{"Arganzuela"}} (sucursal) \bowtie \sigma_{saldo > 1000} (cuenta)$$

La Figura 14.3 muestra la expresión inicial y la expresión final después de todas estas transformaciones. También se podría haber usado la regla 7.b para obtener directamente la expresión final, sin usar la regla 1 para dividir la selección en dos selecciones. De hecho, la regla 7.b puede obtenerse de las reglas 1 y 7.a.

Se dice que un conjunto de reglas de equivalencia es **mínimo** si no se puede obtener ninguna regla a partir de una reunión de las demás. El ejemplo anterior muestra que el conjunto de reglas de equivalencia del Apartado 14.2.1 no es mínimo. Se puede generar una expresión equivalente a la original de diferentes maneras; el número de maneras diferentes de generar una expresión aumenta cuando se usa un conjunto de reglas de equivalencia que no es mínimo. Los optimizadores de consultas, por tanto, usan conjuntos mínimos de reglas de equivalencia.

Considérese ahora la siguiente forma de la consulta de ejemplo:

$$\Pi_{nombre_cliente} ((\sigma_{ciudad_sucursal = \text{"Arganzuela"}} (sucursal) \bowtie cuenta) \bowtie impositor)$$

Cuando se calcula la subexpresión

$$(\sigma_{ciudad_sucursal = \text{"Arganzuela"}} (sucursal) \bowtie cuenta)$$

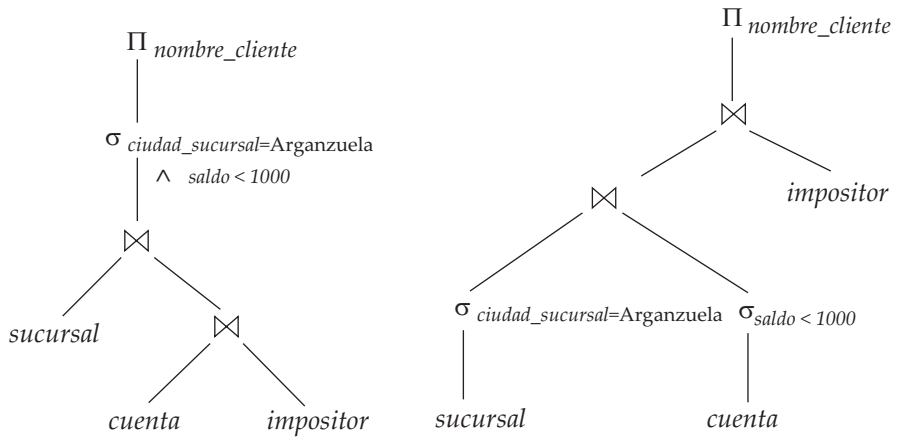
se obtiene una relación cuyo esquema es:

(*nombre_sucursal*, *ciudad_sucursal*, *activos*, *número_cuenta*, *saldo*)

Es posible eliminar varios atributos del esquema, forzando las proyecciones de acuerdo con las reglas de equivalencia 8.a y 8.b. Los únicos atributos que se deben conservar son los que aparecen en el resultado de la consulta y los que se necesitan para procesar las operaciones subsiguientes. Al eliminar los atributos innecesarios se reduce el número de columnas del resultado intermedio. Por tanto, se reduce el tamaño del resultado intermedio. En el ejemplo el único atributo que se necesita de la reunión de *sucursal* y de *cuenta* es *número_cuenta*. Por tanto, se puede modificar la expresión hasta

$$\Pi_{nombre_cliente} ((\Pi_{número_cuenta} ((\sigma_{ciudad_sucursal = \text{"Arganzuela"}} (sucursal)) \bowtie cuenta)) \bowtie impositor)$$

La proyección $\Pi_{número_cuenta}$ reduce el tamaño de los resultados de las reuniones intermedias.



(a) Árbol inicial de la expresión (b) Árbol tras varias transformaciones

Figura 14.3 Varias transformaciones.

14.2.3 Ordenación de las reuniones

Es importante conseguir una buena ordenación de las operaciones reunión para reducir el tamaño de los resultados temporales; por tanto, la mayor parte de los optimizadores de consultas prestan mucha atención al orden de las reuniones. Como se mencionó en el Capítulo 3 y en la regla de equivalencia 6.a, la operación reunión natural es asociativa. Por tanto, para todas las relaciones r_1, r_2 y r_3,

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

Aunque estas expresiones sean equivalentes, los costes de calcular cada una de ellas pueden ser diferentes. Considérese una vez más la expresión

$$\Pi_{nombre_cliente} ((\sigma_{ciudad_sucursal = \text{“Arganzuela”}} (sucursal)) \bowtie cuenta \bowtie impositor)$$

Se podría escoger calcular primero *cuenta* $\bowtie$ *impositor* y luego combinar el resultado con

$$\sigma_{ciudad_sucursal = \text{“Arganzuela”}} (sucursal)$$

Sin embargo, es probable que *cuenta* $\bowtie$ *impositor* sea una relación de gran tamaño, ya que contiene una tupla por cada cuenta. En cambio,

$$\sigma_{ciudad_sucursal = \text{“Arganzuela”}} (sucursal) \bowtie cuenta$$

es, probablemente, una relación de pequeño tamaño. Para comprobar que es así, se observa que, dado que el banco tiene un gran número de sucursales ampliamente distribuidas, es probable que sólo una pequeña parte de los clientes del banco tenga cuenta en las sucursales ubicadas en Arganzuela. Por tanto, la expresión anterior da lugar a una tupla por cada cuenta abierta por un residente de Arganzuela. Así, la relación temporal que se debe almacenar es menor que si se hubiera calculado primero *cuenta* $\bowtie$ *impositor*.

Hay otras opciones a considerar a la hora de evaluar la consulta. No hay que preocuparse del orden en que aparecen los atributos en las reuniones, ya que resulta sencillo cambiarlo antes de mostrar el resultado. Por tanto, para todas las relaciones r_1 y r_2,

$$r_1 \bowtie r_2 = r_2 \bowtie r_1$$

Es decir, la reunión natural es conmutativa (regla de equivalencia 5).

Mediante la asociatividad y la conmutatividad de la reunión natural (reglas 5 y 6) se puede considerar reescribir la expresión del álgebra relacional como

$$\Pi_{nombre_cliente} (((\sigma_{ciudad_sucursal = \text{“Arganzuela”}} (sucursal)) \bowtie impositor) \bowtie cuenta)$$

Es decir, se puede calcular primero

$$(\sigma_{ciudad_sucursal = \text{“Arganzuela”}} (sucursal)) \bowtie impositor$$

y, luego, reunir el resultado con *cuenta*. Obsérvese, no obstante, que no existen atributos en común entre *Esquema_sucursal* y *Esquema_impositor*, por lo que la reunión no es más que un producto cartesiano. Si hay s sucursales en Arganzuela e i tuplas en la relación *impositor*, este producto cartesiano genera $s * i$ tuplas, una por cada par posible de tuplas de impositor y de sucursal (independientemente de si la cuenta de *impositor* está abierta en la sucursal). Por tanto, parece que este producto cartesiano producirá una relación temporal de gran tamaño. En consecuencia, esta estrategia se rechaza. No obstante, si el usuario ha introducido la expresión anterior, se pueden usar la asociatividad y la conmutatividad de la reunión natural para transformarla en la expresión más eficiente que se usó anteriormente.

14.2.4 Enumeración de expresiones equivalentes

Los optimizadores de consultas usan las reglas de equivalencia para generar de manera sistemática expresiones equivalentes a la expresión de consulta dada. Conceptualmente el proceso se desarrolla de la manera siguiente. Dada una expresión, si alguna subexpresión coincide con el lado izquierdo o derecho de una regla de equivalencia, el optimizador genera una nueva expresión con la subexpresión transformada de modo que coincida con el otro lado de la regla. Este proceso continúa hasta que no se puedan generar expresiones nuevas.

El proceso anterior resulta costoso tanto en espacio como en tiempo. Éste es el modo en que se puede reducir los requisitos de espacio: si se genera una expresión E_1 a partir de una expresión E_2 empleando una regla de equivalencia, E_1 y E_2 son parecidas en estructura y tienen subexpresiones que son idénticas. Las técnicas de representación de expresiones que permiten que las dos expresiones apunten a las subexpresiones compartidas pueden reducir de manera significativa los requisitos de espacio, y muchos optimizadores de consultas las usan.

Además, no siempre resulta necesario generar todas las posibles expresiones con las reglas de equivalencia. Si un optimizador tiene en cuenta las estimaciones de costes, puede que logre evitar el examen de algunas de las expresiones, como se verá en el Apartado 14.4. Se puede reducir el tiempo necesario para la optimización empleando técnicas como éstas.

14.3 Estimación de las estadísticas de los resultados de las expresiones

El coste de cada operación depende del tamaño y de otras estadísticas de sus valores de entrada. Dada una expresión como $a \bowtie (b \bowtie c)$, para estimar el coste de combinar a con $(b \bowtie c)$ hay que hacer estimaciones de estadísticas como el tamaño de $b \bowtie c$.

En este apartado se relacionarán en primer lugar algunas estadísticas de las relaciones de bases de datos que se almacenan en los catálogos de los sistemas de bases de datos y luego se mostrará el modo de usar las estadísticas para estimar estadísticas de los resultados de varias operaciones relacionales.

Algo que quedará claro más adelante en este apartado es que las estimaciones no son muy precisas, ya que se basan en suposiciones que pueden no cumplirse exactamente. El plan de evaluación de consultas que tenga el coste estimado de ejecución más reducido puede, por tanto, no tener el coste real de ejecución más bajo. Sin embargo, la experiencia real ha mostrado que, aunque las estimaciones no sean muy precisas, los planes con los costes estimados más reducidos tienen costes de ejecución reales que, o bien son los más reducidos o bien se hallan cercanos al menor coste real de ejecución.

14.3.1 Información de catálogo

Los catálogos de los SGBD almacenan la siguiente información estadística sobre las relaciones de las bases de datos:

- n_r, el número de tuplas de la relación *r*.
- b_r, el número de bloques que contienen tuplas de la relación r.
- t_r, el tamaño de cada tupla de la relación *r* en bytes.
- f_r, el factor de bloqueo de la relación r—es decir, el número de tuplas de la relación r que caben en un bloque.
- $V(A, r)$, el número de valores distintos que aparecen en la relación *r* para el atributo A. Este valor es igual que el tamaño de $\Pi_A(r)$. Si A es una clave de la relación r, $V(A, r)$ es n_r.

La última estadística, $V(A, r)$, también puede calcularse para conjuntos de atributos, si se desea, en vez de sólo para atributos aislados. Por tanto, dado un conjunto de atributos, $\mathcal{A}$, $V(\mathcal{A}, r)$ es el tamaño de $\Pi_{\mathcal{A}}(r)$.

Si se da por supuesto que las tuplas de la relación r se almacenan físicamente juntas en un archivo, se cumple la ecuación siguiente:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

Las estadísticas sobre los índices, como las alturas de los árboles B^+ y el número de páginas hojas de los índices, también se conservan en el catálogo.

Por tanto, si se desean conservar estadísticas precisas, cada vez que se modifica una relación también hay que actualizar las estadísticas. Esta actualización supone una sobrecarga sustancial. Por tanto, la

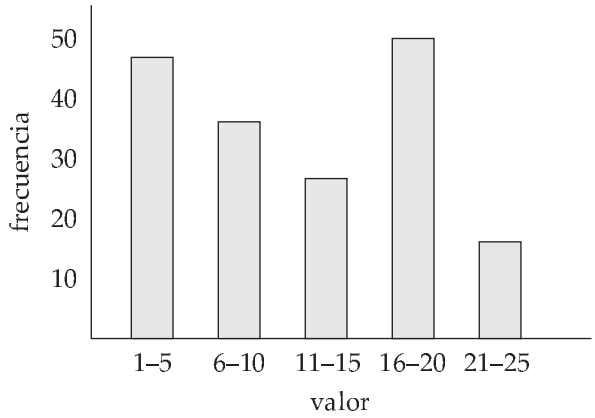


Figura 14.4 Ejemplo de histograma.

mayor parte de los sistemas no actualizan las estadísticas con cada modificación. En lugar de eso, actualizan las estadísticas durante los periodos de poca carga del sistema. En consecuencia, puede que las estadísticas usadas para escoger una estrategia de procesamiento de consultas no sean completamente exactas. Sin embargo, si no se producen demasiadas actualizaciones en los intervalos entre las actualizaciones de las estadísticas, éstas serán lo bastante precisas como para proporcionar una buena estimación de los costes relativos de los diferentes planes.

La información estadística indicada aquí está simplificada. Los optimizadores reales suelen conservar información estadística adicional para mejorar la precisión de sus estimaciones de costes de los planes de evaluación. Por ejemplo, la mayoría de las bases de datos almacenan la distribución de los valores de cada atributo en forma de **histograma**: en los histogramas los valores del atributo se dividen en una serie de rangos, y con cada rango el histograma asocia el número de tuplas cuyo valor del atributo se halla en ese rango. En la Figura 14.4 se muestra un ejemplo de histograma para un atributo entero que toma valores en el rango de 1 a 25.

Los histogramas usados en los sistemas de bases de datos registran normalmente el número de los valores distintos en cada rango, además del número de tuplas con en ese rango.

Como ejemplo de histograma, el rango de valores del atributo *edad* de la relación *persona* puede dividirse en $0{-}9$, $10{-}19$, ..., $90{-}99$ (suponiendo una edad máxima de 99). Con cada rango se almacena un recuento del número de tuplas *persona* cuyos valores de *edad* se hallan en ese rango, junto con el número de valores distintos de edad que se encuentran en ese rango. Sin la información del histograma un optimizador tendría que suponer que la distribución de los valores es uniforme; es decir, que cada rango tiene el mismo recuento.

Un histograma ocupa muy poco espacio, por lo que los histogramas sobre varios atributos diferentes se pueden almacenar en el catálogo del sistema. En los sistemas de bases de datos se usan varios tipos de histogramas. Por ejemplo, el **histograma de equianchura** divide el rango de valores en rangos de igual tamaño, mientras que el **histograma de equiprofundidad** ajusta las fronteras de los rangos de forma que cada rango tenga el mismo número de valores.

14.3.2 Estimación del tamaño de la selección

La estimación del tamaño del resultado de una operación selección depende del predicado de la selección. En primer lugar se considerará un solo predicado de igualdad, luego un solo predicado de comparación y, finalmente, combinaciones de predicados.

- $\sigma_{A=a}(r)$: Si se supone una distribución uniforme de los valores (es decir, que cada valor aparece con igual probabilidad), se puede estimar que el resultado de la selección tiene $n_r/V(A,r)$ tuplas, suponiendo que el valor a aparece en el atributo A de algún registro de r. La suposición de que el valor a de la selección aparece en algún registro suele ser cierta, y las estimaciones de costes suelen hacerla de manera implícita. No obstante, no suele ser realista suponer que cada valor aparece con igual probabilidad. El atributo *nombre_sucursal* de la relación *cuenta* es un ejemplo en

el que esta suposición no es válida. Hay una tupla de la relación *cuenta* para cada cuenta. Resulta razonable esperar que las sucursales grandes tengan más cuentas que las pequeñas. Por tanto, algunos valores *nombre_sucursal* aparecen con mayor probabilidad que otros. Pese al hecho de que la suposición de distribución uniforme no suele ser correcta, resulta una aproximación razonable de la realidad en muchos casos, y ayuda a que la representación siga siendo relativamente sencilla.

Si hay disponible un histograma sobre el atributo A, se puede localizar el rango que contiene el valor a y modificar la estimación anterior $n_r/V(A,r)$ usando el contador de frecuencia para el rango en lugar de n_r, y el número de valores distintos que aparecen en el rango en lugar de $V(A,r)$.

- $\sigma_{A \leq v}(r)$: Considérese una selección de la forma $\sigma_{A \leq v}(r)$. Si el valor real usado en la comparación (v) está disponible en el momento de la estimación del coste, puede hacerse una estimación más precisa. Los valores mínimo y máximo ($\min(A,r)$ y $\max(A,r)$) del atributo pueden almacenarse en el catálogo. Suponiendo que los valores están distribuidos de manera uniforme, se puede estimar el número de registros que cumplirán la condición $A \leq v$ como 0 si $v < \min(A,r)$, como n_r si $v \geq \max(A,r)$ y como

$$n_r \cdot \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$$

en el resto de los casos.

Si hay disponible un histograma sobre el atributo A, se puede obtener una estimación más precisa; se dejan los detalles al lector.

En algunos casos, como cuando la consulta forma parte de un procedimiento almacenado, puede que el valor v no esté disponible cuando se optimice la consulta. En esos casos, se supondrá que aproximadamente la mitad de los registros cumplen la condición de comparación. Es decir, se supone que el resultado tiene $n_r/2$ tuplas; la estimación puede resultar muy imprecisa, pero es lo mejor que se puede hacer sin más información.

- Selecciones complejas:
  - **Conjunción**. Una *selección conjuntiva* es una selección de la forma

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(r)$$

Se puede estimar el tamaño del resultado de esta selección: para cada θ_i, se estima el tamaño de la selección $\sigma_{\theta_i}(r)$, denotada por s_i, como se ha descrito anteriormente. Por tanto, la probabilidad de que una tupla de la relación satisfaga la condición de selección θ_i es s_i/n_r.

La probabilidad anterior se denomina **selectividad** de la selección $\sigma_{\theta_i}(r)$. Suponiendo que las condiciones sean *independientes* entre sí, la probabilidad de que una tupla satisfaga todas las condiciones es simplemente el producto de todas estas probabilidades. Por tanto, se estima el número de tuplas de la selección completa como

$$n_r * \frac{s_1 * s_2 * \cdots * s_n}{n_r^n}$$

  - **Disyunción**. Una *selección disyuntiva* es una selección de la forma

$$\sigma_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n}(r)$$

Una condición disyuntiva se satisface por la unión de todos los registros que satisfacen las condiciones simples θ_i.

Como anteriormente, admitamos que s_i/n_r denota la probabilidad de que una tupla satisfaga la condición θ_i. La probabilidad de que la tupla satisfaga la disyunción es, pues, 1 menos la probabilidad de que no satisfaga *ninguna* de las condiciones:

$$1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \cdots * (1 - \frac{s_n}{n_r})$$

Multiplicando este valor por n_r se obtiene el número estimado de tuplas que satisfacen la selección.

- **Negación**. A falta de valores nulos el resultado de una selección $\sigma_{\neg\theta}(r)$ es simplemente las tuplas de r que no están en $\sigma_\theta(r)$. Ya se sabe el modo de estimar el número de tuplas de $\sigma_\theta(r)$. El número de tuplas de $\sigma_{\neg\theta}(r)$ se estima, por tanto, que es $n(r)$ menos el número estimado de tuplas de $\sigma_\theta(r)$.

  Se pueden tener en cuenta los valores nulos estimando el número de tuplas para las que la condición θ se evalúa como *desconocida*, y restar ese número de la estimación anterior que ignora los valores nulos. La estimación de ese número exige conservar estadísticas adicionales en el catálogo.

14.3.3 Estimación del tamaño de las reuniones

En este apartado se verá el modo de estimar el tamaño del resultado de una reunión.

El producto cartesiano $r \times s$ contiene $n_r * n_s$ tuplas. Cada tupla de $r \times s$ ocupa $t_r + t_s$ bytes, de donde se puede calcular el tamaño del producto cartesiano.

La estimación del tamaño de una reunión natural resulta algo más complicada que la estimación del tamaño de una selección del producto cartesiano. Sean $r(R)$ y $s(S)$ dos relaciones.

- Si $R \cap S = \emptyset$, es decir, las relaciones no tienen ningún atributo en común, entonces $r \bowtie s$ es igual que $r \times s$, y se puede usar la técnica de estimación anterior para los productos cartesianos.
- Si $R \cap S$ es clave de R, entonces se sabe que cada tupla de s se combinará como máximo con una tupla de r. Por tanto, el número de tuplas de $r \bowtie s$ no es mayor que el número de tuplas de s. El caso de que $R \cap S$ sea clave de S es simétrico al caso que se acaba de describir. Si $R \cap S$ forma una clave externa de S, que hace referencia a R, el número de tuplas de $r \bowtie s$ es exactamente el mismo que el número de tuplas de s.
- El caso más difícil es que $R \cap S$ no sea clave de R ni de S. En ese caso se supone, como se hizo para las selecciones, que todos los valores aparecen con igual probabilidad. Considérese una tupla t de r y supóngase que $R \cap S = \{A\}$. Se estima que la tupla t produce

$$\frac{n_s}{V(A,s)}$$

  tuplas en $r \bowtie s$, ya que este número es el número promedio de tuplas de s con un valor dado para los atributos A. Considerando todas las tuplas de r se estima que hay

$$\frac{n_r * n_s}{V(A,s)}$$

  tuplas in $r \bowtie s$. Obsérvese que, si se invierten los papeles de r y de s en la estimación anterior, se obtiene una estimación de

$$\frac{n_r * n_s}{V(A,r)}$$

  tuplas en $r \bowtie s$. Estas dos estimaciones son diferentes si $V(A,r) \neq V(A,s)$. Si esta situación se produce, probablemente haya tuplas pendientes que no participen en la reunión. Por tanto, probablemente la menor de las dos estimaciones sea la más precisa.

  La estimación anterior del tamaño de la reunión puede ser demasiado elevada si los valores de $V(A,r)$ para el atributo A de r tienen pocas tuplas en común con los valores de $V(A,s)$ para el atributo A de s. No obstante, es improbable que se dé esta situación en la realidad, ya que las tuplas pendientes o bien no existen o sólo constituyen una pequeña fracción de las tuplas, en la mayor parte de las relaciones reales.

  Lo más importante es que la estimación anterior depende de la suposición de que todos los valores aparecen con igual probabilidad. Hay que usar técnicas más sofisticadas para la estimación del tamaño si esta suposición no resulta válida. Por ejemplo, si se tuviesen histogramas sobre los atributos de la reunión de ambas relaciones, y ambos histogramas tuviesen los mismos rangos, se podría usar la técnica de la estimación anterior en los dos rangos usando el número de filas con valores en el rango en lugar de n_r o de n_s, y el número de valores distintos en el rango en lugar de $V(A,r)$ o de $V(A,s)$. Después se sumarían las estimaciones de tamaño obtenidas para cada rango para estimar el tamaño total. Como ejercicio, se deja al lector el caso en que ambas

relaciones tengan histogramas sobre el atributo de la reunión, pero con rangos diferentes en los histogramas.

Se puede estimar el tamaño de una reunión zeta $r \bowtie_\theta s$ reescribiendo la reunión como $\sigma_\theta(r \times s)$ y empleando las estimaciones de tamaño de los productos cartesianos junto con las estimaciones de tamaño de las selecciones, que se vieron en el Apartado 14.3.2.

Para ilustrar todas estas maneras de estimar el tamaño de las reuniones, considérese la expresión:

$$\textit{impositor} \bowtie \textit{cliente}$$

Supóngase que se dispone de la siguiente información de catálogo sobre las dos relaciones:

- $n_{cliente} = 10.000$.
- $f_{cliente} = 25$, lo que implica que $b_{cliente} = 10.000/25 = 400$.
- $n_{impositor} = 5.000$.
- $f_{impositor} = 50$, lo que implica que $b_{impositor} = 5.000/50 = 100$.
- V(*nombre_cliente, impositor*) $= 2.500$, lo que implica que, en promedio, cada cliente tiene dos cuentas.

Supóngase también que *nombre_cliente* de *impositor* es clave externa de *cliente*.

En el ejemplo de *impositor* $\bowtie$ *cliente*, *nombre_cliente* de *impositor* es una clave externa que hace referencia a *cliente*; por tanto, el tamaño del resultado es exactamente $n_{impositor}$, que es 5.000.

Calculemos ahora las estimaciones de tamaño de *impositor* $\bowtie$ *cliente* sin usar la información sobre las claves externas. Como V(*nombre_cliente, impositor*) $= 2.500$ y V(*nombre_cliente, cliente*) $= 10.000$, las dos estimaciones que se consiguen son $5.000 * 10.000/2.500 = 20.000$ y $5.000 * 10.000/10.000 = 5.000$, y se escoge la menor. En este caso, la menor de las estimaciones es igual que la que se calculó anteriormente a partir de la información sobre las claves externas.

14.3.4 Estimación del tamaño de otras operaciones

A continuación se esbozará el modo de estimar el tamaño de los resultados de otras operaciones del álgebra relacional.

- **Proyección:** El tamaño estimado (número de registros de las tuplas) de una proyección de la forma $\Pi_A(r)$ es $V(A, r)$, ya que la proyección elimina los duplicados.
- **Agregación:** El tamaño de ${}_A\mathcal{G}_F(r)$ es simplemente $V(A, r)$, ya que hay una tupla de ${}_A\mathcal{G}_F(r)$ por cada valor distinto de A.
- **Operaciones de conjuntos:** Si las dos entradas de una operación de conjuntos son selecciones de la misma relación se puede reescribir la operación de conjuntos como disyunciones, conjunciones o negaciones. Por ejemplo, $\sigma_{\theta_1}(r) \cup \sigma_{\theta_2}(r)$ puede reescribirse como $\sigma_{\theta_1 \vee \theta_2}(r)$. De manera parecida, las intersecciones se pueden reescribir como conjunciones y la diferencia de conjuntos empleando la negación, siempre que las dos relaciones que participan en la operación de conjuntos sean selecciones de la misma relación. Luego se pueden usar las estimaciones de las selecciones que impliquen conjunciones, disyunciones y negaciones del Apartado 14.3.2.

  Si las entradas no son selecciones de la misma relación se estiman los tamaños de esta manera: el tamaño estimado de $r \cup s$ es la suma de los tamaños de r y de s. El tamaño estimado de $r \cap s$ es el mínimo de los tamaños de r y de s. El tamaño estimado de $r - s$ es el mismo tamaño de r. Las tres estimaciones pueden ser imprecisas, pero proporcionan cotas superiores para los tamaños.
- **Reunión externa:** El tamaño estimado de $r \mathbin{⟕} s$ es el tamaño de $r \bowtie s$ más el tamaño de r; el de $r \mathbin{⟖} s$ es simétrico, mientras que el de $r \mathbin{⟗} s$ es el tamaño de $r \bowtie s$ más los tamaños de r y s. Las tres estimaciones pueden ser imprecisas, pero proporcionan cotas superiores para los tamaños.

14.3.5 Estimación del número de valores distintos

Para las selecciones el número de valores distintos de un atributo (o de un conjunto de atributos) A en el resultado de una selección, $V(A, \sigma_\theta(r))$, puede estimarse de la manera siguiente:

- Si la condición de selección θ obliga a que A adopte un valor especificado (por ejemplo, $A = 3$), $V(A, \sigma_\theta(r)) = 1$.
- Si θ obliga a que A adopte un valor de entre un conjunto especificado de valores (por ejemplo, $(A = 1 \vee A = 3 \vee A = 4)$), entonces $V(A, \sigma_\theta(r))$ se define como el número de valores especificados.
- Si la condición de selección θ es de la forma $Aopv$, donde op es un operador de comparación, $V(A, \sigma_\theta(r))$ se estima que es $V(A, r) * s$, donde s es la selectividad de la selección.
- En todos los demás casos de selecciones se da por supuesto que la distribución de los valores de A es independiente de la distribución de los valores para los que se especifican las condiciones de selección y se usa una estimación aproximada de min $\min(V(A, r), n_{\sigma_\theta(r)})$. Se puede obtener una estimación más precisa para este caso usando la teoría de la probabilidad, pero la aproximación anterior funciona bastante bien.

Para las reuniones el número de valores distintos de un atributo (o de un conjunto de atributos) A en el resultado de una reunión, $V(A, r \bowtie s)$, puede estimarse de la manera siguiente:

- Si todos los atributos de A proceden de r, $V(A, r \bowtie s)$ se estima como $\min(V(A, r), n_{r \bowtie s})$, y de manera parecida si todos los atributos de A proceden de s, $V(A, r \bowtie s)$ se estima que es $\min(V(A, s), n_{r \bowtie s})$.
- Si A contiene atributos $A1$ de r y $A2$ de s, entonces $V(A, r \bowtie s)$ se estima como

$$\min(V(A1, r) * V(A2 - A1, s), V(A1 - A2, r) * V(A2, s), n_{r \bowtie s})$$

  Obsérvese que algunos atributos pueden estar en $A1$ y en $A2$, y que $A1 - A2$ y $A2 - A1$ denotan, respectivamente, a los atributos de A que sólo proceden de r y a los atributos de A que sólo proceden de s. Nuevamente, se pueden obtener estimaciones más precisas usando la teoría de la probabilidad, pero las aproximaciones anteriores funcionan bastante bien.

Las estimaciones de los distintos valores son directas para las proyecciones: son iguales en $\Pi_A(r)$ que en r. Lo mismo resulta válido para los atributos de agrupación de las agregaciones. Para los resultados de **suma**, **cuenta** y **promedio**, se puede suponer, por simplificar, que todos los valores agregados son distintos. Para **min**(A) y **max**(A), el número de valores distintos puede estimarse como $\min(V(A, r), V(G, r))$, donde G denota los atributos de agrupamiento. Se omiten los detalles de la estimación de los valores distintos para otras operaciones.

14.4 Elección de los planes de evaluación

La generación de expresiones sólo es una parte del proceso de optimización de consultas, ya que cada operación de la expresión puede implementarse con algoritmos diferentes. Por tanto, se necesita un plan de evaluación para definir exactamente el algoritmo que se usará para cada operación y el modo en que se coordinará la ejecución de las operaciones. La Figura 14.5 muestra un plan de evaluación posible para la expresión de la Figura 14.3. Como ya se ha visto, se pueden emplear varios algoritmos diferentes para cada operación relacional, lo que da lugar a planes de evaluación alternativos. Además, hay que tomar decisiones sobre el encauzamiento. En esta figura, los trazos de las operaciones de selección hasta la operación reunión mezcla están marcados como encauzados; el encauzamiento es factible si las operaciones de selección generan sus resultados ordenados según los atributos de reunión. Lo harán si los índices de *sucursal* y *cuenta* almacenan los registros con valores iguales de los atributos de índice ordenados por *nombre_sucursal*.

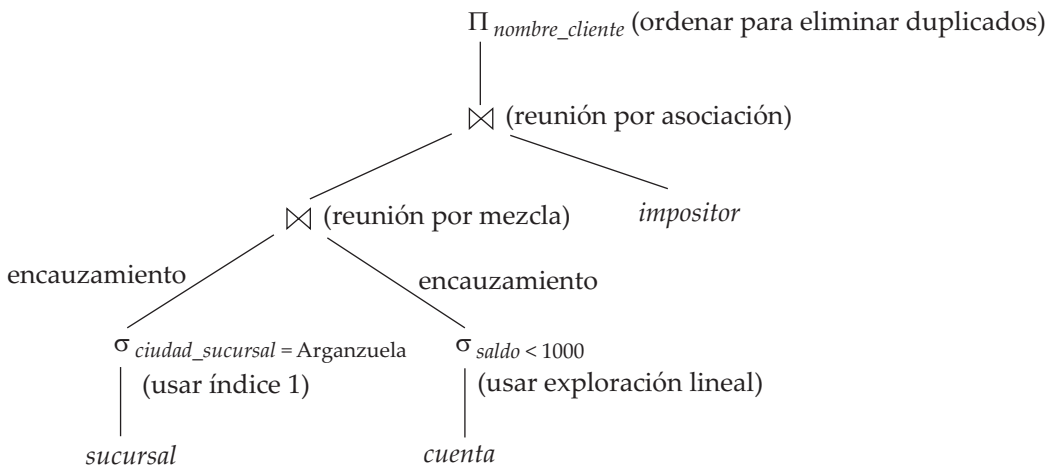


Figura 14.5 Un plan de evaluación.

14.4.1 Interacción de las técnicas de evaluación

Una manera de escoger un plan de evaluación para una expresión de consulta es sencillamente escoger el algoritmo más económico para evaluar cada operación. Se puede escoger cualquier ordenación de las operaciones que asegure que las operaciones ubicadas por debajo en el árbol se ejecuten antes que las operaciones situadas más arriba.

Sin embargo, la selección del algoritmo más económico para cada operación no es necesariamente una buena idea. Aunque puede que una reunión por mezcla en un nivel dado resulte más costosa que una reunión por asociación, puede que proporcione un resultado ordenado que haga más económica la evaluación de operaciones posteriores (como la eliminación de duplicados, la intersección u otra reunión por asociación). De manera parecida, puede que una reunión en bucle anidado indexada proporcione oportunidades para el encauzamiento de los resultados de la operación siguiente y, por tanto, puede que resulte útil aunque no sea la manera más económica de llevar a cabo la reunión. Para escoger el mejor algoritmo global hay que considerar incluso los algoritmos no óptimos para cada una de las operaciones.

Por tanto, además de considerar las expresiones alternativas de cada consulta, también hay que considerar los algoritmos alternativos para cada operación de cada expresión. Se pueden usar reglas muy parecidas a las reglas de equivalencia para definir los algoritmos que pueden usarse para cada operación, y si su resultado puede encauzarse o se debe materializar. Se pueden usar estas reglas para generar todos los planes de evaluación de consultas para una expresión dada.

Dado un plan de evaluación, se puede estimar su coste empleando las estadísticas estimadas mediante las técnicas del Apartado 14.3 junto con las estimaciones de costes de varios algoritmos y métodos de evaluación descritos en el Capítulo 13. Dependiendo de los índices disponibles, ciertas operaciones selección se pueden evaluar sólo con un índice y sin acceder a la relación en sí. Eso sigue dejando el problema de la selección del mejor plan de evaluación de la consulta. Hay dos enfoques generales: el primero busca todos los planes y escoge el mejor de una manera basada en los costes. El segundo usa la heurística para escoger el plan. A continuación se estudiarán los dos enfoques. Los optimizadores de consultas prácticos incorporan elementos de ambos enfoques.

14.4.2 Optimización basada en el coste

Los **optimizadores basados en el coste** generan una gama de planes de evaluación a partir de la consulta dada empleando las reglas de equivalencia y escogen el de coste mínimo. Para las consultas complejas el número de planes de consulta diferentes que son equivalentes a un plan dado puede ser grande. A modo de ejemplo, considérese la expresión

$$r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

en la que las reuniones se expresan sin ninguna ordenación. Con $n = 3$, hay 12 ordenaciones diferentes de la mezcla:

$$\begin{array}{llll} r_1 \bowtie (r_2 \bowtie r_3) & r_1 \bowtie (r_3 \bowtie r_2) & (r_2 \bowtie r_3) \bowtie r_1 & (r_3 \bowtie r_2) \bowtie r_1 \\ r_2 \bowtie (r_1 \bowtie r_3) & r_2 \bowtie (r_3 \bowtie r_1) & (r_1 \bowtie r_3) \bowtie r_2 & (r_3 \bowtie r_1) \bowtie r_2 \\ r_3 \bowtie (r_1 \bowtie r_2) & r_3 \bowtie (r_2 \bowtie r_1) & (r_1 \bowtie r_2) \bowtie r_3 & (r_2 \bowtie r_1) \bowtie r_3 \end{array}$$

En general, con n relaciones, hay $(2(n-1))!/(n-1)!$ órdenes de reunión diferentes (se deja el cálculo de esta expresión al lector en el Ejercicio 14.10). Para las reuniones que implican números pequeños de relaciones, este número resulta aceptable; por ejemplo, con $n = 5$, el número es 1.680. Sin embargo, a medida que n se incrementa, este número crece rápidamente. Con $n = 7$, el número es 665.280; con $n = 10$, ¡el número es mayor de 17.600 millones!

Afortunadamente, no es necesario generar todas las expresiones equivalentes a la expresión dada. Por ejemplo, supóngase que se desea hallar el mejor orden de reunión de la forma

$$(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$$

que representa todos los órdenes de reunión en que r_1, r_2 y r_3 se reúnen primero (en algún orden), y el resultado se reúne (en algún orden) con r_4 and r_5. Hay doce órdenes de reunión diferentes para calcular $r_1 \bowtie r_2 \bowtie r_3$, y otros doce órdenes para calcular la reunión de este resultado con r_4 y r_5. Por tanto, parece que hay 144 órdenes de reunión que examinar. Sin embargo, una vez hallado el mejor orden de reunión para el subconjunto de relaciones $\{r_1, r_2, r_3\}$, se puede usar ese orden para las reuniones posteriores con r_4 and r_5, y se pueden ignorar todos los órdenes de reunión más costosos de $r_1 \bowtie r_2 \bowtie r_3$. Por lo que, en lugar de examinar 144 opciones, sólo hace falta examinar 12 + 12 opciones.

Usando esta idea se puede desarrollar un *algoritmo de programación dinámica* para hallar los órdenes de reunión óptimos. Los algoritmos de programación dinámica almacenan los resultados de los cálculos y los vuelven a usar, un procedimiento que puede reducir enormemente el tiempo de ejecución. En la Figura 14.6 aparece un procedimiento recursivo que implementa el algoritmo de programación dinámica.

El procedimiento almacena los planes de evaluación que calcula en el array asociado $mejorplan$, que está indexado por conjuntos de relaciones. Cada elemento del array asociativo contiene dos componentes: el coste del mejor plan de S y el propio plan. El valor de $mejorplan[S].coste$ se supone que se inicializa como ∞ si $mejorplan[S]$ no se ha calculado todavía.

El procedimiento comprueba en primer lugar si el mejor plan para calcular la reunión del conjunto de relaciones dado S se ha calculado ya (y se ha almacenado en el array asociativa $mejorplan$); si es así, devuelve el plan ya calculado.

Si S contiene sólo una relación, se registra en $mejorplan$ la mejor forma de acceder a S (teniendo en cuenta las selecciones sobre S, si las hay). Esto puede implicar el uso de un índice para identificar las

```
procedure HallarMejorPlan(S)
    if (mejorplan[S].coste ≠ ∞) /* mejorplan[S] ya se ha calculado */
        return mejorplan[S]
    if (S contiene sólo una relación)
        establecer mejorplan[S].plan y mejorplan[S].coste en función de la mejor manera de acceder a S
    else for each subconjunto no vacío S1 de S tal que S1 ≠ S
        P1 = HallarMejorPlan(S1)
        P2 = HallarMejorPlan(S − S1)
        A = mejor algoritmo para reunir los resultados de P1 y de P2
        coste = P1.coste + P2.coste + coste de A
        if coste < mejorplan[S].coste
            mejorplan[S].coste = coste
            mejorplan[S].plan = "ejecutar P1.plan; ejecutar P2.plan;
                                 reunir resultados de P1 y P2 usando A"
    return mejorplan[S]
```

Figura 14.6 Algoritmo de programación dinámica para la optimización del orden de la reunión.

tuplas, y después buscar las tuplas (conocido frecuentemente como *exploración del índice*), o explorar la relación completa (conocido frecuentemente como *exploración de la relación*)[1].

En caso contrario, el procedimiento prueba todas las manerasposibles de dividir S en dos subconjuntos disjuntos. Para cada división el procedimiento halla de manera recursiva los mejores planes para cada uno de los dos subconjuntos y luego calcula el coste del plan global usando esa división. El procedimiento escoge el plan más económico de entre todas las alternativas para dividir S en dos conjuntos. El procedimiento almacena el plan más económico y su coste en el array $mejorplan$ y los devuelve. La complejidad temporal del procedimiento puede demostrarse que es $O(3^n)$ (véase el Ejercicio práctico 14.8).

En realidad, el orden en que la reunión de un conjunto de relaciones genera las tuplas también es importante para hallar el mejor orden global de reunión, ya que puede afectar al coste de las reuniones posteriores (por ejemplo, si se usa una mezcla). Se dice que un orden determinado de las tuplas es un **orden interesante** si puede resultar útil para alguna operación posterior. Por ejemplo, la generación del resultado de $r_1 \bowtie r_2 \bowtie r_3$ ordenado según los atributos comunes con r_4 o r_5 puede resultar útil, pero generarlo ordenado según los atributos comunes solamente con r_1 y r_2 no resulta útil. El uso de la reunión por mezcla para calcular $r_1 \bowtie r_2 \bowtie r_3$ puede resultar más costoso que emplear algún otro tipo de técnica de reunión, pero puede que proporcione un resultado ordenado según un orden interesante.

Por tanto, no basta con hallar el mejor orden de reunión para cada subconjunto del conjunto de n relaciones dadas. Por el contrario, hay que hallar el mejor orden de reunión para cada subconjunto para cada orden interesante de la reunión resultante para ese subconjunto. El número de subconjuntos de n relaciones es 2^n. El número de criterios de ordenación interesantes no suele ser grande. Así, hay que almacenar alrededor de 2^n expresiones de reunión. El algoritmo de programación dinámica para hallar el mejor orden de reunión puede extenderse de manera sencilla para que trabaje con los criterios de ordenación. El coste del algoritmo extendido depende del número de órdenes interesantes para cada subconjunto de relaciones; dado que se ha hallado que este número en la práctica es pequeño, el coste se queda en $O(3^n)$. Con $n = 10$ este número es de alrededor de 59.000, que es mucho mejor que los 17.600 millones de órdenes de reunión diferentes. Y lo que es más importante, el almacenamiento necesario es mucho menor que antes, ya que sólo hace falta almacenar un orden de reunión por cada orden interesante de cada uno de los 1024 subconjuntos de $r_1, \ldots, r_{10}$. Aunque los dos números siguen creciendo rápidamente con n, las reuniones que se producen con frecuencia suelen tener menos de diez relaciones y pueden manejarse con facilidad.

Se pueden usar varias técnicas para reducir aún más el coste de la búsqueda entre un gran número de planes. Por ejemplo, al examinar los planes para una expresión se puede concluir tras examinar sólo una parte de la expresión si se determina que el plan más económico para esa parte ya resulta más costoso que el plan de evaluación más económico para una expresión completa examinada anteriormente. De manera parecida, supóngase que se determina que la manera más económica de evaluar una subexpresión es más costosa que el plan de evaluación más económico para una expresión completa examinada anteriormente. Entonces, no hace falta examinar ninguna expresión completa que incluya esa subexpresión. Se puede reducir aún más el número de planes de evaluación que hay que considerar completamente llevando a cabo antes una selección heurística de un buen plan y estimando el coste de ese plan. Luego, sólo unos pocos planes competidores necesitarán un análisis completo de los costes. Estas optimizaciones pueden reducir de manera significativa la sobrecarga de la optimización de consultas.

Las complejidades de SQL introducen complicaciones para los optimizadores de consultas. El tratamiento de las consultas anidadas se describe brevemente en el Apartado 14.4.4.

La optimización descrita anteriormente se concentra en la optimización del orden de la reunión. En cambio, los optimizadores usados en algunos otros sistemas, principalmente en SQL Server de Microsoft, se basan en reglas de equivalencia. La ventaja de usar reglas de equivalencia es que se puede ampliar fácilmente el optimizador con nuevas reglas. Por ejemplo, las consultas anidadas se pueden representar usando constructoras del álgebra relacional extendida, y las transformaciones de las consultas anidadas se pueden expresar como reglas de equivalencia. Para que este enfoque sea eficaz se requieren técnicas

1. Si un índice contiene todos los atributos de la relación que se usan en la consulta, es posible realizar una *exploración sólo del índice* que recupera los valores de atributo requeridos del índice sin capturar las tuplas reales.

eficientes para detectar las derivaciones de duplicados, y una forma de programación dinámica para evitar optimizar de nuevo las mismas subexpresiones. Este enfoque se propuso en el proyecto de investigación Volcano. Véanse las notas bibliográficas para consultar referencias con más información.

14.4.3 Heurísticas de optimización

Un inconveniente de la optimización basada en el coste es el coste de la propia optimización. Aunque el coste de la optimización de las consultas puede reducirse mediante algoritmos inteligentes, el número de planes de evaluación distintos para una consulta puede ser muy grande y buscar el plan óptimo a partir de este conjunto requiere un gran esfuerzo de cómputo. Por ello, muchos sistemas usan la **heurística** para reducir el coste de la optimización.

Un ejemplo de regla heurística es la siguiente regla para la transformación de consultas del álgebra relacional:

- Realizar las operaciones de selección tan pronto como sea posible.

Los optimizadores heurísticos usan esta regla sin averiguar si se reduce el coste mediante esta transformación. En el primer ejemplo de transformación del Apartado 14.2 se forzó la operación selección en una reunión.

Se dice que la regla anterior es heurística porque suele ayudar a reducir el coste, aunque no lo haga siempre. Como ejemplo de dónde puede dar lugar a un incremento del coste, considérese una expresión $\sigma_\theta(r \bowtie s)$, donde la condición θ sólo hace referencia a atributos de s. Ciertamente, la selección puede llevarse a cabo antes de la reunión. Sin embargo, si r es tremendamente pequeña comparada con s, y si hay un índice basado en los atributos de reunión de s pero no hay ningún índice basado en los atributos usados por θ, probablemente resulte una mala idea llevar a cabo la selección pronto. Llevar a cabo pronto la selección (es decir, directamente sobre s) exigiría hacer una exploración de todas las tuplas de s. Es posible que resulte más económico calcular la reunión usando el índice y luego rechazar las tuplas que no superen la selección.

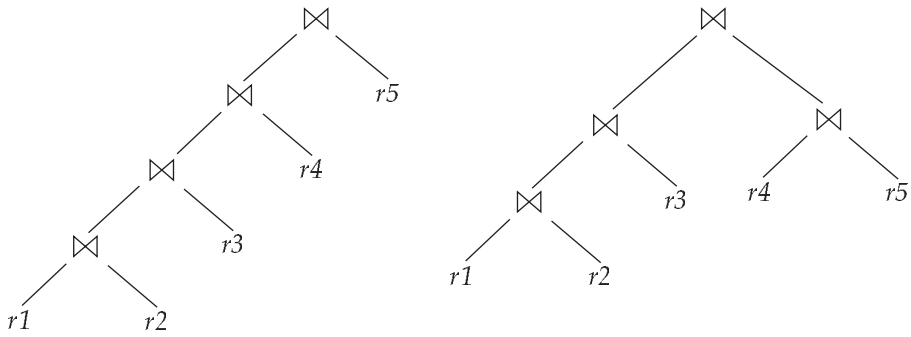
La operación proyección, como la operación selección, reduce el tamaño de las relaciones. Por tanto, siempre que haya que generar una relación temporal, resulta ventajoso aplicar inmediatamente cuantas proyecciones sea posible. Esta ventaja sugiere un acompañante a la heurística "realizar las selecciones tan pronto como sea posible":

- Realizar las proyecciones tan pronto como sea posible.

Suele resultar mejor llevar a cabo las selecciones antes que las proyecciones, ya que las selecciones tienen la posibilidad de reducir mucho el tamaño de las relaciones y permiten el empleo de índices para tener acceso a las tuplas. Un ejemplo parecido al usado para la heurística de selección debería convencer al lector de que esta heurística no siempre reduce el coste.

La mayoría de optimizadores de consultas tienen más heurísticas para reducir el coste de la optimización. Por ejemplo, muchos optimizadores de consultas, como el optimizador de System R, no toman en consideración todos los órdenes de reunión, sino que restringen la búsqueda a tipos concretos de órdenes de reunión. El optimizador de System R sólo toma en consideración los órdenes de reunión en que el operando de la derecha de cada reunión es una de las relaciones iniciales $r_1, \ldots, r_n$. Estos órdenes de reunión se denominan **órdenes de reunión en profundidad por la izquierda**. Los órdenes de reunión en profundidad por la izquierda resultan especialmente convenientes para la evaluación encauzada, ya que el operando de la derecha es una relación almacenada y, así, sólo se encauza una entrada por cada reunión.

La Figura 14.7 muestra la diferencia entre un árbol de reunión en profundidad por la izquierda y otro que no lo es. El tiempo que se tarda en tomar en consideración todos los órdenes de reunión en profundidad por la izquierda es $O(n!)$, que es mucho menor que el tiempo necesario para tomar en consideración todos los órdenes de reunión. Con el empleo de las optimizaciones de programación dinámica, el optimizador de System R puede hallar el mejor orden de reunión en un tiempo de $O(n2^n)$. Compárese este coste con el tiempo de $O(3^n)$ necesario para hallar el mejor orden de reunión global.



(a) Árbol de reunión en profundidad por la izquierda

(b) Árbol de reunión que no es en profundidad por la izquierda

Figura 14.7 Árboles de reunión en profundidad por la izquierda.

El optimizador System R usa la heurística para forzar el desplazamiento de las selecciones y de las proyecciones hacia la parte inferior del árbol de consultas.

El enfoque heurístico para reducir el coste de la selección del orden de reunión, que se usó originalmente en algunas versiones de Oracle, funciona básicamente de esta manera: para cada reunión de grado n toma en consideración n planes de evaluación. Cada plan usa un orden de reunión en profundidad por la izquierda, comenzando con una relación diferente de las n existentes. La heurística crea el orden de reunión para cada uno de los n planes de evaluación seleccionando de manera repetida la "mejor" relación que reunir a continuación, con base en la clasificación de los caminos de acceso disponibles. Se escoge la reunión en bucle anidado o mezcla-ordenación para cada una de las reuniones, en función de los caminos de acceso disponibles. Finalmente, la heurística escoge uno de los n planes de evaluación de manera heurística, basada en la minimización del número de reuniones de bucle anidado que no tienen disponible un índice para la relación interna y en el número de reuniones por mezcla-ordenación.

Los enfoques de la optimización de consultas que integran la selección heurística y la generación de planes de acceso alternativos se han adoptado en varios sistemas. El enfoque usado en System R y en su sucesor, el proyecto Starburst, es un procedimiento jerárquico basado en el concepto de bloques anidados de SQL. Las técnicas de optimización basadas en costes aquí descritas se usan por separado para cada bloque de la consulta. Los optimizadores de varios productos de bases de datos, como DB2 de IBM y Oracle, se basan en este enfoque, con extensiones para tratar otras operaciones como la agregación. Para las consultas compuestas de SQL (que usan la operación $\cup$, $\cap$ o $-$), el optimizador procesa cada componente por separado y combina los planes de evaluación para formar el plan global de evaluación.

Muchas aplicaciones ejecutan la misma consulta repetidamente pero con diferentes valores de las constantes. Por ejemplo, una aplicación bancaria puede ejecutar repetidamente una consulta para hallar las transacciones recientes sobre una cuenta, pero con diferentes valores del número de cuenta. Como heurística, muchos optimizadores optimizan una vez la consulta con los valores de la primera vez que se emitió, y almacenan en caché el plan de la consulta. Cada vez que se ejecuta de nuevo la consulta, quizás con nuevos valores de las constantes, el plan de la consulta en caché se reutiliza (obviamente usando los nuevos valores de las constantes). El plan óptimo para las nuevas constantes puede diferir del plan óptimo para los valores iniciales, pero como heurística se reutiliza el plan guardado en la caché.

Incluso con el uso de la heurística la optimización de consultas basada en los costes impone una sobrecarga sustancial al procesamiento de las consultas. No obstante, el coste añadido de la optimización de las consultas basada en los costes suele compensarse con creces por el ahorro en tiempo de ejecución de la consulta, que queda dominado por los accesos lentos a los discos. La diferencia en el tiempo de ejecución entre un buen plan y uno malo puede ser enorme, lo cual hace que la optimización de las consultas sea esencial. El ahorro conseguido se multiplica en las aplicaciones que se ejecutan de manera regular, en las que se puede optimizar la consulta una sola vez y usarse el plan de consultas seleccionado cada vez que se ejecute la consulta. Por tanto, la mayor parte de los sistemas comerciales incluyen

optimizadores relativamente sofisticados. Las notas bibliográficas dan referencias de las descripciones de los optimizadores de consultas de los sistemas de bases de datos reales.

14.4.4 Optimización de subconsultas anidadas**

SQL trata conceptualmente a las subconsultas anidadas de la cláusula **where** como funciones que toman parámetros y devuelven un solo valor o un conjunto de valores (quizás, un conjunto vacío). Los parámetros son las variables de la consulta del nivel externo que se usan en la subconsulta anidada (estas variables se denominan **variables de correlación**). Por ejemplo, supóngase que se tiene la consulta siguiente:

```
select nombre_cliente
from prestatario
where exists (select *
              from impositor
              where impositor.nombre_cliente = prestatario.nombre_cliente)
```

Conceptualmente, la subconsulta puede considerarse como una función que toma un parámetro (aquí, *prestatario.nombre_cliente*) y devuelve el conjunto de todos los impositores con el mismo nombre.

SQL evalúa la consulta global (conceptualmente) calculando el producto cartesiano de las relaciones de la cláusula **from** externa y comprobando luego los predicados de la cláusula **where** para cada tupla del producto. En el ejemplo anterior, el predicado comprueba si el resultado de la evaluación de la subconsulta está vacío.

Esta técnica para evaluar una consulta con una subconsulta anidada se denomina **evaluación correlacionada**. La evaluación correlacionada no resulta muy eficiente, ya que la subconsulta se evalúa por separado para cada tupla de la consulta del nivel externo. Puede dar lugar a gran número de operaciones aleatorias de E/S a disco.

Por tanto, los optimizadores de SQL intentan transformar las subconsultas anidadas en reuniones, siempre que resulta posible. Los algoritmos de reunión eficientes evitan las costosas operaciones aleatorias de E/S. Cuando la transformación no resulta posible el optimizador conserva las subconsultas como expresiones independientes, las optimiza por separado y luego las evalúa mediante la evaluación correlacionada.

Como ejemplo de transformación de una subconsulta anidada en una reunión, la consulta del ejemplo anterior puede reescribirse como

```
select nombre_cliente
from prestatario, impositor
where impositor.nombre_cliente = prestatario.nombre_cliente
```

Para reflejar correctamente la semántica de SQL no debe cambiar el número de derivaciones duplicadas debido a la reescritura; la consulta reescrita puede modificarse para asegurarse de que se cumple esta propiedad, como se verá en breve.

En el ejemplo la subconsulta anidada era muy sencilla. En general, puede que no resulte posible desplazar las relaciones de la subconsulta anidada a la cláusula **from** de la consulta externa. En lugar de eso, se crea una relación temporal que contiene el resultado de la consulta anidada *sin* las selecciones empleando las variables de correlación de la consulta externa y se reúne la tabla temporal con la consulta del nivel exterior. Por ejemplo, una consulta de la forma

```
select ...
from L1
where P1 and exists (select *
                     from L2
                     where P2)
```

donde P_2 es una conjunción de predicados más sencillos, puede reescribirse como

create table t_1 **as**
  select distinct V
    from L_2
    where P_2^1
select ...
from L_1, t_1
where P_1 **and** P_2^2

donde P_2^1 contiene los predicados de P_2 sin las selecciones que implican las variables de correlación, y P_2^2 reintroduce las selecciones que implican las variables de correlación (con las relaciones a que se hace referencia en el predicado renombradas adecuadamente). Aquí V contiene todos los atributos que se usan en las selecciones con las variables de correlación en la subconsulta anidada.

En el ejemplo la consulta original se habría transformado en:

create table t_1 **as**
  select distinct *nombre_cliente*
  from *impositor*
select *nombre_cliente*
from *prestatario*, t_1
where t_1.*nombre_cliente* = *prestatario.nombre_cliente*

La consulta que se reescribió para mostrar la creación de relaciones temporales puede obtenerse simplificando la consulta transformada más arriba, suponiendo que el número de duplicados de cada tupla no importa.

El proceso de sustituir una consulta anidada por una consulta con una reunión (acaso con una relación temporal) se denomina **descorrelación**.

La descorrelación resulta más complicada cuando la subconsulta anidada usa la agregación o cuando el resultado de la subconsulta anidada se usa para comprobar la igualdad, o cuando la condición que enlaza la subconsulta anidada con la consulta exterior es **not exists**, etc. No se intentará dar algoritmos para el caso general y, en vez de eso, se remitirá al lector a los elementos de importancia en las notas bibliográficas.

La optimización de las subconsultas anidadas complejas es una labor difícil, como puede deducirse del estudio anterior, y muchos optimizadores sólo llevan a cabo una cantidad limitada de descorrelación. Resulta más conveniente evitar el empleo de subconsultas anidadas complejas, siempre que sea posible, ya que no se puede asegurar que el optimizador de consultas tenga éxito en su conversión a una forma que pueda evaluarse de manera eficiente.

14.5 Vistas materializadas**

Cuando se define una vista, normalmente la base de datos sólo almacena la consulta que define la vista. Por el contrario, una **vista materializada** es una vista cuyo contenido se calcula y se almacena. Las vistas materializadas constituyen datos redundantes, en el sentido de que su contenido puede deducirse de la definición de la vista y del resto del contenido de la base de datos. No obstante, resulta mucho más económico en muchos casos leer el contenido de una vista materializada que calcular el contenido de la vista ejecutando la consulta que la define.

Las vistas materializadas resultan importantes para la mejora del rendimiento de algunas aplicaciones. Considérese esta vista, que da el importe total de los préstamos de cada sucursal:

create view *total_préstamos_sucursal*(*nombre_sucursal*, *total_préstamos*) **as**
select *nombre_sucursal*, **sum**(*importe*)
from *préstamos*
group by *nombre_sucursal*

Supóngase que el importe total de los préstamos de la sucursal se solicita con frecuencia (antes de conceder un nuevo préstamo, por ejemplo). El cálculo de la vista exige la lectura de cada tupla de *préstamos* correspondiente a la sucursal y sumar los importes de los préstamos, lo que puede llevar mucho tiempo.

En cambio, si la definición de la vista del importe total de los préstamos estuviera materializada, el importe total de los préstamos podría hallarse buscando una sola tupla de la vista materializada.

14.5.1 Mantenimiento de las vistas

Un problema con las vistas materializadas es que hay que mantenerlas actualizadas cuando se modifican los datos empleados en la definición de la vista. Por ejemplo, si se actualiza el valor *importe* de un préstamo, la vista materializada se vuelve inconsistente con los datos subyacentes y hay que actualizarla. La tarea de mantener actualizada una vista materializada con los datos subyacentes se denomina **mantenimiento de la vista**.

Las vistas pueden mantenerse mediante código escrito a mano: es decir, cada fragmento del código que actualiza el valor *importe* del préstamo puede modificarse para actualizar el importe total de los préstamos de la sucursal correspondiente.

Otra opción para el mantenimiento de las vistas materializadas es la definición de desencadenadores para la inserción, el borrado y la actualización de cada relación de la definición de la vista. Los desencadenadores deben modificar el contenido de la vista materializada para tener en cuenta el cambio que ha provocado que se active el desencadenador. Una manera sencilla de hacerlo es volver a calcular completamente la vista materializada con cada actualización.

Una opción mejor consiste en modificar sólo las partes afectadas de la vista materializada, lo que se conoce como **mantenimiento incremental de la vista**. En el Apartado 14.5.2 se describe la manera de llevar a cabo el mantenimiento incremental de la vista.

Los sistemas modernos de bases de datos proporcionan más soporte directo para el mantenimiento incremental de las vistas. Los programadores de bases de datos ya no necesitan definir disparadores para el mantenimiento de las vistas. Por el contrario, una vez que se ha declarado materializada una vista, el sistema de bases de datos calcula su contenido y actualiza de manera incremental el contenido cuando se modifican los datos subyacentes.

La mayoría de sistemas de bases de datos realizan un **mantenimiento inmediato de las vistas**; es decir, se realiza un mantenimiento incremental tan pronto como sucede una actualización como parte de una transacción de actualización. Algunos sistemas de bases de datos también permiten el **mantenimiento diferido de las vistas**, donde se difiere el mantenimiento a un momento posterior; por ejemplo, las actualizaciones se pueden recopilar a lo largo del día y las vistas materializadas se pueden actualizar durante la noche. Este enfoque reduce la sobrecarga en las transacciones de actualización. Sin embargo, es posible que las vistas materializadas con mantenimiento diferido no sean consistentes con las relaciones sobre las que están definidas.

14.5.2 Mantenimiento incremental de las vistas

Para comprender el modo de mantener de manera incremental las vistas materializadas se comenzará por considerar las operaciones individuales y luego se verá la manera de manejar una expresión completa.

Los cambios de cada relación que puedan hacer que se quede desactualizada una vista materializada son las inserciones, los borrados y las actualizaciones. Para simplificar la descripción se sustituyen las actualizaciones a cada tupla por el borrado de esa tupla seguida de la inserción de la tupla actualizada. Por tanto, sólo hay que considerar las inserciones y los borrados. Los cambios (inserciones y borrados) en la relación o en la expresión se denominan su **diferencial**.

14.5.2.1 La operación reunión

Considérese la vista materializada $v = r \bowtie s$. Supóngase que se modifica r insertando un conjunto de tuplas denotado por i_r. Si el valor antiguo de r se denota por r^{vieja} y el valor nuevo de r por r^{nueva}, $r^{nueva} = r^{vieja} \cup i_r$. Ahora bien, el valor antiguo de la vista, v^{vieja} viene dado por $r^{vieja} \bowtie s$, y el valor nuevo v^{nueva} viene dado por $r^{nueva} \bowtie s$. Se puede reescribir $r^{nueva} \bowtie s$ como $(r^{vieja} \cup i_r) \bowtie s$, lo que se

puede reescribir una vez más como $(r^{vieja} \bowtie s) \cup (i_r \bowtie s)$. En otras palabras,

$$v^{nueva} = v^{vieja} \cup (i_r \bowtie s)$$

Por tanto, para actualizar la vista materializada v, sólo hace falta añadir las tuplas $i_r \bowtie s$ al contenido antiguo de la vista materializada. Las inserciones en s se manejan de una manera completamente simétrica.

Supóngase ahora que se modifica r borrando un conjunto de tuplas denotado por d_r. Usando el mismo razonamiento que anteriormente se obtiene

$$v^{nueva} = v^{vieja} - (d_r \bowtie s)$$

Las operaciones de borrado en s se manejan de una manera completamente simétrica.

14.5.2.2 Las operaciones selección y proyección

Considérese una vista $v = \sigma_\theta(r)$. Si se modifica r insertando un conjunto de tuplas i_r, el nuevo valor de v puede calcularse como

$$v^{nueva} = v^{vieja} \cup \sigma_\theta(i_r)$$

De manera parecida, si se modifica r borrando un conjunto de tuplas e_r, el valor nuevo de v puede calcularse como

$$v^{nueva} = v^{vieja} - \sigma_\theta(e_r)$$

La proyección es una operación más difícil de tratar. Hay que considerar una vista materializada $v = \Pi_A(r)$. Supóngase que la relación r está en el esquema $R = (A, B)$ y que r contiene dos tuplas, $(a, 2)$ y $(a, 3)$. Entonces, $\Pi_A(r)$ tiene una sola tupla, (a). Si se borra la tupla $(a, 2)$ de r, no se puede borrar la tupla (a) de $\Pi_A(r)$: si se hiciera, el resultado sería una relación vacía, mientras que en realidad $\Pi_A(r)$ sigue teniendo una tupla ((a)). El motivo es que la misma tupla ((a)) se obtiene de dos maneras, y que el borrado de una tupla de r sólo borra una de las formas de obtener (a); la otra sigue presente.

Este motivo también ofrece una pista de la solución: para cada tupla de una proyección como $\Pi_A(r)$, se lleva la cuenta del número de veces que se ha obtenido.

Cuando se borra un conjunto de tuplas d_r de r, para cada tupla t de d_r hay que hacer lo siguiente. $t.A$ denota la proyección de t sobre el atributo A. Se busca $(t.A)$ en la vista materializada y se disminuye la cuenta almacenada con ella en 1. Si la cuenta llega a 0, se borra $(t.A)$ de la vista materializada.

El manejo de las inserciones resulta relativamente directo. Cuando un conjunto de tuplas i_r se inserta en r, para cada tupla t de i_r se hace lo siguiente. Si $(t.A)$ ya está presente en la vista materializada, se incrementa la cuenta almacenada con ella en 1. En caso contrario, se añade $(t.A)$ a la vista materializada con la cuenta puesta a 1.

14.5.2.3 Las operaciones de agregación

Las operaciones de agregación se comportan aproximadamente como las proyecciones. Las operaciones de agregación en SQL son **count, sum, avg, min** y **max**:

- **count**. Considérese la vista materializada $v =_A \mathcal{G}_{count(B)}(r)$, que calcula la cuenta del atributo B, después de agrupar r según el atributo A.

  Cuando se inserta un conjunto de tuplas i_r en r, para cada tupla t de i_r se realiza lo siguiente. Se busca el grupo $t.A$ en la vista materializada. Si no se halla presente, se añade $(t.A, 1)$ a la vista materializada. Si el grupo $t.A$ se halla presente, se añade 1 a su cuenta.

  Cuando un conjunto e_r se borra de r, para cada tupla t de e_r se realiza lo siguiente. Se busca el grupo $t.A$ en la vista materializada y se resta 1 de la cuenta del grupo. Si la cuenta llega a 0, se borra la tupla para el grupo $t.A$ de la vista materializada.

- **sum**. Considérese la vista materializada $v =_A \mathcal{G}_{sum(B)}(r)$.

  Cuando un conjunto de tuplas i_r se inserta en r, para cada tupla t de i_r se realiza lo siguiente. Se busca el grupo $t.A$ en la vista materializada. Si no se halla presente se añade $(t.A, t.B)$ a la vista materializada; además, se almacena una cuenta de 1 asociada con $(t.A, t.B)$, igual que se

hizo para las proyecciones. Si el grupo $t.A$ se halla presente, se añade el valor de $t.B$ al valor agregado para el grupo y se añade 1 a su cuenta.

Cuando se borra un conjunto de tuplas e_r de r, para cada tupla t de e_r se realiza lo siguiente. Se busca el grupo $t.A$ en la vista materializada y se resta $t.B$ del valor agregado para el grupo. También se resta 1 de la cuenta del grupo y, si la cuenta llega a 0, se borra la tupla para el grupo $t.A$ de la vista materializada.

Si no se guardara el valor de cuenta adicional no se podría distinguir el caso de que la suma para el grupo sea 0 del caso en que se ha borrado la última tupla de un grupo.

- **avg**. Considérese la vista materializada $v =_A \mathcal{G}_{avg(B)}(r)$.

  La actualización directa del promedio de una inserción o de un borrado no resulta posible, ya que no sólo depende del promedio antiguo y de la tupla que se inserta o borra, sin también del número de tuplas del grupo.

  En lugar de eso, para tratar el caso de **avg**, se conservan los valores de agregación **sum** y **count** como se describieron anteriormente y se calcula el promedio como la suma dividida por la cuenta.

- **min, max**. Considérese la vista materializada $v =_A \mathcal{G}_{min(B)}(r)$ (el caso de **max** es completamente equivalente).

  El tratamiento de las inserciones en r es inmediato. La conservación de los valores de agregación **min** and **max** para los borrados puede resultar más costoso. Por ejemplo, si la tupla correspondiente al valor mínimo para un grupo se borra de r, hay que examinar las demás tuplas de r que están en el mismo grupo para hallar el nuevo valor mínimo.

14.5.2.4 Otras operaciones

La operación de conjuntos *intersección* se conserva de la manera siguiente. Dada la vista materializada $v = r \cap s$, cuando una tupla se inserta en r se comprueba si está presente en s y, en caso afirmativo, se añade a v. Si se borra una tupla de r, se borra de la intersección si se halla presente. Las otras operaciones con conjuntos, *unión* y *diferencia de conjuntos*, se tratan de manera parecida; los detalles se dejan al lector.

Las reuniones externas se tratan de manera muy parecida a las reuniones, pero con algún trabajo adicional. En el caso del borrado de r hay que manejar las tuplas de s que ya no coinciden con ninguna tupla de r. En el caso de una inserción en r, hay que manejar las tuplas de s que no coincidían con ninguna tupla de r. De nuevo se dejan los detalles al lector.

14.5.2.5 Tratamiento de expresiones

Hasta ahora se ha visto el modo de actualizar de manera incremental el resultado de una sola operación. Para tratar una expresión entera se pueden obtener expresiones para el cálculo del cambio incremental en el resultado de cada subexpresión, comenzando por las de menor tamaño.

Por ejemplo, supóngase que se desea actualizar de manera incremental la vista materializada $E_1 \bowtie E_2$ cuando se inserta un conjunto de tuplas i_r en la relación r. Supóngase que r se usa sólo en E_1. Supóngase que el conjunto de tuplas que se va a insertar en E_1 viene dado por la expresión D_1. Entonces, la expresión $D_1 \bowtie E_2$ da el conjunto de tuplas que hay que insertar en $E_1 \bowtie E_2$.

Véanse las notas bibliográficas para obtener más detalles sobre la conservación incremental de las vistas con expresiones.

14.5.3 Optimización de consultas y vistas materializadas

La optimización de consultas puede llevarse a cabo tratando las vistas materializadas igual que a las relaciones normales. No obstante, las vistas materializadas ofrecen más oportunidades para la optimización:

- Reescritura de las consultas para el empleo de vistas materializadas:

  Supóngase que está disponible la vista materializada $v = r \bowtie s$ y que un usuario emite la consulta $r \bowtie s \bowtie t$. Puede que la reescritura de la consulta como $v \bowtie t$ proporcione un plan de

consulta más eficiente que la optimización de la consulta tal y como se ha emitido. Por tanto, es función del optimizador de consultas reconocer si se puede usar una vista materializada para acelerar una consulta.

- Sustitución del uso de una vista materializada por la definición de la vista:
  Supóngase que está disponible la vista materializada $v = r \bowtie s$, pero sin ningún índice definido sobre ella, y que un usuario emite la consulta $\sigma_{A=10}(v)$. Supóngase también que s tiene un índice sobre el atributo común B, y que r tiene un índice sobre el atributo A. Puede que el mejor plan para esta consulta sea sustituir v por $r \bowtie s$, lo que puede llevar al plan de consulta $\sigma_{A=10}(r) \bowtie s$; la selección y la reunión pueden llevarse a cabo de manera eficiente empleando los índices sobre $r.A$ y sobre $s.B$, respectivamente. Por el contrario, puede que la evaluación de la selección directamente sobre v necesite una exploración completa de v, lo que puede resultar más costoso.

Las notas bibliográficas dan indicaciones para investigar el modo de llevar a cabo de manera eficiente la optimización de las consultas con vistas materializadas.

Otro problema de optimización relacionado es el de la **selección de las vistas materializadas**, es decir, "la identificación del mejor conjunto de vistas para su materialización". Esta decisión debe tomarse con base en la **carga de trabajo** del sistema, que es una secuencia de consultas y de actualizaciones que refleja la carga típica del sistema. Un criterio sencillo sería la selección de un conjunto de vistas materializadas que minimice el tiempo global de ejecución de la carga de trabajo de consultas y de actualizaciones, incluido el tiempo empleado para conservar las vistas materializadas. Los administradores de bases de datos suelen modificar este criterio para tener en cuenta la importancia de las diferentes consultas y actualizaciones: puede ser necesaria una respuesta rápida para algunas consultas y actualizaciones, mientras que puede resultar aceptable una respuesta lenta para otras.

Los índices son como las vistas materializadas, en el sentido de que también son datos obtenidos, pueden acelerar las consultas y pueden desacelerar las actualizaciones. Por tanto, el problema de la **selección de índices** se halla íntimamente relacionado con el de la selección de las vistas materializadas, aunque resulta más sencillo.

Estos problemas se examinan con más detalle en los Apartados 23.1.5 y 23.1.6.

La mayoría de sistemas de bases de datos proporcionan herramientas para ayudar a los administradores de bases de datos en la selección de los índices y de las vistas materializadas. Estas herramientas examinan el historial de consultas y de actualizaciones y sugieren los índices y las vistas que hay que materializar. Database Tuning Assistant de SQL Server de Microsoft, Design Advisor de DB2 de IBM y Tuning Wizard de SQL de Oracle son ejemplos de estas herramientas.

14.6 Resumen

- Dada una consulta, suele haber gran variedad de métodos para calcular la respuesta. Es responsabilidad del sistema transformar la consulta tal y como la introdujo el usuario en una consulta equivalente que pueda calcularse de manera más eficiente. El proceso de búsqueda de una buena estrategia para el procesamiento de la consulta se denomina *optimización de consultas*.
- La evaluación de las consultas complejas implica muchos accesos a disco. Dado que la transferencia de los datos desde el disco resulta lenta en comparación con la velocidad de la memoria principal y de la CPU del sistema informático, merece la pena asignar una cantidad considerable de procesamiento a la elección de un método que minimice los accesos al disco.
- Hay varias reglas de equivalencia que se pueden emplear para transformar una expresión en otra equivalente. Estas reglas se emplean para generar de manera sistemática todas las expresiones equivalentes a la consulta dada.
- Cada expresión del álgebra relacional representa una secuencia concreta de operaciones. El primer paso para la selección de una estrategia de procesamiento de consultas es la búsqueda de una expresión del álgebra relacional que sea equivalente a la expresión dada y que se estime menos costosa de ejecutar.

- La estrategia que escoja el sistema de bases de datos para la evaluación de una operación depende del tamaño de cada relación y de la distribución de los valores dentro de las columnas. Para que puedan basar su elección de estrategia en información de confianza, los sistemas de bases de datos almacenan estadísticas para cada relación *r*. Entre estas estadísticas están
  - El número de tuplas de la relación *r*.
  - El tamaño del registro (tupla) de la relación *r* en bytes.
  - El número de valores diferentes que aparecen en la relación *r* para un atributo determinado.
- La mayoría de sistemas de bases de datos usan histogramas para almacenar el número de valores de un atributo en cada rango de valores.
- Estas estadísticas permiten estimar el tamaño del resultado de varias operaciones, así como el coste de su ejecución. La información estadística sobre las relaciones resulta especialmente útil cuando se dispone de varios índices para ayudar al procesamiento de una consulta. La presencia de estas estructuras tiene una influencia significativa en la elección de una estrategia de procesamiento de consultas.
- Los planes alternativos de evaluación para cada expresión pueden generarse mediante reglas de equivalencia y se puede escoger el plan más económico para todas las expresiones. Se dispone de varias técnicas de optimización para reducir el número de expresiones y planes alternativos que hace falta generar.
- Se usan heurísticas para reducir el número de planes considerados y, por tanto, para reducir el coste de la optimización. Entre las reglas heurísticas para transformar las consultas del álgebra relacional están "Llevar a cabo las operaciones de selección tan pronto como sea posible", "Llevar a cabo las proyecciones tan pronto como sea posible" y "Evitar los productos cartesianos".
- Las vistas materializadas pueden usarse para acelerar el procesamiento de las consultas. La conservación incremental de las vistas es necesaria para actualizar de forma eficiente las vistas materializadas cuando se modifican las relaciones subyacentes. El diferencial de cada operación puede calcularse mediante expresiones algebraicas que impliquen a los diferenciales de las entradas de la operación. Entre otros aspectos relacionados con las vistas materializadas están el modo de optimizar las consultas haciendo uso de las vistas materializadas disponibles y el modo de seleccionar las vistas que hay que materializar.

Términos de repaso

- Optimización de consultas.
- Transformación de expresiones.
- Equivalencia de expresiones.
- Reglas de equivalencia.
  - Conmutatividad de la reunión.
  - Asociatividad de la reunión.
- Conjunto mínimo de reglas de equivalencia.
- Enumeración de las expresiones equivalentes.
- Estimación de la estadística.
- Información de catálogo.
- Estimación del tamaño.
  - Selección.
  - Selectividad.
  - Reunión.
- Histogramas.
- Estimación de los valores distintos.
- Elección de los planes de evaluación.
- Interacción de las técnicas de evaluación.
- Optimización basada en el coste.
- Optimización del orden de reunión.
  - Algoritmo de programación dinámica.
  - Orden de reunión en profundidad por la izquierda.
- Optimización heurística.
- Elección del plan de acceso.
- Evaluación correlacionada.
- Descorrelación.
- Vistas materializadas.

- Mantenimiento de las vistas materializadas.
  - ☐ Recálculo.
  - ☐ Mantenimiento incremental.
  - ☐ Inserción.
  - ☐ Borrado.
  - ☐ Actualización.
- Optimización de las consultas con las vistas materializadas.
- Selección de índices.
- Selección de las vistas materializadas.

Ejercicios prácticos

14.1 Demuéstrese que se cumplen las equivalencias siguientes. Explíquese el modo en que se pueden aplicar para mejorar la eficiencia de determinadas consultas:

a. $E_1 \bowtie_\theta (E_2 - E_3) = (E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3)$.
b. $\sigma_\theta({}_A\mathcal{G}_F(E)) = {}_A\mathcal{G}_F(\sigma_\theta(E))$, donde θ sólo usa atributos de A.
c. $\sigma_\theta(E_1 \rightarrowtriangle E_2) = \sigma_\theta(E_1) \rightarrowtriangle E_2$, donde θ sólo usa atributos de E_1.

14.2 Para cada uno de los siguientes pares de expresiones, dense ejemplos de relaciones que muestren que las expresiones no son equivalentes.

a. $\Pi_A(R - S)$ y $\Pi_A(R) - \Pi_A(S)$.
b. $\sigma_{B<4}({}_A\mathcal{G}_{max(B)}(R))$ y ${}_A\mathcal{G}_{max(B)}(\sigma_{B<4}(R))$.
c. En las expresiones anteriores, si las dos apariciones de max se sustituyeran por min, indicar si las expresiones serían equivalentes.
d. $(R \rightarrowtriangle S) \rightarrowtriangle T$ y $R \rightarrowtriangle (S \rightarrowtriangle T)$
En otras palabras, la reunión externa por la izquierda no es asociativa.
Sugerencia: supóngase que los esquemas de las tres relaciones son $R(a, b1)$, $S(a, b2)$ y $T(a, b3)$, respectivamente.
e. $\sigma_\theta(E_1 \rightarrowtriangle E_2)$ y $E_1 \rightarrowtriangle \sigma_\theta(E_2)$, donde θ usa sólo atributos de E_2.

14.3 SQL permite las relaciones con duplicados (Capítulo 3).

a. Defínanse las versiones de las operaciones básicas del álgebra relacional σ, Π, $\times$, $\bowtie$, $-$, $\cup$ y $\cap$ que se utilizan en relaciones con duplicados, de manera consistente con SQL.
b. Compruébese cuáles de las reglas de equivalencia de la 1 a la 7.b se cumplen para la versión multiconjunto del álgebra relacional definida en el apartado a.

14.4 Considérense las relaciones $r_1(A, B, C)$, $r_2(C, D, E)$ y $r_3(E, F)$, con las claves principales *A*, *C*, and *E*, respectivamente. Supóngase que r_1 tiene 1.000 tuplas, r_2 tiene 1.500 tuplas y r_3 tiene 750 tuplas. Estímese el tamaño de $r_1 \bowtie r_2 \bowtie r_3$ y diséñese una estrategia eficiente para el cálculo de la reunión.

14.5 Considérense las relaciones $r_1(A, B, C)$, $r_2(C, D, E)$ y $r_3(E, F)$ del Ejercicio práctico 14.4. Supóngase que no hay claves principales, excepto el esquema completo. Sean $V(C, r_1)$ 900, $V(C, r_2)$ 1.100, $V(E, r_2)$ 50 y $V(E, r_3)$ 100. Supóngase que r_1 tiene 1.000 tuplas, r_2 tiene 1.500 tuplas y que r_3 tiene 750 tuplas. Estímese el tamaño de $r_1 \bowtie r_2 \bowtie r_3$ y diséñese una estrategia eficiente para el cálculo de la reunión.

14.6 Supóngase que se dispone de un árbol B^+ para *ciudad_sucursal* para la relación *sucursal* y que no se dispone de ningún otro índice. Indíquese la mejor manera de tratar las siguientes selecciones que implican a la negación:

a. $\sigma_{\neg(ciudad_sucursal < \text{“Arganzuela”})}(sucursal)$
b. $\sigma_{\neg(ciudad_sucursal = \text{“Arganzuela”})}(sucursal)$
c. $\sigma_{\neg(ciudad_sucursal < \text{“Arganzuela”} \vee activos < 5000)}(sucursal)$

14.7 Demuéstrese que, con n relaciones, hay $(2(n-1))!/(n-1)!$ órdenes de reunión diferentes.
Sugerencia: un **árbol binario completo** es aquél en el que cada nodo interno tiene exactamente dos hijos. Utilícese el hecho de que el número de árboles binarios completos diferentes con n

nodos hojas es

$$\frac{1}{n}\binom{2(n-1)}{(n-1)}$$

Si se desea, se puede obtener la fórmula para el número de árboles binarios completos con n nodos a partir de la fórmula para el número de árboles binarios con n nodos. El número de árboles binarios con n nodos es

$$\frac{1}{n+1}\binom{2n}{n}$$

Este número se conoce como **número de Catalan** y su obtención puede hallarse en cualquier libro de texto estándar sobre estructuras de datos o algoritmos.

14.8 Demuéstrese que el orden de reunión de menor coste puede calcularse en un tiempo $O(3^n)$. Supóngase que se puede almacenar y examinar la información sobre un conjunto de relaciones (como el orden óptimo de reunión para el conjunto y el coste de ese orden de reunión) en un tiempo constante. (Si se encuentra difícil este ejercicio, demuéstrese al menos la cota de tiempo menos estricta de $O(2^{2n})$).

14.9 Demuéstrese que, si sólo se toman en consideración los árboles de reunión en profundidad por la izquierda, como en el optimizador System R, el tiempo empleado en buscar el orden de reunión más eficiente es del orden de $n2^n$. Supóngase que sólo hay un orden interesante.

14.10 Descorrelación:

- **a.** Escríbase una consulta anidada sobre la relación *cuenta* para buscar, para cada sucursal cuyo nombre comience por B, todas las cuentas con el saldo máximo de cada sucursal.
- **b.** Reescríbase la consulta anterior, sin emplear consultas anidadas; en otras palabras, descorrelaciónese la consulta.
- **c.** Diséñese un procedimiento (parecido al descrito en el Apartado 14.4.4) para descorrelacionar estas consultas.

Ejercicios

14.11 Supóngase que se dispone de un árbol B^+ para (*nombre_sucursal*, *ciudad_sucursal*) para la relación *sucursal*. Indíquese la mejor manera de tratar la selección siguiente:

$$\sigma_{(ciudad_sucursal<\text{“Arganzuela”}) \wedge (activos<5000) \wedge (nombre_sucursal=\text{“Centro”})}(sucursal)$$

14.12 Muéstrese el modo de obtener las equivalencias siguientes mediante una secuencia de transformaciones usando las reglas de equivalencia del Apartado 14.2.1.

- **a.** $\sigma_{\theta_1 \wedge \theta_2 \wedge \theta_3}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(\sigma_{\theta_3}(E)))$
- **b.** $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_3} E_2) = \sigma_{\theta_1}(E_1 \bowtie_{\theta_3} (\sigma_{\theta_2}(E_2)))$, donde θ_2 sólo implica atributos de E_2

14.13 Se dice que un conjunto de reglas de equivalencia está *completo* si, siempre que dos expresiones son equivalentes, se puede obtener una de la otra mediante una secuencia de usaciones de las reglas de equivalencia. Indíquese si el conjunto de reglas de equivalencia que se consideró en el Apartado 14.2.1 es completo. *Sugerencia*: considérese la equivalencia $\sigma_{3=5}(r) = \{\}$.

14.14 Descríbase el modo de usar un histograma para estimar el tamaño de una selección de la forma $\sigma_{A \leq v}(r)$.

14.15 Supóngase que dos relaciones r y s tienen histogramas sobre los atributos $r.A$ y $s.A$ respectivamente, pero con rangos diferentes. Sugiérase el modo de usar los histogramas para estimar el tamaño de $r \bowtie s$. *Sugerencia*: divídanse más los rangos de cada histograma.

14.16 Descríbase el modo de conservar de manera incremental el resultado de las operaciones siguientes, tanto para inserciones como para borrados:

- **a.** Unión y diferencia de conjuntos.
- **b.** Reunión externa por la izquierda.

14.17 Dese un ejemplo de expresión que defina una vista materializada y dos situaciones (conjuntos de estadísticas para las relaciones de entrada y sus diferenciales) tales que la conservación incremental de la vista sea mejor que su recálculo en una de las situaciones y el recálculo sea mejor en la otra.

14.18 Supóngase que se desea obtener resultados ordenados para $r \bowtie s$ según un atributo de r y sólo se desean las primeras K respuestas para algún K relativamente pequeño. Dese una buena forma de evaluar la consulta

a. Cuando la reunión es sobre una clave externa de r que referencia a s.
b. Cuando la reunión no es sobre una clave externa.

14.19 Considérese la relación $r(A, B, C)$ con un índice sobre el atributo A. Dese un ejemplo de una consulta que se pueda responder usando sólo el índice, sin tener que examinar las tuplas de la relación. (Los planes de consulta que usan sólo el índice sin acceder a la relación se denominan planes *sólo de índices*).

Notas bibliográficas

El trabajo precursor de Selinger et al. [1979] describe la selección del camino de acceso en el optimizador System R, que fue uno de los primeros optimizadores de consultas relacionales.El procesamiento de consultas en Starburst, que se describe en Haas et al. [1989], construye las bases de la optimización de consultas en DB2 de IBM.

Graefe y McKenna [1993] describen Volcano, un optimizador de consultas basado en reglas de equivalencia que, junto a su sucesor Cascades (Graefe [1995]) forma el fundamento de la optimización de consultas en SQL Server de Microsoft.

Véanse los Capítulos 27, 28 y 29 para más información sobre el procesamiento y optimización de consultas en Oracle, DB2 de IBM y SQL Server de Microsoft respectivamente.

Estadísticas y estimación del coste: la estimación de las estadísticas de los resultados de las consultas, como el tamaño del resultado, se aborda en Ioannidis y Poosala [1995], Poosala et al. [1996] y Ganguly et al. [1996], entre otros. Las distribuciones no uniformes de valores causan problemas para la estimación del tamaño y del coste de las consultas. Las técnicas de estimación del coste que usan histogramas de las distribuciones de los valores se han propuesto para abordar el problema. Ioannidis y Christodoulakis [1993], Ioannidis y Poosala [1995] y Poosala et al. [1996] presentan los resultados en este área.

Todos los principales sistemas de bases de datos comerciales usan en gran medida los histogramas para la estimación del coste. También soportan el muestreo para calcular eficientemente los histogramas sin necesidad de examinar la relación completa: es muy probable que un histograma construido a partir de un subconjunto de la relación seleccionado aleatoriamente sea muy parecido al histograma de la relación completa si el subconjunto de muestra es suficientemente grande.

Heurísticas: la búsqueda exhaustiva de todos los planes de consultas no resulta práctica para la optimizacion de las reuniones que implican a muchas relaciones. La mayoría de optimizadores usan heurísticas para encontrar algunos planes buenos, además de usar un algoritmo exhaustivo para generar órdenes alternativos de la reunión. Los algoritmos exhaustivos terminan cuando se excede un límite de sobrecarga; por ejemplo, si el coste de optimización alcanza una fracción significativa del coste estimado de ejecución.

Optimización paramétrica de consultas: Ioannidis et al. [1992], Ganguly [1998] y Hulgeri y Sudarshan [2003] han propuesto técnicas para optimizar consultas cuando la selectividad de los parámetros de la consulta no se conoce en el momento de la optimización. Se calcula un conjunto de planes (uno por cada una de las diferentes selectividades de las consultas) y el optimizador lo almacena, en el momento de la compilación. Uno de estos planes se elige en el momento de la ejecución, con base en las selectividades reales, evitando el coste de la optimización completa en el momento de la ejecución.

Optimización de agregación: Klug [1982] fue uno de los primeros trabajos sobre optimización de expresiones del álgebra relacional con funciones de agregación. Yan y Larson [1995] and Chaudhuri y Shim [1994] estudian la optimización de consultas con agregación. La optimización de las consultas que con-

tienen reuniones externas se describe en Rosenthal y Reiner [1984], Galindo-Legaria y Rosenthal [1992] y Galindo-Legaria [1994].

Optimización de los primeros K: muchas consultas recuperan los resultados ordenados bajo algunos atributos y requieren solamente los resultados de los primeros K para algún K. Cuando K es pequeño, un plan de la optimización de consulta que genere el conjunto completo de resultados, después ordene y genere los primeros K sería muy ineficaz puesto que desecharía la mayoría de los resultados intermedios que calcula. Se han propuesto varias técnicas para optimizar estas *consultas de los primeros K*. Un enfoque es usar los planes encauzados que puedan generar los resultados ordenados. Otro enfoque es estimar el mayor valor de los atributos que aparecerán en la salida de los primeros K, e introducir los predicados de selección que eliminan los valores mayores. Si se generan tuplas adicionales por debajo de los primeros K, se desechan y, si se generan muy pocas tuplas, entonces se cambia la condición de la selección y se ejecuta nuevamente la consulta. La optimización de consultas de los primeros K se trata en Carey y Kossmann [1998] y en Bruno et al. [2002].

El lenguaje SQL plantea varios desafíos para la optimización de las consultas, incluidas la presencia de valores duplicados y nulos y la semántica de las subconsultas anidadas. La extensión del álgebra relacional a los valores duplicados se describe en Dayal et al. [1982]. La optimización de las subconsultas anidadas se trata en Kim [1982], Ganski y Wong [1987], Dayal [1987], Seshadri et al. [1996] y, más recientemente, en Galindo-Legaria y Joshi [2001].

Minimización de la reunión: cuando las consultas se generan mediante vistas, se suelen reunir más relaciones de las necesarias para el cálculo de cada consulta. Se ha agrupado un conjunto de técnicas para la minimización de las reuniones bajo el nombre de *optimización con tableau*. El concepto de tableau lo introdujeron Aho et al. [1979b] y Aho et al. [1979a] y lo ampliaron Sagiv y Yannakakis [1981]. Ullman [1988] y Maier [1983] proporcionan un tratamiento de los tableaux con un nivel de libro de texto.

Optimización de multiconsultas: Sellis [1988] y Roy et al. [2000] describen la *optimización de multiconsultas*, que es el problema de optimización de la ejecución de varias consultas como si fueran un grupo. Si se toma en consideración todo un grupo de consultas, resulta posible descubrir *subexpresiones comunes* que pueden evaluarse una sola vez para todo el grupo. Finkelstein [1982] y Hall [1976] consideran la optimización de un grupo de consultas y el empleo de las subexpresiones comunes. Dalvi et al. [2001] estudia los problemas de optimización en los encauzamientos con espacio de memorias intermedias limitado combinadas con el compartimiento de subexpresiones comunes.

Optimización semántica de consultas: la optimización de consultas puede hacer uso de la información semántica, como las dependencias funcionales y otras restricciones de integridad. La *optimización semántica de consultas* en las bases de datos relacionales se estudia en King [1981], Chakravarthy et al. [1990] y, en el contexto de la agregación, en Sudarshan y Ramakrishnan [1991].

Vistas materializadas: Blakeley et al. [1986], Blakeley et al. [1989] y Griffin y Libkin [1995] describen las técnicas para la conservación de las vistas materializadas. Gupta y Mumick [1995] proporcionan una reseña de la conservación de las vistas materializadas. La optimización de los planes de conservación de las vistas materializadas se describe en Vista [1998] y Mistry et al. [2001]. La optimización de consultas en presencia de vistas materializadas se aborda en Larson y Yang [1985], Chaudhuri et al. [1995], Dar et al. [1996] y Roy et al. [2000]. La selección de índices y la selección de vistas materializadas se aborda en Ross et al. [1996], Labio et al. [1997], Gupta [1997], Chaudhuri y Narasayya [1997] y Roy et al. [2000]. Los problemas que aparecen en la selección de índices y vistas materializadas se estudian más tarde en los Apartados 23.1.5, 23.1.6 y 23.1.7.

Optimización de actualizaciones: Galindo-Legaria et al. [2004] describe el proceso y la optimización de consultas para las actualizaciones de la base de datos, incluyendo la optimización del mantenimiento del índice, los planes materializados del mantenimiento de la visión y la comprobación de las restricciones de integridad. Las preguntas de la actualización implican a menudo subconsultas en las cláusulas **set** y **where**, que también deben tomarse en consideración al optimizar la actualización.

Las actualizaciones que implican una selección en la columna actualizada (e.g. dan un aumento del 10 por ciento a todos los empleados cuyo sueldo sea $\geq$ 100.000 €) se deben tratar con cuidado. Si se hace la actualización mientras una exploración del índice evalúa la selección, se puede volver a insertar una tupla actualizada en el índice por delante de la zona explorada y la exploración puede volver a tenerla en cuenta; la misma tupla de empleado, por tanto, se puede actualizar varias veces de manera

incorrecta (un número infinito de veces, en este caso). Un problema parecido se presenta también con las actualizaciones que implican subconsultas cuyo resultado queda afectado por la actualización.

El problema de que una actualización afecte a la ejecución de una consulta asociada a la actualización se conoce como *problema de Halloween* (denominado así por la fecha en que IBM lo reconoció por primera vez). Este problema se puede evitar ejecutando en primer lugar las consultas que definen la actualización, creando una lista de tuplas afectadas y actualizando las tuplas y los índices como último paso. Sin embargo, dividir el plan de ejecución de esa manera aumenta el coste de ejecución. Los planes de actualización se pueden optimizar comprobando si se puede dar el problema de Halloween y, si no es el caso, las actualizaciones se pueden llevar a cabo mientras se procesa la consulta, reduciendo la sobrecarga de actualización. Este tipo de optimización se implementa en la mayor parte de los sistemas de bases de datos, y se describe, por ejemplo, en Galindo-Legaria et al. [2004].

Indexación, procesamiento de consultas y optimización de consultas XML: la indexación de datos XMLy el procesamiento y optimización de consultas XML han sido áreas de gran interés en los últimos años. Se ha publicado gran cantidad de artículos en este área. Uno de los retos de la indexación de datos es que las consultas puedan especificar una selección de ruta, tal como */a/b//c*[*d*="CSE"]; el índice debe soportar la recuperación eficiente de los nodos que satisfagan la especificación de ruta y la selección de valores. En Pal et al. [2004] y Kaushik et al. [2004] se puede encontrar trabajo reciente sobre la indexación de datos XML. Si los datos se dividen y almacenan en relaciones, la evaluación de las expresiones de ruta se corresponde con el cálculo de reuniones. Se han propuesto varias técnicas para el cálculo eficiente de esas reuniones, en especial cuando la expresión de ruta especifica cualquier descendiente (//). Se han propuesto varias técnicas para la numeración de los nodos de los datos XML que se pueden usar para comprobar de manera eficiente si un nodo es un descendiente de otro; véase, por ejemplo, O'Neil et al. [2004]. McHugh y Widom [1999], Wu et al. [2003] y Krishnaprasad et al. [2004] son trabajos sobre la optimización de consultas XML.

PARTE 5

Gestión de transacciones

El término *transacción* hace referencia a un conjunto de operaciones que forman una única unidad lógica de trabajo. Por ejemplo, la transferencia de dinero de una cuenta a otra es una transacción que consta de dos actualizaciones, una para cada cuenta.

Resulta importante que, o bien se ejecuten completamente todas las acciones de una transacción, o bien, en caso de fallo, se deshagan los efectos parciales de cada transacción incompleta. Esta propiedad se denomina *atomicidad*. Además, una vez ejecutada con éxito una transacción, sus efectos deben persistir en la base de datos—un fallo en el sistema no debe tener como consecuencia que la base de datos descarte una transacción que se haya completado con éxito. Esta propiedad se denomina *durabilidad*.

En los sistemas de bases de datos en los que se ejecutan de manera concurrente varias transacciones, si no se controlan las actualizaciones de los datos compartidos, existe la posibilidad de que las transacciones operen sobre estados intermedios inconsistentes creados por las actualizaciones de otras transacciones. Esta situación puede dar lugar a actualizaciones erróneas de los datos almacenados en la base de datos. Por tanto, los sistemas de bases de datos deben proporcionar los mecanismos para aislar las transacciones de otras transacciones que se ejecuten de manera concurrente. Esta propiedad se denomina *aislamiento*.

El Capítulo 15 describe con detalle el concepto de transacción, incluidas las propiedades de atomicidad, durabilidad, aislamiento y otras propiedades proporcionadas por la abstracción de las transacciones. En concreto, el capítulo precisa el concepto de aislamiento por medio de un concepto denominado secuencialidad.

El Capítulo 16 describe varias técnicas de control de concurrencia que ayudan a implementar la propiedad del aislamiento. El Capítulo 17 describe el componente de las bases de datos para la administración de las recuperaciones, que implementa las propiedades de atomicidad y de durabilidad.

Entendido como un todo, el componente de gestión de transacciones de un sistema de bases de datos permite a los desarrolladores de bases de datos centrarse en la implementación de las transacciones individualmente, ignorando los aspectos de concurrencia y tolerancia a fallos.

CAPÍTULO 15

Transacciones

A menudo, desde el punto de vista del usuario de una base de datos, se considera a un conjunto de varias operaciones sobre una base de datos como una única operación. Por ejemplo, una transferencia de fondos desde una cuenta corriente a una cuenta de ahorros es una operación simple desde el punto de vista del cliente; sin embargo, en el sistema de base de datos, está compuesta internamente por varias operaciones. Evidentemente es esencial que tengan lugar todas las operaciones o que, en caso de fallo, ninguna de ellas se produzca. Sería inaceptable efectuar el cargo de la transferencia en la cuenta corriente y que no se abonase en la cuenta de ahorros.

Se llama **transacción** a una colección de operaciones que forman una única unidad lógica de trabajo. Un sistema de base de datos debe asegurar que la ejecución de las transacciones se realice adecuadamente a pesar de la existencia de fallos—o se ejecuta la transacción completa o no se ejecuta en absoluto. Además debe gestionar la ejecución concurrente de las transacciones evitando introducir inconsistencias. Volviendo al ejemplo de la transferencia de fondos, una transacción que calcule el saldo total del cliente podría ver el saldo de la cuenta corriente antes de que sea cargado por la transacción de la transferencia de fondos, y el saldo de la cuenta de ahorros después del abono. Como resultado, se obtendría un resultado incorrecto.

Este capítulo es una introducción a los conceptos básicos en el procesamiento de transacciones. En los Capítulos 16 y 17 se incluyen más detalles sobre el procesamiento concurrente de transacciones y la recuperación de fallos. En el Capítulo 24 se tratan temas adicionales acerca del procesamiento de transacciones.

15.1 Concepto de transacción

Una **transacción** es una **unidad** de la ejecución de un programa que accede y posiblemente actualiza varios elementos de datos. Una transacción se inicia por la ejecución de un programa de usuario escrito en un lenguaje de manipulación de datos de alto nivel o en un lenguaje de programación (por ejemplo SQL, C++ o Java), y está delimitado por instrucciones (o llamadas a función) de la forma **begin transaction** (inicio transacción) y **end transaction** (fin transacción). La transacción consiste en todas las operaciones que se ejecutan entre **begin transaction** y **end transaction**.

Para asegurar la integridad de los datos se necesita que el sistema de base de datos mantenga las siguientes propiedades de las transacciones:

- **Atomicidad**. O bien todas las operaciones de la transacción se realizan adecuadamente en la base de datos o ninguna de ellas.
- **Consistencia**. La ejecución aislada de la transacción (es decir, sin otra transacción que se ejecute concurrentemente) conserva la consistencia de la base de datos.

- **Aislamiento**. Aunque se ejecuten varias transacciones concurrentemente, el sistema garantiza que para cada par de transacciones T_i y T_j, se cumple que para los efectos de T_i, o bien T_j ha terminado su ejecución antes de que comience T_i, o bien que T_j ha comenzado su ejecución después de que T_i termine. De este modo, cada transacción ignora al resto de las transacciones que se ejecuten concurrentemente en el sistema.
- **Durabilidad**. Tras la finalización con éxito de una transacción, los cambios realizados en la base de datos permanecen, incluso si hay fallos en el sistema.

Estas propiedades a menudo reciben el nombre de **propiedades ACID**; el acrónimo se obtiene de la primera letra de cada una de las cuatro propiedades en inglés (*Atomicity, Consistency, Isolation* y *Durability*, respectivamente).

Para comprender mejor las propiedades ACID y la necesidad de dichas propiedades, considérese un sistema bancario simplificado constituido por varias cuentas y un conjunto de transacciones que acceden y actualizan dichas cuentas. Por ahora se asume que la base de datos reside permanentemente en disco, pero una porción de la misma reside temporalmente en la memoria principal.

El acceso a la base de datos se lleva a cabo mediante las dos operaciones siguientes:

- leer(X), que transfiere el dato X de la base de datos a una memoria intermedia local perteneciente a la transacción que ejecuta la operación leer.
- escribir(X), que transfiere el dato X desde la memoria intermedia local de la transacción que ejecuta la operación escribir a la base de datos.

En un sistema de base de datos real, la operación escribir no tiene por qué producir necesariamente una actualización de los datos en disco; la operación escribir puede almacenarse temporalmente en memoria y llevarse a disco más tarde. Sin embargo, por el momento se supondrá que la operación escribir actualiza inmediatamente la base de datos. Se volverá a este tema en el Capítulo 17.

Sea T_i una transacción para transferir 50 € de la cuenta A a la cuenta B. Se puede definir dicha transacción como

T_i: leer(A);
$A := A - 50$;
escribir(A);
leer(B);
$B := B + 50$;
escribir(B).

Considérense ahora cada una de las propiedades ACID (para una presentación más cómoda, se consideran en distinto orden al indicado por A-C-I-D).

- **Consistencia**. En este caso el requisito de consistencia es que la suma de A y B no sea alterada al ejecutar la transacción. Sin el requisito de consistencia, ¡la transacción podría crear o destruir dinero! Se puede comprobar fácilmente que si una base de datos es consistente antes de ejecutar una transacción, sigue siéndolo después de ejecutar dicha transacción.

  La responsabilidad de asegurar la consistencia de una transacción es del programador de la aplicación que codifica dicha transacción. La comprobación automática de las restricciones de integridad puede facilitar esta tarea, como se vio en el Apartado 4.2.
- **Atomicidad**. Supóngase que justo antes de ejecutar la transacción T_i los valores de las cuentas A y B son de 1.000 € y de 2.000 €, respectivamente. Supóngase ahora que durante la ejecución de la transacción T_i se produce un fallo que impide que dicha transacción finalice con éxito su ejecución. Ejemplos de este tipo de fallos pueden ser los fallos en la alimentación, los fallos del hardware y los errores software. Además, supóngase que el fallo tiene lugar después de ejecutarse la operación escribir(A), pero antes de ejecutarse la operación escribir(B). Es ese caso, los valores de las cuentas A y B que se ven reflejados en la base de datos son 950 € y 2.000 €. Se han

perdido 50 € de la cuenta A como resultado de este fallo. En particular se puede ver que ya no se conserva la suma $A + B$.

Así, como resultado del fallo, el estado del sistema deja de reflejar el estado real del mundo que se supone que modela la base de datos. Un estado así se denomina **estado inconsistente**. Hay que asegurarse de que estas inconsistencias no sean visibles en un sistema de base de datos. Obsérvese, sin embargo, que un sistema puede en algún momento alcanzar un estado inconsistente. Incluso si la transacción T_i se ejecuta por completo, existe un punto en el que el valor de la cuenta A es de 950 € y el de la cuenta B es de 2.000 €, lo cual constituye claramente un estado inconsistente. Este estado, sin embargo, se sustituye eventualmente por otro estado consistente en el que el valor de la cuenta A es de 950 € y el de la cuenta B es de 2.050 €. De este modo, si la transacción no empieza nunca o se garantiza que se complete, un estado inconsistente así no será visible excepto durante la ejecución de la transacción. Ésta es la razón de que aparezca el requisito de atomicidad. Si se proporciona la propiedad de atomicidad, o todas las acciones de la transacción se ven reflejadas en la base de datos, o ninguna de ellas.

La idea básica para asegurar la atomicidad es la siguiente. El sistema de base de datos mantiene los valores antiguos (en disco) de aquellos datos sobre los que una transacción realiza una escritura y, si la transacción no completa su ejecución, los valores antiguos se recuperan para que parezca que la transacción no se ha ejecutado. Estas ideas se muestran más adelante en el Apartado 15.2. La responsabilidad de asegurar la atomicidad es del sistema de base de datos; en concreto, lo maneja un componente llamado **componente de gestión de transacciones**, que se describe en detalle en el Capítulo 17.

- **Durabilidad**. Una vez que se completa con éxito la ejecución de una transacción, y después de comunicar al usuario que inició la transacción que se ha realizado la transferencia de fondos, un fallo en el sistema no debe producir la pérdida de datos correspondientes a dicha transferencia.

  La propiedad de durabilidad asegura que, una vez que se completa con éxito una transacción, persisten todas las modificaciones realizadas en la base de datos, incluso si hay un fallo en el sistema después de completarse la ejecución de dicha transacción.

  A partir de ahora se asume que un fallo en la computadora del sistema produce una pérdida de datos de la memoria principal, pero los datos almacenados en disco nunca se pierden. Se puede garantizar la durabilidad si se asegura que

  1. Las modificaciones realizadas por la transacción se guardan en disco antes de que finalice la transacción.
  2. La información de las modificaciones realizadas por la transacción guardada en disco es suficiente para permitir a la base de datos reconstruir dichas modificaciones cuando el sistema se reinicie después del fallo.

  La responsabilidad de asegurar la durabilidad pertenece a un componente software del sistema de base de datos llamado **componente de gestión de recuperaciones**. El componente de gestión de transacciones y el componente de gestión de recuperaciones están estrechamente relacionados, y su implementación se describe en el Capítulo 17.

- **Aislamiento**. Incluso si se aseguran las propiedades de consistencia y de atomicidad para cada transacción, si varias transacciones se ejecutan concurrentemente, se pueden entrelazar sus operaciones de un modo no deseado, produciendo un estado inconsistente.

  Por ejemplo, como se ha visto antes, la base de datos es inconsistente temporalmente durante la ejecución de la transacción para transferir fondos de la cuenta A a la cuenta B, con el total deducido escrito ya en A y el total incrementado todavía sin escribir en B. Si una segunda transacción que se ejecuta concurrente lee A y B en este punto intermedio y calcula $A + B$, observará un valor inconsistente. Además, si esta segunda transacción realiza después modificaciones en A y B basándose en los valores leídos, la base de datos puede permanecer en un estado inconsistente aunque ambas transacciones terminen.

  Una solución para el problema de ejecutar transacciones concurrentemente es ejecutarlas secuencialmente—es decir, una tras otra. Sin embargo, la ejecución concurrente de transacciones

produce notables beneficios en el rendimiento, como se verá en el Apartado 15.4. Se han desarrollado otras soluciones que permiten la ejecución concurrente de varias transacciones.

Los problemas que causa la ejecución concurrente de transacciones se muestra en el Apartado 15.4. La propiedad de aislamiento asegura que el resultado obtenido al ejecutar concurrentemente las transacciones es un estado del sistema equivalente a uno obtenido al ejecutar una tras otra en algún orden. Los principios de aislamiento se presentarán más adelante en el Apartado 15.5. La responsabilidad de asegurar la propiedad de aislamiento es de un componente del sistema de base de datos llamado **componente de control de concurrencia**, que se presenta en el Capítulo 16.

15.2 Estados de una transacción

En ausencia de fallos, todas las transacciones se completan con éxito. Sin embargo, como se ha visto antes, una transacción puede que no siempre termine su ejecución con éxito. Una transacción de este tipo se denomina **abortada**. Si se pretende asegurar la propiedad de atomicidad, una transacción abortada no debe tener efecto sobre el estado de la base de datos. Así, cualquier cambio que haya hecho la transacción abortada sobre la base de datos debe deshacerse. Una vez que se han deshecho los cambios efectuados por la transacción abortada, se dice que la transacción está **retrocedida**. Parte de la responsabilidad del esquema de recuperaciones es gestionar las transacciones abortadas.

Una transacción que termina con éxito se dice que está **comprometida**. Una transacción comprometida que haya hecho modificaciones transforma la base de datos llevándola a un nuevo estado consistente, que permanece incluso si hay un fallo en el sistema.

Cuando una transacción se ha comprometido no se pueden deshacer sus efectos abortándola. La única forma de deshacer los cambios de una transacción comprometida es ejecutando una **transacción compensadora**. Por ejemplo, si una transacción añade 20 € a una cuenta, la transacción compensadora debería restar 20 € de la cuenta. Sin embargo, no siempre se puede crear dicha transacción compensadora. Por tanto, se deja al usuario la responsabilidad de crear y ejecutar transacciones compensadoras, y no la gestiona el sistema de base de datos. En el Capítulo 25 se incluye un estudio de las transacciones compensadoras.

Es necesario precisar qué se entiende por *terminación con éxito* de una transacción. Se establece por tanto un simple modelo abstracto de transacción. Una transacción debe estar en uno de los estados siguientes:

- **Activa**, el estado inicial; la transacción permanece en este estado durante su ejecución.
- **Parcialmente comprometida**, después de ejecutarse la última instrucción.
- **Fallida**, tras descubrir que no puede continuar la ejecución normal.
- **Abortada**, después del retroceso de la transacción y de haber restablecido la base de datos la base de datos a su estado anterior al comienzo de la transacción.
- **Comprometida**, tras completarse con éxito.

El diagrama de estados correspondiente a una transacción se muestra en la Figura 15.1. Se dice que una transacción se ha comprometido sólo si ha llegado al estado comprometida. Análogamente, se dice que una transacción ha abortado sólo si ha llegado al estado abortada. Una transacción se dice que ha **terminado** si se ha comprometido o bien se ha abortado.

Una transacción comienza en el estado activa. Cuando acaba su última instrucción pasa al estado de parcialmente comprometida. En este punto la transacción ha terminado su ejecución, pero es posible que aún tenga que ser abortada, puesto que los datos actuales pueden estar todavía en la memoria principal y puede producirse un fallo en el hardware antes de que se complete con éxito.

El sistema de base de datos escribe en disco la información suficiente para que, incluso al producirse un fallo, puedan reproducirse los cambios hechos por la transacción al reiniciar el sistema tras el fallo. Cuando se termina de escribir esta información, la transacción pasa al estado comprometida.

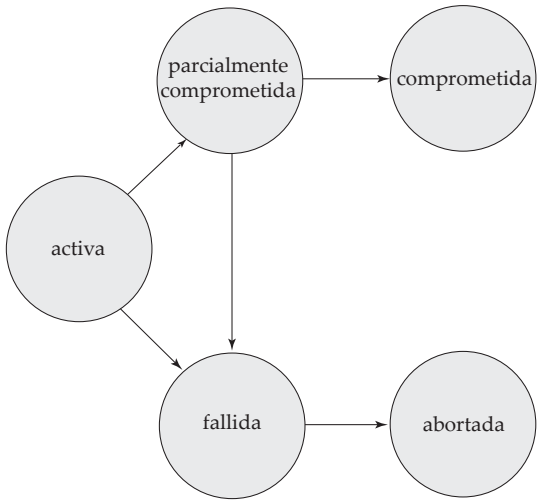


Figura 15.1 Diagrama de transición de estado de una transacción.

Como se ha mencionado antes, se asume que los fallos no provocan pérdidas de datos en disco. Las técnicas para tratar las pérdidas de datos en disco se muestran en el Capítulo 17.

Una transacción llega al estado fallida después de que el sistema determine que dicha transacción no puede continuar su ejecución normal (por ejemplo, a causa de errores de hardware o lógicos). Una transacción de este tipo se debe retroceder. Después pasa al estado abortada. En este punto, el sistema tiene dos opciones:

- **Reiniciar** la transacción, pero sólo si la transacción se ha abortado a causa de algún error hardware o software que no lo haya provocado la lógica interna de la transacción. Una transacción reiniciada se considera una nueva transacción.
- **Cancelar** la transacción. Normalmente se hace esto si hay algún error interno lógico que sólo se puede corregir escribiendo de nuevo el programa de aplicación, o debido a una entrada incorrecta o a que no se han encontrado los datos deseados en la base de datos.

Hay que tener cuidado cuando se trabaja con **escrituras externas observables**, como en un terminal o en una impresora. Cuando una escritura así tiene lugar, no puede borrarse puesto que puede haber sido vista fuera del sistema de base de datos. Muchos sistemas permiten que tales escrituras tengan lugar sólo después de que la transacción llegue al estado comprometida. Una manera de implementar dicho esquema es hacer que el sistema de base de datos almacene temporalmente cualquier valor asociado con estas escrituras externas en memoria no volátil, y realice las escrituras actuales sólo si la transacción llega al estado comprometida. Si el sistema falla después de que la transacción llegue al estado comprometida, pero antes de que finalicen las escrituras externas, el sistema de base de datos puede llevar a cabo dichas escrituras externas (usando los datos de la memoria no volátil) cuando el sistema se reinicie.

La gestión de las escrituras externas puede ser más complicada en ciertas situaciones. Por ejemplo, supóngase que la acción externa es dispensar dinero en un cajero automático y el sistema falla justo antes de que se dispense el dinero (se asume que el dinero se dispensa atómicamente). No tiene sentido dispensar el dinero cuando se reinicie el sistema, ya que el usuario probablemente no esté en el cajero. En tal caso es necesario una transacción compensadora, como devolver el dinero a la cuenta del usuario.

Para algunas aplicaciones puede ser deseable permitir a las transacciones activas que muestren datos a los usuarios, particularmente para transacciones de larga duración que se ejecutan durante minutos u horas. Desafortunadamente no se puede permitir dicha salida de datos observables a no ser que se quiera arriesgar la atomicidad de la transacción. Muchos sistemas de bases de datos actuales aseguran la atomicidad y, por tanto, prohíben esta forma de interacción con el usuario. En el Capítulo 25 se describen modelos alternativos que proporcionan transacciones interactivas de larga duración.

15.3 Implementación de la atomicidad y la durabilidad

El componente de gestión de recuperaciones de un sistema de base de datos implementa el soporte para la atomicidad y durabilidad. En primer lugar consideramos un esquema simple pero extremadamente ineficiente, denominado **copia en la sombra**. Este esquema, que se basa en hacer copias de la base de datos, denominadas copias *sombra*, asume que sólo una transacción está activa en cada momento. El esquema asume que la base de datos es simplemente un archivo en disco. En disco se mantiene un puntero llamado puntero_bd que apunta a la copia actual de la base de datos.

En el esquema de copia en la sombra, una transacción que quiera actualizar una base de datos crea primero una copia completa de dicha base de datos. Todos los cambios se hacen en la nueva copia de la base de datos dejando la copia original, la **copia en la sombra**, inalterada. Si en cualquier punto hay que abortar la transacción, la copia nueva simplemente se borra. La copia antigua de la base de datos no se ve afectada.

Si la transacción se completa, se compromete como sigue. En primer lugar se consulta al sistema operativo para asegurarse de que todas las páginas de la nueva copia de la base de datos se han escrito en disco (en los sistemas Unix se usa el comando fsync para este propósito). Después de terminar este comando se actualiza el puntero puntero_bd para que apunte a la nueva copia de la base de datos; la nueva copia se convierte entonces en la copia de la base de datos actual. La copia antigua de la base de datos se borra después. El esquema se describe en la Figura 15.2, en la cual se muestra el estado de la base de datos antes y después de la actualización. Se dice que la transacción está *comprometida* en el momento en que puntero_bd actualizado se escribe en disco.

Considérese ahora la manera en que esta técnica trata los fallos de las transacciones y del sistema. En primer lugar, considérese un fallo en la transacción. Si la transacción falla en algún momento antes de actualizar puntero_bd, el contenido de la base de datos anterior no se ve afectado. Se puede abortar la transacción simplemente borrando la nueva copia de la base de datos. Una vez que se ha comprometido la transacción, puntero_bd apunta a todas las modificaciones que ésta ha realizado en la base de datos. De este modo, o todas las modificaciones de la transacción se ven reflejadas o ninguna de ellas, independientemente del fallo de la transacción.

Considérese ahora el resultado de un fallo en el sistema. Supóngase que el sistema falla en algún momento antes de escribir en disco el puntero_bd actualizado. Entonces cuando se reinicie el sistema, leerá puntero_bd y verá el contenido original de la base de datos, y ninguno de los efectos de la transacción será visible en la base de datos. A continuación supóngase que el sistema falla después de actualizar en disco puntero_bd. Antes de que el puntero se actualice, todas las páginas actualizadas de la nueva copia de la base de datos se escriben en disco. De nuevo, se asume que una vez que un archivo se escribe en disco, su contenido no se daña incluso si hay un fallo del sistema. Por tanto, cuando el sistema se reinicie, leerá puntero_bd y verá el contenido de la base de datos *después* de la transacción haya realizado todas las modificaciones.

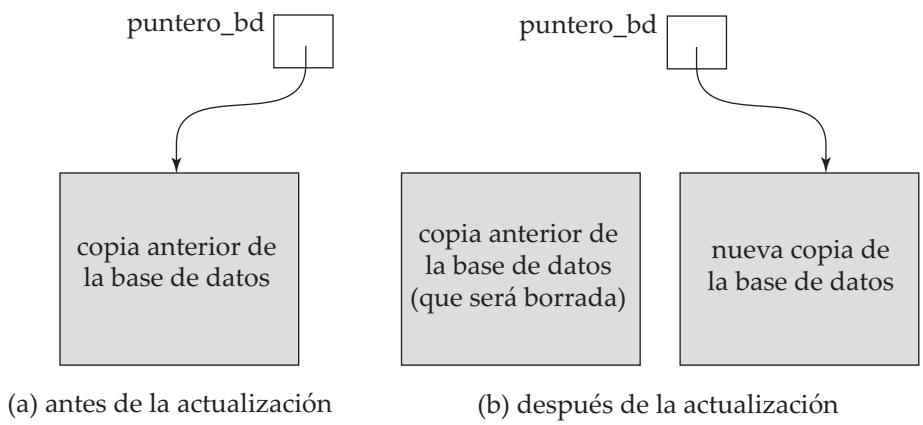


Figura 15.2 Técnica de copia en la sombra para la atomicidad y durabilidad.

La implementación depende realmente de que escribir puntero_bd sea una operación atómica; es decir, o se escriben todos sus bytes o ninguno de ellos. Si se escribieran algunos de los bytes del puntero y otros no, el puntero no tendría sentido y al reiniciarse el sistema no se podrían encontrar ni la versión anterior ni la nueva de la base de datos. Afortunadamente los sistemas de disco proporcionan actualizaciones atómicas de bloques enteros, o al menos de un sector del disco. En otras palabras, el sistema de disco garantiza la actualización automática de puntero_bd siempre que se cumpla que puntero_bd quepa en un único sector, lo que se puede asegurar almacenándolo al comienzo de un bloque.

De este modo, la implementación de la copia en la sombra del componente de gestión de recuperaciones asegura las propiedades de atomicidad y durabilidad de las transacciones.

Como ejemplo simple fuera del dominio de las bases de datos, considérese una sesión de edición de texto. Una sesión completa de edición de texto puede modelar una transacción. Las acciones que ejecuta la transacción son leer y actualizar el archivo. Guardar el archivo cuando se termina de editar significa completar la transacción de edición; terminar la sesión de edición sin guardar el archivo significa abortar la transacción de edición.

Muchos editores de texto usan fundamentalmente la implementación que se acaba de describir para asegurar que una sesión de edición es como una transacción. Se usa un nuevo archivo para almacenar las modificaciones. Al finalizar la sesión de edición, si el archivo modificado se va a guardar, se usa un comando para *renombrar* el nuevo archivo para que tenga el nombre del archivo actual. Se asume que el renombramiento está implementado como una operación atómica en el sistema de archivos, y que al mismo tiempo borra el archivo antiguo.

Desafortunadamente, esta implementación es extremadamente ineficiente en el contexto de grandes bases de datos, ya que la ejecución de una simple transacción requiere copiar la base de datos *completa*. Además, la implementación no permite a las transacciones ejecutarse concurrentemente unas con otras. Existen formas prácticas de implementar la atomicidad y durabilidad que son mucho menos costosas y más potentes. Estas técnicas se estudian en el Capítulo 17.

15.4 Ejecuciones concurrentes

Los sistemas de procesamiento de transacciones permiten normalmente la ejecución de varias transacciones concurrentemente. Permitir varias transacciones que actualizan concurrentemente los datos provoca complicaciones en la consistencia de los mismos, como se ha visto antes. Asegurar la consistencia a pesar de la ejecución concurrente de las transacciones requiere un trabajo extra; es mucho más sencillo exigir que las transacciones se ejecuten **secuencialmente**—es decir, una a una, comenzado cada una sólo después de que la anterior se haya completado. Sin embargo, existen dos buenas razones para permitir la concurrencia:

- **Productividad y utilización de recursos mejoradas**. Una transacción consiste en varios pasos. Algunos implican operaciones de E/S; otros implican operaciones de CPU. La CPU y los discos pueden trabajar en paralelo en una computadora. Por tanto, las operaciones de E/S se pueden realizar en paralelo con el procesamiento de la CPU. Se puede entonces explotar el paralelismo de la CPU y del sistema de E/S para ejecutar varias transacciones en paralelo. Mientras una transacción ejecuta una lectura o una escritura en un disco, otra puede ejecutarse en la CPU mientras una tercera transacción ejecuta una lectura o una escritura en otro disco. Todo esto incrementa la **productividad** (*throughput*) del sistema—es decir, en el número de transacciones que puede ejecutar en un tiempo dado. Análogamente, la **utilización** del procesador y del disco aumenta también; en otras palabras, el procesador y el disco están menos tiempo desocupados o sin hacer ningún trabajo útil.

- **Tiempo de espera reducido**. Debe haber una mezcla de transacciones que se ejecutan en el sistema, algunas cortas y otras largas. Si las transacciones se ejecutan secuencialmente, la transacción corta debe esperar a que la transacción larga anterior se complete, lo cual puede llevar a un retardo impredecible en la ejecución de la transacción. Si las transacciones operan en partes diferentes de la base de datos es mejor hacer que se ejecuten concurrentemente, compartiendo los ciclos de la CPU y los accesos a disco entre ambas. La ejecución concurrente reduce los retardos im-

predecibles en la ejecución de las transacciones. Además se reduce también el **tiempo medio de respuesta**: el tiempo medio desde que una transacción comienza hasta que se completa.

La razón para usar la ejecución concurrente en una base de datos es esencialmente la misma que para usar **multiprogramación** en un sistema operativo.

Cuando se ejecutan varias transacciones concurrentemente, la consistencia de la base de datos se puede destruir a pesar de que cada transacción individual sea correcta. En este apartado se presenta el concepto de planificaciones que ayudan a identificar aquellas ejecuciones que garantizan que se asegura la consistencia.

El sistema de base de datos debe controlar la interacción entre las transacciones concurrentes para evitar que se destruya la consistencia de la base de datos. Esto se lleva a cabo a través de una serie de mecanismos denominados **esquemas de control de concurrencia**. En el Capítulo 16 se estudian los esquemas de control de concurrencia; por ahora nos centraremos en el concepto de ejecución concurrente correcta.

Considérese de nuevo el sistema bancario simplificado del Apartado 15.1, el cual tiene varias cuentas, y un conjunto de transacciones que acceden y modifican dichas cuentas. Sean T_1 y T_2 dos transacciones para transferir fondos de una cuenta a otra. La transacción T_1 transfiere 50 € de la cuenta A a la cuenta B y se define como sigue

T_1: leer(A);
$A := A - 50$;
escribir(A);
leer(B);
$B := B + 50$;
escribir(B).

La transacción T_2 transfiere el 10 por ciento del saldo de la cuenta A a la cuenta B, y se define

T_2: leer(A);
$temp := A * 0.1$;
$A := A - temp$;
escribir(A);
leer(B);
$B := B + temp$;
escribir(B).

Supóngase que los valores actuales de las cuentas A y B son 1.000 € y 2.000 € respectivamente. Supóngase que las dos transacciones se ejecutan de una en una en el orden T_1 seguida de T_2. Esta secuencia de ejecución se representa en la Figura 15.3. En esta figura la secuencia de pasos o instrucciones aparece en orden cronológico de arriba abajo, con las instrucciones de T_1 en la columna izquierda y las de T_2 en la derecha. Los valores finales de las cuentas A y B, después de que tenga lugar la ejecución de la Figura 15.3, son de 855 € y de 2.145 € respectivamente. De este modo, la suma total de saldo de las cuentas A y B—es decir, la suma $A + B$—se conserva tras la ejecución de ambas transacciones.

Análogamente, si las transacciones se ejecutan de una en una en el orden T_2 seguida de T_1, entonces la secuencia de ejecución es la de la Figura 15.4. De nuevo, como se esperaba, se conserva la suma $A + B$ y los valores finales de las cuentas A y B son de 850 € y de 2.150 € respectivamente.

Las secuencias de ejecución que se acaban de describir se denominan **planificaciones**. Representan el orden cronológico en el que se ejecutan las instrucciones en el sistema. Obviamente una planificación para un conjunto de transacciones debe consistir en todas las instrucciones de dichas transacciones, y debe conservar el orden en que aparecen las instrucciones en cada transacción individual. Por ejemplo, en la transacción T_1 la instrucción escribir(A) debe aparecer antes de la instrucción leer(B) en cualquier planificación válida. En los párrafos siguientes, planificación 1 se referirá a la primera secuencia de ejecución (T_1 seguida de T_2) y planificación 2 a la segunda secuencia de ejecución (T_2 seguida de T_1).

Estas planificaciones son **secuenciales**. Cada planificación secuencial consiste en una secuencia de instrucciones de varias transacciones, en la cual las instrucciones pertenecientes a una única transac-

T_1	T_2
leer(A)	
A := A - 50	
escribir(A)	
leer(B)	
B := B + 50	
escribir(B)	
	leer(A)
	$temp$:= $A * 0,1$
	A := A - $temp$
	escribir(A)
	leer(B)
	B := B + $temp$
	escribir(B)

Figura 15.3 Planificación 1—una planificación secuencial en la que T_2 sigue a T_1.

ción están juntas en dicha planificación. De este modo, para un conjunto de n transacciones existen $n!$ planificaciones secuenciales válidas distintas.

Cuando el sistema de bases de datos ejecuta concurrentemente varias transacciones, la planificación correspondiente no tiene por qué ser secuencial. Si dos transacciones se ejecutan concurrentemente, el sistema operativo puede ejecutar una transacción durante un tiempo pequeño, luego realizar un cambio de contexto, ejecutar la segunda transacción durante un tiempo, cambiar de nuevo a la primera transacción durante un tiempo y así sucesivamente. Si hay muchas transacciones, todas ellas comparten el tiempo de la CPU.

Son posibles muchas secuencias de ejecución, puesto que varias instrucciones de ambas transacciones se pueden intercalar. En general no es posible predecir exactamente cuántas instrucciones se ejecutarán antes de que la CPU cambie a otra transacción. Así, el número de planificaciones posibles para un conjunto de n transacciones es mucho mayor que $n!$.

Volviendo al ejemplo anterior supóngase que las dos transacciones se ejecutan concurrentemente. Una posible planificación se muestra en la Figura 15.5. Una vez que la ejecución tiene lugar, se llega al mismo estado que cuando las transacciones se ejecutan secuencialmente en el orden T_1 seguida de T_2. La suma $A + B$ se conserva igualmente.

No todas las ejecuciones concurrentes producen un estado correcto. Como ejemplo de esto considérese la planificación de la Figura 15.6. Después de ejecutarse esta planificación se llega a un estado cuyos valores finales de las cuentas A y B son de 950 € y de 2.100 € respectivamente. Este estado final es

T_1	T_2
	leer(A)
	$temp$:= $A * 0,1$
	A := A - $temp$
	escribir(A)
	leer(B)
	B := B + $temp$
	escribir(B)
leer(A)	
A := A - 50	
escribir(A)	
leer(B)	
B := B + 50	
escribir(B)	

Figura 15.4 Planificación 2—una planificación secuencial en la cual T_1 sigue a T_2.

T_1	T_2
leer(A)	
A := A - 50	
escribir(A)	
	leer(A)
	$temp$:= $A * 0,1$
	A := A - $temp$
	escribir(A)
leer(B)	
B := B + 50	
escribir(B)	
	leer(B)
	B := B + $temp$
	escribir(B)

Figura 15.5 Planificación 3—una planificación concurrente equivalente a la planificación 1.

un *estado inconsistente*, ya que se han ganado 50 € al procesar la ejecución concurrente. Realmente la ejecución de las dos transacciones no conserva la suma $A + B$.

Si se deja el control de la ejecución concurrente completamente al sistema operativo son posibles muchas planificaciones, incluyendo las que dejan a la base de datos en un estado inconsistente, como la que se acaba de describir. Es una tarea del sistema de base de datos asegurar que cualquier planificación que se ejecute lleva a la base de datos a un estado consistente. El componente del sistema de base de datos que realiza esta tarea se denomina **componente de control de concurrencia**.

Se puede asegurar la consistencia de la base de datos en una ejecución concurrente si se está seguro de que cualquier planificación que se ejecute tiene el mismo efecto que otra que se hubiese ejecutado sin concurrencia. Es decir, la planificación debe ser, en cierto modo, equivalente a una planificación secuencial. Esta idea se examina en el Apartado 15.5.

15.5 Secuencialidad

El sistema de base de datos debe controlar la ejecución concurrente de las transacciones para asegurar que el estado de la base sigue siendo consistente. Antes de examinar cómo debe realizar esta tarea el sistema de base de datos hay que entender primero las planificaciones que aseguran la consistencia y las que no.

T_1	T_2
leer(A)	
A := A - 50	
	leer(A)
	$temp$:= $A * 0,1$
	A := A - $temp$
	escribir(A)
	leer(B)
escribir(A)	
leer(B)	
B := B + 50	
escribir(B)	
	B := B + $temp$
	escribir(B)

Figura 15.6 Planificación 4—una planificación concurrente.

Puesto que las transacciones son programas, es difícil calcular cuáles son las operaciones exactas que realiza una transacción y cómo interaccionan las operaciones de varias transacciones. Por este motivo no se van a interpretar los tipos de operaciones que puede realizar una transacción sobre un elemento de datos. En lugar de esto se consideran sólo dos operaciones: leer y escribir. Se asume así que entre una instrucción leer(Q) y otra escribir(Q) sobre un elemento de datos Q, una transacción puede realizar una secuencia arbitraria de operaciones con la copia Q que reside en la memoria intermedia local de dicha transacción. De este modo las únicas operaciones significativas de la transacción son, desde el punto de vista de la planificación, las instrucciones leer y escribir. Por tanto, normalmente sólo se mostrarán las instrucciones leer y escribir en las planificaciones, como se muestra en la planificación 3 de la Figura 15.7.

En este apartado se estudian diferentes formas de equivalencia de planificación; esto lleva a los conceptos de **secuencialidad en cuanto a conflictos** y **secuencialidad en cuanto a vistas**.

15.5.1 Secuencialidad en cuanto a conflictos

Considérese una planificación P en la cual hay dos instrucciones consecutivas I_i e I_j, pertenecientes a las transacciones T_i y T_j respectivamente ($i \neq j$). Si I_i e I_j se refieren a distintos elementos de datos se pueden intercambiar I_i e I_j sin afectar al resultado de cualquier instrucción de la planificación. Sin embargo, si I_i e I_j se refieren al mismo elemento Q entonces el orden de los dos pasos puede ser importante. Puesto que sólo se tienen en cuenta las instrucciones leer y escribir se deben considerar cuatro casos:

1. I_i = leer(Q), I_j = leer(Q). El orden de I_i e I_j no importa, puesto que T_i y T_j leen el mismo valor de Q, independientemente del orden.
2. I_i = leer(Q), I_j = escribir(Q). Si I_i está antes que I_j, entonces T_i no lee el valor de Q que escribe la instrucción I_j de T_j. Si I_j está antes que I_i, entonces T_i lee el valor de Q escrito por T_j. Por tanto, orden de I_i e I_j es importante.
3. I_i = escribir(Q), I_j = leer(Q). El orden de I_i e I_j es importante por razones similares a las del caso anterior.
4. I_i = escribir(Q), I_j = escribir(Q). Puesto que ambas instrucciones son operaciones escribir el orden de dichas instrucciones no afecta ni a T_i ni a T_j. Sin embargo, el valor que obtendrá la siguiente instrucción leer(Q) de P sí se ve afectado, ya que únicamente se conserva en la base de datos la última de las dos instrucciones escribir. Si no hay ninguna otra instrucción escribir(Q) después de I_i e I_j en P, entonces el orden de I_i e I_j afecta directamente al valor final de Q en el estado de la base de datos que se obtiene con la planificación P.

De esta manera sólo en el caso en el cual I_i e Ij son instrucciones leer no tiene importancia el orden de ejecución de las mismas.

Se dice que I_i e I_j están en **conflicto** si existen operaciones de diferentes transacciones sobre el mismo elemento de datos, y al menos una de esas instrucciones es una operación escribir.

T_1	T_2
leer(A)	
escribir(A)	
	leer(A)
	escribir(A)
leer(B)	
escribir(B)	
	leer(B)
	escribir(B)

Figura 15.7 Planificación 3—sólo se muestran las instrucciones leer y escribir.

T_1	T_2
leer(A)	
escribir(A)	
	leer(A)
leer(B)	
	escribir(A)
escribir(B)	
	leer(B)
	escribir(B)

Figura 15.8 Planificación 5—planificación 3 después de intercambiar un par de instrucciones.

Para ilustrar el concepto de instrucciones conflictivas considérese la planificación 3 mostrada en la Figura 15.7. La instrucción escribir(A) de T_1 está en conflicto con la instrucción leer(A) de T_2. Sin embargo, la instrucción escribir(A) de T_2 no está en conflicto con la instrucción leer(B) de T_1, ya que las dos instrucciones acceden a diferentes elementos de datos.

Sean I_i e I_j instrucciones consecutivas de una planificación P. Si I_i e I_j son instrucciones de transacciones diferentes y además I_i e I_j no están en conflicto, entonces se puede cambiar el orden de I_i e I_j para obtener una nueva planificación P'. Lo esperado es que P sea equivalente a P', ya que todas las instrucciones aparecen en el mismo orden en ambas planificaciones salvo I_i e I_j, cuyo orden no es importante.

Puesto que la instrucción escribir(A) de T_2 en la planificación 3 de la Figura 15.7 no está en conflicto con la instrucción leer(B) de T_1, se pueden intercambiar dichas instrucciones para generar una planificación equivalente, la planificación 5, que se muestra en la Figura 15.8. Independientemente de cuál sea el estado inicial del sistema, las planificaciones 3 y 5 producen el mismo estado final del sistema.

Se puede continuar intercambiando instrucciones no conflictivas como sigue:

- Intercambiar la instrucción leer(B) de T_1 con la instrucción leer(A) de T_2.
- Intercambiar la instrucción escribir(B) de T_1 con la instrucción escribir(A) de T_2.
- Intercambiar la instrucción escribir(B) de T_1 con la instrucción leer(A) de T_2.

El resultado final de estos intercambios, como se muestra en la Figura 15.9, es una planificación secuencial. Así se muestra que la planificación 3 es equivalente a una planificación secuencial. Esta equivalencia implica que independientemente del estado inicial, la planificación 3 produce el mismo estado final que una planificación secuencial.

Si una planificación P se puede transformar en otra P' por medio de una serie de intercambios de instrucciones no conflictivas, se dice que P y P' son **equivalentes en cuanto a conflictos**.

Volviendo a los ejemplos anteriores, se observa que la planificación 1 no es equivalente en cuanto a conflictos a la planificación 2. Sin embargo, la planificación 1 es equivalente en cuanto a conflictos a la

T_1	T_2
leer(A)	
escribir(A)	
leer(B)	
escribir(B)	
	leer(A)
	escribir(A)
	leer(B)
	escribir(B)

Figura 15.9 Planificación 6—una planificación secuencial equivalente a la planificación 3.

T_3	T_4
leer(Q)	
	escribir(Q)
escribir(Q)	

Figura 15.10 Planificación 7.

planificación 3, puesto que las instrucciones leer(B) y escribir(B) de T_1 se pueden intercambiar con las instrucciones leer(A) y escribir(A) de T_2.

El concepto de equivalencia en cuanto a conflictos lleva al concepto de secuencialidad en cuanto a conflictos. Se dice que una planificación P es **secuenciable en cuanto a conflictos** si es equivalente en cuanto a conflictos a una planificación secuencial. Así, la planificación 3 es secuenciable en cuanto a conflictos, ya que es equivalente en cuanto a conflictos a la planificación secuencial 1.

Finalmente considérese la planificación 7 de la Figura 15.10; sólo consta de las operaciones significativas de las transacciones T_3 y T_4 (es decir, leer y escribir). Esta planificación no es secuenciable en cuanto a conflictos, ya que no es equivalente ni a la planificación secuencial $<T_3,T_4>$ ni a $<T_4,T_3>$.

Es posible encontrar dos planificaciones que produzcan el mismo resultado y que no sean equivalentes en cuanto a conflictos. Por ejemplo, considérese la transacción T_5, la cual transfiere 10 € de la cuenta B a la A. Sea la planificación 8 la que se define en la Figura 15.11. Se puede afirmar que la planificación 8 no es equivalente en cuanto a conflictos a la planificación secuencial $<T_1,T_5>$, ya que en la planificación 8 la instrucción escribir(B) de T_5 está en conflicto con la instrucción leer(B) de T_1. Por este motivo no se pueden colocar todas las instrucciones de T_1 antes de las de T_5 intercambiando instrucciones no conflictivas consecutivas. Sin embargo, los valores finales de las cuentas A y B son los mismos después de ejecutar tanto la planificación 8 como la planificación secuencial $<T_1,T_5>$—esto es, 960 € y 2.040 € respectivamente.

Con este ejemplo se puede observar que existen definiciones de equivalencia de planificaciones que son menos rigurosas que la de equivalencia en cuanto a conflictos. Para que el sistema pueda determinar que el resultado de la planificación sea el mismo que el de la planificación secuencial $<T_1,T_5>$, debe analizar los cálculos realizados por T_1 y T_5, en lugar de tener sólo en cuenta las operaciones leer y escribir. En general es difícil de implementar tal análisis y es costoso en términos de cómputo. Sin embargo, existen otras definiciones de equivalencia de planificación que se basan únicamente en las operaciones leer y escribir. En el siguiente apartado se considera una de estas definiciones.

15.5.2 Secuencialidad en cuanto a vistas**

En este apartado se va a considerar una forma de equivalencia que es menos rigurosa que la equivalencia en cuanto a conflictos pero que, al igual que ésta, se basa únicamente en las operaciones leer y escribir de las transacciones.

T_1	T_5
leer(A)	
A := A - 50	
escribir(A)	
	leer(B)
	B := B - 10
	escribir(B)
leer(B)	
B := B + 50	
escribir(B)	
	leer(A)
	A := A + 10
	escribir(A)

Figura 15.11 Planificación 8.

Considérense dos planificaciones, P y P', en las cuales participa el mismo conjunto de transacciones. Se dice que las planificaciones P y P' son **equivalentes en cuanto a vistas** si se cumplen las tres condiciones siguientes:

1. Para todo elemento de datos Q, si la transacción T_i lee el valor inicial de Q en la planificación P, entonces T_i debe leer también el valor inicial de Q en la planificación P'.
2. Para todo elemento de datos Q, si la transacción T_i ejecuta leer(Q) en la planificación P y el valor lo ha producido la transacción T_j (si existe dicha transacción) entonces, en la planificación P', la transacción T_i debe leer también el valor de Q que haya producido la transacción T_j.
3. Para todo elemento de datos Q, la transacción (si existe) que realice la última operación escribir(Q) en la planificación P, debe realizar la última operación escribir(Q) en la planificación P'.

Las condiciones 1 y 2 aseguran que cada transacción lee los mismos valores en ambas planificaciones y por tanto realizan los mismos cálculos. La condición 3, junto con las condiciones 1 y 2, asegura que ambas planificaciones producen como resultado el mismo estado final del sistema.

Volviendo a los ejemplos anteriores, obsérvese que la planificación 1 no es equivalente en cuanto a vistas a la planificación 2, ya que en la planificación 1 el valor de la cuenta A que lee la transacción T_2 lo produce T_1, mientras que esto no ocurre en la planificación 2. Sin embargo, la planificación 1 es equivalente en cuanto a vistas a la planificación 3, ya que los valores de las cuentas A y B que lee la transacción T_2 los produce T_1 en ambas planificaciones.

El concepto de equivalencia en cuanto a vistas lleva al concepto de secuencialidad en cuanto a vistas. Se dice que la planificación P es **secuenciable en cuanto a vistas** si es equivalente en cuanto a vistas a una planificación secuencial.

Como ejemplo supóngase que se aumenta la planificación 7 con la transacción T_6 y se obtiene la planificación 9, tal y como se muestra en la Figura 15.12. La planificación 9 es secuenciable en cuanto a vistas. Realmente es equivalente en cuanto a vistas a la planificación secuencial $<T_3, T_4, T_6>$, ya que la instrucción leer(Q) lee el valor inicial de Q en ambas planificaciones, y T_6 realiza la escritura final de Q en ambas planificaciones.

Toda planificación secuenciable en cuanto a conflictos es secuenciable en cuanto a vistas, pero existen planificaciones secuenciables en cuanto a vistas que no son secuenciables en cuanto a conflictos. Realmente, la planificación 9 no es secuenciable en cuanto a conflictos puesto que para todo par de instrucciones, éstas están en conflicto y por tanto no es posible ningún intercambio de instrucciones.

Obsérvese que en la planificación 9, las transacciones T_4 y T_6 realizan operaciones escribir(Q) sin haber realizado ninguna operación leer(Q). Este tipo de escrituras se denominan **escrituras a ciegas**. Las escrituras a ciegas aparecen en toda planificación secuenciable en cuanto a vistas que no sea secuenciable en cuanto a conflictos.

15.6 Recuperabilidad

Antes se han estudiado las planificaciones que son aceptables desde el punto de vista de la consistencia de la base de datos asumiendo implícitamente que no había fallos en las transacciones. Ahora se va a estudiar el efecto de los fallos en una transacción durante una ejecución concurrente.

Si la transacción T_i falla, por la razón que sea, es necesario deshacer el efecto de dicha transacción para asegurar la propiedad de atomicidad de la misma. En un sistema que permita la concurrencia es necesario asegurar también que toda transacción T_j que dependa de T_i (es decir, T_j lee datos que ha

T_3	T_4	T_6
leer(Q)		
	escribir(Q)	
escribir(Q)		
		escribir(Q)

Figura 15.12 Planificación 9—una planificación secuenciable en cuanto a vistas.

T_8	T_9
leer(A)	
escribir(A)	
	leer(A)
leer(B)	

Figura 15.13 Planificación 10.

escrito T_i) se aborta también. Para alcanzar esta garantía, es necesario poner restricciones al tipo de planificaciones permitidas en el sistema.

En los dos subapartados siguientes se estudian las planificaciones que son aceptables desde el punto de vista que se ha descrito. Como ya se dijo antes, en el Capítulo 16 se describe la manera de asegurar que sólo se generan dichas planificaciones aceptables.

15.6.1 Planificaciones recuperables

Considérese la planificación 10 que se muestra en la Figura 15.13, en la cual la transacción T_9 realiza sólo una instrucción: leer(A). Supóngase que el sistema permite que T_9 se complete inmediatamente después de ejecutar la instrucción leer(A). Así se completa T_9 antes de que lo haga T_8. Supóngase ahora que T_8 falla antes de completarse. Puesto que T_9 ha leído el valor del elemento de datos A escrito por T_8, se debe abortar T_9 para asegurar la atomicidad de la transacción. Sin embargo, T_9 ya se ha comprometido y no puede abortarse. De este modo se llega a una situación en la cual es imposible recuperarse correctamente del fallo de T_8.

La planificación 10, cuyo compromiso tiene lugar inmediatamente después de ejecutar la instrucción leer(A), es un ejemplo de planificación *no recuperable*, la cual no debe permitirse. La mayoría de los sistemas de bases de datos requieren que todas las planificaciones sean *recuperables*. Una **planificación recuperable** es aquélla en la que para todo par de transacciones T_i y T_j tales que T_j lee elementos de datos que ha escrito previamente T_i, la operación comprometer de T_i aparece antes que la de T_j.

15.6.2 Planificaciones sin cascada

Incluso si una planificación es recuperable, hay que retroceder varias transacciones para recuperar correctamente el estado previo a un fallo en una transacción T_i. Tales situaciones ocurren si las transacciones leen datos que ha escrito T_i. Como ejemplo considérese la planificación parcial de la Figura 15.14. La transacción T_{10} escribe un valor de A que lee la transacción T_{11}. La transacción T_{11} escribe un valor de A que lee la transacción T_{12}. Supóngase que en ese momento falla T_{10}. Se debe retroceder T_{10}. Puesto que T_{11} depende de T_{10}, se debe retroceder T_{11}. Puesto que T_{12} depende de T_{11}, se debe retroceder T_{12}. Este fenómeno en el cual un fallo en una única transacción provoca una serie de retrocesos de transacciones se denomina **retroceso en cascada**.

No es deseable el retroceso en cascada, ya que provoca un aumento significativo del trabajo necesario para deshacer cálculos. Es deseable restringir las planificaciones a aquéllas en las que no puedan ocurrir retrocesos en cascada. Tales planificaciones se denominan planificaciones *sin cascada*. Una **planificación sin cascada** es aquélla para la que todo par de transacciones T_i y T_j tales que T_j lee un elemento de

T_{10}	T_{11}	T_{12}
leer(A)		
leer(B)		
escribir(A)		
	leer(A)	
	escribir(A)	
		leer(A)

Figura 15.14 Planificación 11.

datos que ha escrito previamente T_i, la operación comprometer de T_i aparece antes que la operación de lectura de T_j. Es sencillo comprobar que toda planificación sin cascada es también recuperable.

15.7 Implementación del aislamiento

Antes se han visto las propiedades que debe tener una planificación para dejar a la base de datos en un estado consistente y para permitir fallos en la transacción que se puedan manejar de una manera segura. En concreto, las planificaciones que son secuenciables en cuanto a conflictos o secuenciables en cuanto a vistas y sin cascada cumplen estos requisitos.

Existen varios **esquemas de control de concurrencia** que se pueden utilizar para asegurar que, incluso si se ejecutan concurrentemente muchas transacciones, sólo se generen planificaciones aceptables sin tener en cuenta la forma en que el sistema operativo comparte en el tiempo los recursos (tales como el tiempo de CPU) entre las transacciones.

Como ejemplo trivial de esquema de control de concurrencia considérese éste: una transacción realiza un **bloqueo** en la base de datos completa antes de comenzar y lo libera después de haberse comprometido. Mientras una transacción mantiene el bloqueo, no se permite que ninguna otra lo obtenga, y todas ellas deben esperar hasta que se libere el bloqueo. Como resultado de esta política de bloqueo, sólo se puede ejecutar una transacción cada vez. Por tanto, sólo se generan planificaciones secuenciales. Éstas son obviamente secuenciables y es sencillo probar que también son sin cascada.

Un esquema de control de concurrencia como éste produce un rendimiento pobre, ya que fuerza a que las transacciones esperen a que finalicen las precedentes para poder comenzar. En otras palabras, produce un grado de concurrencia pobre. Como se ha explicado en el Apartado 15.4, la ejecución concurrente presenta varios beneficios para el rendimiento.

El objetivo de los esquemas de control de concurrencia es proporcionar un elevado grado de concurrencia, al mismo tiempo que aseguran que todas las planificaciones que se generan son secuenciables en cuanto a conflictos o en cuanto a vistas y son sin cascada.

En el Capítulo 16 se estudian gran número de esquemas de control de concurrencia. Los esquemas adoptan diferentes compromisos en función del aumento de concurrencia que permiten y del aumento de coste en el que incurren. Algunos permiten que se generen planificaciones secuenciables en cuanto a conflictos; otros permiten que se generen planificaciones secuenciables en cuanto a vistas que no son secuenciables en cuanto a conflictos.

15.8 Comprobación de la secuencialidad

Cuando se diseñan esquemas de control de concurrencia es necesario que demostrar que las planificaciones que genera el esquema son secuenciables. Para lograrlo se debe entender primero la forma de determinar si, dada una planificación concreta P, es secuenciable.

A continuación se presenta un método simple y eficiente de determinar la secuencialidad en cuanto a conflictos de una planificación. Considérese una planificación P. Se construye un grafo dirigido, llamado **grafo de precedencia** para P. Este grafo consiste en un par $G = (V,A)$, siendo V un conjunto de vértices y A un conjunto de arcos. El conjunto de vértices consiste en todas las transacciones que participan en la planificación. El conjunto de arcos consiste en todos los arcos $T_i \rightarrow T_j$ para los cuales se dan una de las tres condiciones siguientes:

1. T_i ejecuta escribir(Q) antes de que T_j ejecute leer(Q).
2. T_i ejecuta leer(Q) antes de que T_j ejecute escribir(Q).
3. T_i ejecuta escribir(Q) antes de que T_j ejecute escribir(Q).

Si existe un arco $T_i \rightarrow T_j$ en el grafo de precedencia , entonces en toda planificación secuencial P' equivalente a P, T_i debe aparecer antes de T_j.

Por ejemplo, en la Figura 15.15a se muestra el grafo de precedencia de la planificación 1. Sólo contiene el arco $T_1 \rightarrow T_2$, puesto que todas las instrucciones de T_1 se ejecutan antes de que lo haga la primera de

Figura 15.15 Grafo de precedencia para (a) la planificación 1 y (b) la planificación 2.

T_2. Análogamente, la Figura 15.15b muestra el grafo de precedencia de la planificación 2 con el único arco $T_2 \rightarrow T_1$, ya que todas las instrucciones de T_2 se ejecutan antes de que lo haga la primera de T_1.

El grafo de precedencia de la planificación 4 se representa en la Figura 15.16. Contiene el arco $T_1 \rightarrow T_2$ debido a que T_1 ejecuta leer(A) antes de que T_2 ejecute escribir(A). También contiene el arco $T_2 \rightarrow T_1$ debido a que T_2 ejecuta leer(B) antes de que T_1 ejecute escribir(B).

Si el grafo de precedencia de P tiene un ciclo, entonces la planificación P no es secuenciable en cuanto a conflictos. Si el grafo no contiene ciclos, entonces la planificación P es secuenciable en cuanto a conflictos.

El **orden de secuencialidad** se puede obtener a través de la **ordenación topológica**, la cual determina un orden lineal que consiste en el orden parcial del grafo de precedencia. En general se pueden obtener muchos órdenes lineales posibles a través de la ordenación topológica. Por ejemplo, el grafo de la Figura 15.17a tiene dos órdenes lineales aceptables, como se observa en las Figuras 15.17b y 15.17c.

Así, para probar la secuencialidad en cuanto a conflictos es necesario construir el grafo de precedencia e invocar a un algoritmo de detección de ciclos. Los algoritmos de detección de ciclos se pueden consultar en cualquier libro de texto sobre algoritmos. Los algoritmos de detección de ciclos, tales como los que se basan en la búsqueda primero en profundidad, requieren del orden de n^2 operaciones, siendo n el número de vértices del grafo (es decir, el número de transacciones). De este modo se consigue un esquema práctico para determinar la secuencialidad en cuanto a conflictos.

Volviendo a los ejemplos anteriores, obsérvese que los grafos de precedencia para las planificaciones 1 y 2 (Figura 15.15) no contienen ciclos. El grafo de precedencia para la planificación 4 (Figura 15.16) sin embargo, contiene ciclos, lo que indica que esta planificación no es secuenciable en cuanto a conflictos.

La comprobación de la secuencialidad en cuanto a vistas es más complicada. De hecho, se ha demostrado que el problema de determinar la secuencialidad en cuanto a vistas es *NP*-completo. Por tanto, seguramente no exista ningún algoritmo eficiente para comprobar la secuencialidad en cuanto a vistas.

Véanse la notas bibliográficas para consultar referencias sobre ello. Sin embargo, los esquemas de control de concurrencia aún pueden utilizar *condiciones suficientes* para la secuencialidad en cuanto a vistas. Es decir, si se cumplen las condiciones suficientes, la planificación es secuencialidad en cuanto a vistas, pero puede haber algunas planificaciones secuencialidad en cuanto a vistas que no satisfagan las condiciones suficientes.

15.9 Resumen

- Una *transacción* es una *unidad* de la ejecución de un programa que accede y posiblemente actualiza varios elementos de datos. Es fundamental comprender el concepto de transacción para entender e implementar las actualizaciones de los datos en una base de datos, de manera que las ejecuciones concurrentes y los fallos de varios tipos no den como resultado que la base de datos se vuelva inconsistente.
- Es necesario que las transacciones tengan las propiedades ACID: atomicidad, consistencia, aislamiento y durabilidad.

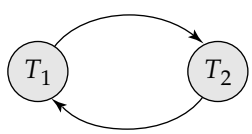


Figura 15.16 Grafo de precedencia para la planificación 4.

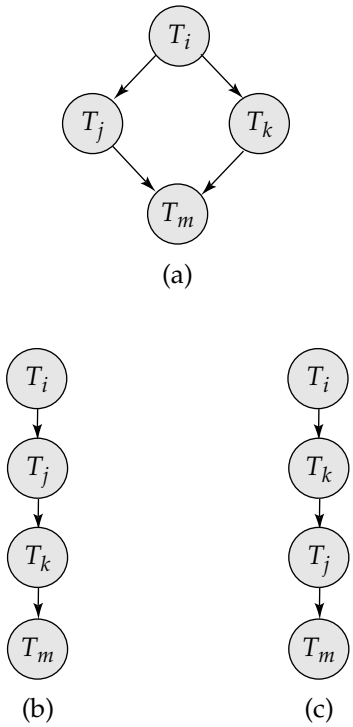


Figura 15.17 Ilustración de la ordenación topológica.

- La atomicidad asegura que, o bien todos los efectos de la transacción se reflejan en la base de datos, o bien ninguno de ellos; un fallo no puede dejar a la base de datos en un estado en el cual una transacción se haya ejecutado parcialmente.
- La consistencia asegura que si la base de datos es consistente inicialmente, la ejecución de la transacción (debido a la misma) deja la base de datos en un estado consistente.
- El aislamiento asegura que en la ejecución concurrente de transacciones, están aisladas entre si, de tal manera que cada una tiene la impresión de que ninguna otra transacción se ejecuta concurrentemente con ella.
- La durabilidad asegura que, una vez que la transacción se ha comprometido, las actualizaciones hechas por la transacción no se pierden incluso si hay un fallo del sistema.

- La ejecución concurrente de transacciones mejora la productividad y la utilización del sistema, y también reduce el tiempo de espera de las transacciones.

- Cuando varias transacciones se ejecutan concurrentemente en la base de datos, puede que deje de conservarse la consistencia de los datos. Es por tanto necesario que el sistema controle la interacción entre las transacciones concurrentes.
  - Puesto que una transacción es una unidad que conserva la consistencia, una ejecución secuencial de transacciones garantiza que se conserve dicha consistencia.
  - Una *planificación* captura las acciones clave de las transacciones que afectan a la ejecución concurrente, tales como las operaciones leer y escribir, a la vez que se abstraen los detalles internos de la ejecución de la transacción.
  - Es necesario que toda planificación producida por el procesamiento concurrente de un conjunto de transacciones tenga el efecto equivalente a una planificación en la cual esas transacciones se ejecutan secuencialmente en un cierto orden.
  - Un sistema que asegure esta propiedad se dice que asegura la *secuencialidad*.
  - Existen varias nociones distintas de equivalencia que llevan a los conceptos de *secuencialidad en cuanto a conflictos* y *secuencialidad en cuanto a vistas*.

- Se puede asegurar la secuencialidad de las planificaciones generadas por la ejecución concurrente de transacciones por medio de una gran variedad de mecanismos llamados esquemas de *control de concurrencia*.
- Las planificaciones deben ser recuperables para asegurar que si la transacción *a* observa los efectos de la transacción *b* y ésta aborta, entonces *a* también se aborte.
- Las planificaciones deben ser preferentemente sin cascada para que el hecho de abortar una transacción no provoque abortos en cascada de otras transacciones.
- El componente de gestión de control de concurrencia de la base de datos es el responsable de manejar los esquemas de control de concurrencia. En el Capítulo 16 se describen esquemas de control de concurrencia.
- El componente de gestión de recuperaciones de la base de datos es el responsable de asegurar las propiedades de las transacciones de atomicidad y durabilidad.

  El esquema de copia en la sombra se usa para asegurar la atomicidad y durabilidad en los editores de texto; sin embargo, tiene sobrecargas extremadamente altas cuando se usa en los sistemas de bases de datos y, más aún, no soporta la ejecución concurrente. El Capítulo 17 trata esquemas mejores.
- Se puede comprobar si una planificación es secuenciable en cuanto a conflictos construyendo el *grafo de precedencia* para dicha planificación y viendo que no hay ciclos en el grafo. Sin embargo, hay esquemas de control de concurrencia más eficientes para asegurar la secuencialidad.

Términos de repaso

- Transacción.
- Propiedades ACID:
  - Atomicidad.
  - Consistencia.
  - Aislamiento.
  - Durabilidad.
- Estado inconsistente.
- Estados de una transacción:
  - Activa.
  - Parcialmente comprometida.
  - Fallida.
  - Abortada.
  - Comprometida.
  - Terminada.
- Transacción.
  - Reiniciar.
  - Cancelar.
- Escrituras externas observables.
- Esquema de copia en la sombra.
- Ejecuciones concurrentes.
- Ejecución secuencial.
- Planificaciones.
- Conflictos entre operaciones.
- Equivalencia en cuanto a conflictos.
- Secuencialidad en cuanto a conflictos.
- Equivalencia en cuanto a vistas.
- Secuencialidad en cuanto a vistas.
- Escrituras a ciegas.
- Recuperabilidad.
- Planificaciones recuperables.
- Retroceso en cascada.
- Planificaciones sin cascada.
- Esquema de control de concurrencia.
- Bloqueo.
- Determinación de la secuencialidad.
- Grafo de precedencia.
- Orden de secuencialidad.

Ejercicios prácticos

15.1 Supóngase que existe un sistema de base de datos que nunca falla. ¿Se necesita un gestor de recuperaciones para este sistema?

15.2 Considérese un sistema de archivos como el de su sistema operativo preferido.

a. ¿Cuáles son los pasos involucrados en la creación y borrado de archivos, y en la escritura de datos a archivos?

b. Explíquese por qué son relevantes los aspectos de atomicidad y durabilidad en la creación y borrado de archivos, y en la escritura de datos a archivos.

15.3 Los implementadores de sistemas de bases de datos prestan mucha más atención a las propiedades ACID que los implementadores de sistemas de archivos. ¿Por qué tiene sentido esto?

15.4 Justifíquese lo siguiente. La ejecución concurrente de transacciones es más importante cuando los datos se deben extraer de disco (lento) o cuando las transacciones duran mucho, y es menos importante cuando hay pocos datos en memoria y las transacciones son muy cortas.

15.5 Puesto que toda planificación secuenciable en cuanto a conflictos es secuenciable en cuanto a vistas, ¿por qué se hace hincapié en la secuencialidad en cuanto a conflictos en vez de en la secuencialidad en cuanto a vistas?

15.6 Considérese el grafo de precedencia de la Figura 15.18. ¿Es secuenciable en cuanto a conflictos la planificación correspondiente? Razónese la respuesta.

15.7 ¿Qué es una planificación sin cascada? ¿Por qué es conveniente la planificación sin cascada? ¿Hay circunstancias bajo las cuales puede ser conveniente permitir planificaciones que no sean sin cascada? Razónese la respuesta.

Ejercicios

15.8 Lístense las propiedades ACID. Explíquese la utilidad de cada una.

15.9 Durante su ejecución, una transacción pasa a través de varios estados hasta que se compromete o aborta. Lístense todas las secuencias posibles de estados por los que puede pasar una transacción. Explíquese por qué puede ocurrir cada una de las transiciones de estados.

15.10 Explíquese la diferencia entre los términos *planificación secuencial* y *planificación secuenciable*.

15.11 Considérense las dos transacciones siguientes:

T_1: leer(A);
leer(B);
if $A = 0$ **then** $B := B + 1$;
escribir(B).

T_2: leer(B);
leer(A);
if $B = 0$ **then** $A := A + 1$;
escribir(A).

Sea el requisito de consistencia $A = 0 \vee B = 0$, siendo los valores iniciales $A = B = 0$.

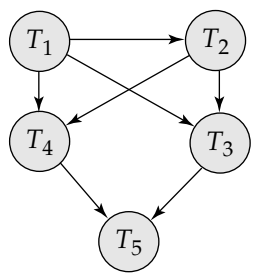


Figura 15.18 Grafo de precedencia.

a. Demuéstrese que toda ejecución secuencial en la que aparezcan estas transacciones conserva la consistencia de la base de datos.
b. Muéstrese una ejecución concurrente de T_1 y T_2 que produzca una planificación no secuenciable.
c. ¿Existe una ejecución concurrente de T_1 y T_2 que produzca una planificación secuenciable?

15.12 ¿Qué es una planificación recuperable? ¿Por qué es conveniente la recuperabilidad de las planificaciones? ¿Hay circunstancias bajo las cuales puede ser conveniente permitir planificaciones no recuperables? Razónese la respuesta.

15.13 ¿Por qué los sistemas de bases de datos permiten la ejecución concurrente de transacciones, a pesar del esfuerzo de programación necesario para asegurar que la ejecución concurrente no causa ningún problema?

Notas bibliográficas

Gray y Reuter [1993] proporcionan un extenso tratamiento de los conceptos y técnicas de procesamiento de transacciones, técnicas y detalles de implementación, incluyendo resultados de recuperación y control de concurrencia. Bernstein y Newcomer [1997] proporcionan varios aspectos del procesamiento de transacciones.

Los primeros estudios del control de concurrencia y la recuperación se incluyen en los libros Papadimitriou [1986] y Bernstein et al. [1987]. En Gray [1978] se presenta una visión de conjunto sobre resultados de implementación de control de concurrencia y recuperación.

El concepto de secuencialidad se formuló en Eswaran et al. [1976] en conexión con su trabajo sobre control de concurrencia para System R. Los resultados sobre la determinación de secuencialidad y NP-completitud de la determinación de la secuencialidad en cuanto a vistas son de Papadimitriou et al. [1977] y Papadimitriou [1979]. Los algoritmos de detección de ciclos se pueden encontrar en libros de algoritmos estándar, como en Cormen et al. [1990].

Otras referencias que cubren aspectos específicos de procesamiento de transacciones, tales como control de concurrencia y recuperación, se citan en los Capítulos 16, 17 y 25.

CAPÍTULO 16

Control de concurrencia

En el Capítulo 15 se estudió que una de las propiedades fundamentales de las transacciones es el aislamiento. Cuando se ejecutan varias transacciones concurrentemente en la base de datos, puede que deje de conservarse la propiedad de aislamiento. Es necesario que el sistema controle la interacción entre las transacciones concurrentes; dicho control se lleva a cabo a través de uno de los muchos mecanismos existentes llamado esquema de *control de concurrencia*.

Todos los esquemas de control de concurrencia que se describen en este capítulo se basan en la propiedad de secuencialidad. Es decir, todos los esquemas que se presentan aseguran que las planificaciones son secuenciables. En el Capítulo 25 se describen esquemas de control de concurrencia que admiten planificaciones no secuenciables. En este capítulo se trata la gestión de la ejecución concurrente de transacciones y se ignoran los fallos. En el Capítulo 17 se verá la manera en que el sistema se puede recuperar de los fallos.

16.1 Protocolos basados en el bloqueo

Una forma de asegurar la secuencialidad es exigir que el acceso a los elementos de datos se haga en exclusión mutua; es decir, mientras una transacción accede a un elemento de datos, ninguna otra transacción puede modificar dicho elemento. El método más habitual que se usa para implementar este requisito es permitir que una transacción acceda a un elemento de datos sólo si posee actualmente un bloqueo sobre dicho elemento.

16.1.1 Bloqueos

Existen muchos modos mediante los cuales se puede bloquear un elemento de datos. En este apartado se centra la atención en dos de dichos modos:

1. **Compartido**. Si una transacción T_i obtiene un **bloqueo en modo compartido** (denotado por C) sobre el elemento Q, entonces T_i puede leer Q pero no lo puede escribir.
2. **Exclusivo**. Si una transacción T_i obtiene un **bloqueo en modo exclusivo** (denotado por X) sobre el elemento Q, entonces T_i puede tanto leer como escribir Q.

Es necesario que toda transacción **solicite** un bloqueo del modo apropiado sobre el elemento de datos Q dependiendo de los tipos de operaciones que se vayan a realizar sobre Q. La petición se hace al gestor de control de concurrencia. La transacción puede realizar la operación sólo después de que el gestor de control de concurrencia **conceda** el bloqueo a la transacción.

Dado un conjunto de modos de bloqueo, se puede definir sobre ellos una **función de compatibilidad** como se explica a continuación. Se usarán A y B para representar dos modos de bloqueo arbitrarios. Supóngase que la transacción T_i solicita un bloqueo en modo A sobre el elemento Q bajo el que la

	C	X
C	cierto	falso
X	falso	falso

Figura 16.1 Matriz de compatibilidad de bloqueos comp.

transacción T_j ($T_i \neq T_j$) posee actualmente un bloqueo de modo B. Si a la transacción T_i se le puede conceder un bloqueo sobre Q a pesar de la presencia del bloqueo de modo B, entonces se dice que el modo A es **compatible** con el modo B. Tal función se puede representar convenientemente en forma de matriz. La relación de compatibilidad entre los dos modos de bloqueo que se usan en este apartado se presenta en la matriz comp de la Figura 16.1. Un elemento comp(A, B) de la matriz tiene el valor *cierto* si y sólo si el modo A es compatible con el modo B.

Obsérvese que el modo compartido es compatible consigo mismo, pero no con el modo exclusivo. En todo momento se pueden tener varios bloqueos en modo compartido (por varias transacciones) sobre un elemento de datos en concreto. Una petición posterior de bloqueo en modo exclusivo debe esperar hasta que se liberen los bloqueos en modo compartido que se poseen actualmente.

Una transacción solicita un bloqueo compartido sobre el elemento de datos Q a través de la instrucción bloquear-C(Q). De forma similar se solicita un bloqueo exclusivo a través de la instrucción bloquear-X(Q). Se puede desbloquear un elemento de datos Q por medio de la instrucción desbloquear(Q).

Para acceder a un elemento de datos, una transacción T_i debe en primer lugar bloquear dicho elemento. Si éste ya se encuentra bloqueado por otra transacción en un modo incompatible, el gestor de control de concurrencia no concederá el bloqueo hasta que todos los bloqueos incompatibles que posean otras transacciones hayan sido liberados. De este modo T_i debe **esperar** hasta que se liberen todos los bloqueos incompatibles que posean otras transacciones.

La transacción T_i puede desbloquear un elemento de datos que haya bloqueado en algún momento anterior. Obsérvese que la transacción debe mantener un bloqueo sobre un elemento de datos mientras acceda a dicho elemento. Además, no siempre es aconsejable que una transacción desbloquee un elemento de datos inmediatamente después de finalizar su acceso sobre él, ya que puede dejar de asegurarse la secuencialidad.

Como ejemplo, considérese de nuevo el sistema bancario simplificado que se presentó en el Capítulo 15. Sean A y B dos cuentas a las que acceden las transacciones T_1 y T_2. La transacción T_1 transfiere 50 € desde la cuenta B a la A (Figura 16.2). La transacción T_2 visualiza la cantidad total de dinero de las cuentas A y B—es decir, la suma $A + B$ (Figura 16.3).

Supóngase que los valores de las cuentas A y B son 100 € y 200 € respectivamente. Si estas dos transacciones se ejecutan secuencialmente, tanto en el orden T_1, T_2 como en el orden T_2, T_1, entonces la transacción T_2 visualizará el valor 300 €. Si por el contrario estas transacciones se ejecutan concurrentemente, entonces puede darse la planificación 1, que se muestra en la Figura 16.4. En ese caso la transacción T_2 visualiza 250 €, lo cual es incorrecto. El motivo por el que se produce esta incorrección es que la transacción T_1 desbloquea el elemento B demasiado pronto, lo cual provoca que T_2 perciba un estado inconsistente.

```
T1: bloquear-X(B);
    leer(B);
    B := B − 50;
    escribir(B);
    desbloquear(B);
    bloquear-X(A);
    leer(A);
    A := A + 50;
    escribir(A);
    desbloquear(A).
```

Figura 16.2 Transacción T_1.

```
T2: bloquear-C(A);
    leer(A);
    desbloquear(A);
    bloquear-C(B);
    leer(B);
    desbloquear(B);
    visualizar(A + B).
```

Figura 16.3 Transacción T_2.

La planificación muestra las acciones que ejecuta cada transacción, así como los puntos en los que el gestor de control de concurrencia concede los bloqueos. La transacción que realiza una petición de bloqueo no puede ejecutar su siguiente acción hasta que el gestor de control de concurrencia conceda dicho bloqueo. Por tanto, el bloqueo debe concederse en el intervalo de tiempo entre la operación de petición del bloqueo y la siguiente acción de la transacción. No es importante el momento exacto del intervalo en el cual se produce la concesión del bloqueo; se puede asumir sin problemas que la concesión del bloqueo se produce justo antes de la siguiente acción de la transacción. Por tanto se va a obviar la columna que describe las acciones del gestor de control de concurrencia en todas las planificaciones que aparezcan en el resto del capítulo. Se deja al lector como ejercicio la deducción del momento en que se conceden los bloqueos.

Supóngase ahora que el desbloqueo se retrasa hasta el final de la transacción. La transacción T_3 corresponde a T_1 con el desbloqueo retrasado (Figura 16.5). La transacción T_4 corresponde a T_2 con el desbloqueo retrasado (Figura 16.6).

Se puede verificar que la secuencia de lecturas y escrituras de la planificación 1, que provoca que se visualice un total incorrecto de 250 €, ya no es posible con T_3 y T_4. Son posibles otras planificaciones. T_4 no visualiza un resultado inconsistente en ninguna de ellas; más adelante se verá por qué.

Desafortunadamente, el uso de bloqueos puede conducir a una situación no deseada. Considérese la planificación parcial de la Figura 16.7 para T_3 y T_4. Puesto que T_3 posee en bloqueo sobre B en modo

T_1	T_2	gestor de control de concurrencia
bloquear-X(B)		
		conceder-X(B, T_1)
leer(B)		
B:= B - 50		
escribir(B)		
desbloquear(B)		
	bloquear-C(A)	
		conceder-C(A, T_2)
	leer(A)	
	desbloquear(A)	
	bloquear-C(B)	
		conceder-C(B, T_2)
	leer(B)	
	desbloquear(B)	
	visualizar(A + B)	
bloquear-X(A)		
		conceder-X(A, T_1)
leer(A)		
A:= A + 50		
escribir(A)		
desbloquear(A)		

Figura 16.4 Planificación 1.

```
T3: bloquear-X(B);
    leer(B);
    B := B − 50;
    escribir(B);
    bloquear-X(A);
    leer(A);
    A := A + 50;
    escribir(A);
    desbloquear(B);
    desbloquear(A).
```

Figura 16.5 Transacción T_3.

exclusivo y T_4 solicita un bloqueo sobre B en modo compartido, T_4 espera a que T_3 desbloquee B. De forma similar, puesto que T_4 posee un bloqueo sobre A en modo compartido y T_3 solicita un bloqueo sobre A en modo exclusivo, T_3 espera a que T_4 desbloquee A. Así se llega a un estado en el cual ninguna de las transacciones puede continuar su ejecución normal. Esta situación se denomina **interbloqueo**. Cuando aparece un interbloqueo, el sistema debe retroceder una de las dos transacciones. Una vez que se ha provocado el retroceso de una de ellas, se desbloquean los elementos de datos que estuvieran bloqueados por la transacción. Estos elementos de datos están disponibles entonces para otra transacción, la cual puede continuar su ejecución. Se volverá al tratamiento de interbloqueos en el Apartado 16.6.

Si no se usan bloqueos, o se desbloquean los elementos de datos tan pronto como sea posible después de leerlos o escribirlos, se pueden obtener estados inconsistentes. Por otro lado, si no se desbloquea un elemento de datos antes de solicitar un bloqueo sobre otro, pueden producirse interbloqueos. Existen formas de evitar los interbloqueos en algunas situaciones, como se verá en el Apartado 16.1.5. Sin embargo, en general, los interbloqueos son un mal necesario asociado a los bloqueos si se quieren evitar los estados inconsistentes. Los interbloqueos son absolutamente preferibles a los estados inconsistentes, ya que se pueden tratar haciendo retroceder las transacciones, mientras que los estados inconsistentes producen problemas en el mundo real que el sistema de base de datos no puede manejar.

Se exige que toda transacción del sistema siga un conjunto de reglas llamado **protocolo de bloqueo**, que indica el momento en que una transacción puede bloquear y desbloquear cada uno de los elementos de datos. Los protocolos de bloqueo restringen el número de planificaciones posibles. El conjunto de

```
T4: bloquear-C(A);
    leer(A);
    bloquear-C(B);
    leer(B);
    visualizar(A + B);
    desbloquear(A);
    desbloquear(B).
```

Figura 16.6 Transacción T_4.

T_3	T_4
bloquear-X(B)	
leer(B)	
B:= B - 50	
escribir(B)	
	bloquear-C(A)
	leer(A)
	bloquear-C(B)
bloquear-X(A)	

Figura 16.7 Planificación 2.

tales planificaciones es un subconjunto propio de todas las planificaciones secuenciables posibles. Se van a mostrar varios protocolos de bloqueo que sólo permiten planificaciones secuenciables en cuanto a conflictos. Antes de hacer esto se necesitan algunas definiciones.

Sean $\{T_0, T_1, \ldots, T_n\}$ un conjunto de transacciones que participan en la planificación S. Se dice que T_i **precede** a T_j en S, denotado por $T_i \rightarrow T_j$, si existe un elemento de datos Q tal que T_i ha obtenido un bloqueo en modo A sobre Q, y T_j ha obtenido un bloqueo en modo B sobre Q más tarde y comp(A, B) = falso. Si $T_i \rightarrow T_j$ entonces esta precedencia implica que en cualquier planificación secuencial equivalente, T_i debe aparecer antes que T_j. Obsérvese que este grafo es similar al grafo de precedencia que se usaba en el Apartado 15.8 para comprobar la secuencialidad en cuanto a conflictos. Los conflictos entre instrucciones corresponden a modos de bloqueo no compatibles.

Se dice que una planificación S es **legal** bajo un protocolo de bloqueo dado si S es una planificación posible para un conjunto de transacciones que sigan las reglas del protocolo de bloqueo. Se dice que un protocolo **asegura** la secuencialidad en cuanto a conflictos si y sólo si todas las planificaciones legales son secuenciables en cuanto a conflictos; en otras palabras, para todas las planificaciones legales la relación $\rightarrow$ asociada es acíclica.

16.1.2 Concesión de bloqueos

Cuando una transacción solicita un bloqueo de un modo particular sobre un elemento de datos y ninguna otra transacción posee un bloqueo sobre el mismo elemento de datos en un modo conflictivo, se puede conceder el bloqueo. Sin embargo, hay que tener cuidado para evitar la siguiente situación. Supóngase que la transacción T_2 posee un bloqueo en modo compartido sobre un elemento de datos y que la transacción T_1 solicita un bloqueo en modo exclusivo sobre dicho elemento de datos. Obviamente T_1 debe esperar a que T_2 libere el bloqueo en modo compartido. Mientras tanto, la transacción T_3 puede solicitar un bloqueo en modo compartido sobre el mismo elemento de datos. La petición de bloqueo es compatible con el bloqueo que se ha concedido a T_2, por tanto se puede conceder a T_3 el bloqueo en modo compartido. En este punto T_2 puede liberar el bloqueo, pero T_1 debe seguir esperando hasta que T_3 termine. Pero de nuevo puede haber una nueva transacción T_4 que solicite un bloqueo en modo compartido sobre el mismo elemento de datos y a la cual se conceda el bloqueo antes de que T_3 lo libere. De hecho, es posible que haya una secuencia de transacciones que soliciten un bloqueo en modo compartido sobre el elemento de datos, y que cada una de ellas libere el bloqueo un poco después de que sea concedido, de forma que T_1 nunca obtenga el bloqueo en modo exclusivo sobre el elemento de datos. La transacción T_1 nunca progresa y se dice que tiene **inanición**.

Se puede evitar la inanición de las transacciones al conceder los bloqueos de la siguiente manera: cuando una transacción T_i solicita un bloqueo sobre un elemento de datos Q en un modo particular M, el gestor de control de concurrencia concede el bloqueo siempre que

1. No exista otra transacción que posea un bloqueo sobre Q en un modo que esté en conflicto con M.
2. No exista otra transacción que esté esperando un bloqueo sobre Q y que lo haya solicitado antes que T_i.

De este modo, una petición de bloqueo nunca se quedará bloqueada por otra petición de bloqueo solicitada más tarde.

16.1.3 Protocolo de bloqueo de dos fases

Un protocolo que asegura la secuencialidad es el **protocolo de bloqueo de dos fases**. Este protocolo exige que cada transacción realice las peticiones de bloqueo y desbloqueo de dos fases:

1. **Fase de crecimiento**. Una transacción puede obtener bloqueos pero no puede liberarlos.
2. **Fase de decrecimiento**. Una transacción puede liberar bloqueos pero no puede obtener ninguno nuevo.

Inicialmente una transacción está en la fase de crecimiento. La transacción adquiere los bloqueos que necesite. Una vez que la transacción libera un bloqueo, entra en la fase de decrecimiento y no puede realizar más peticiones de bloqueo.

Por ejemplo, las transacciones T_3 y T_4 son de dos fases. Por otro lado, las transacciones T_1 y T_2 no son de dos fases. Obsérvese que no es necesario que las instrucciones de desbloqueo aparezcan al final de la transacción. Por ejemplo, en el caso de la transacción T_3 se puede trasladar la instrucción desbloquear(B) hasta justo antes de la instrucción bloquear-X(A) y se sigue cumpliendo la propiedad del bloqueo de dos fases.

Se puede mostrar que el protocolo de bloqueo de dos fases asegura la secuencialidad en cuanto a conflictos. Considérese cualquier transacción. El punto de la planificación en el cual la transacción obtiene su bloqueo final (el final de la fase de crecimiento) se denomina **punto de bloqueo** de la transacción. Ahora se pueden ordenar las transacciones según sus puntos de bloqueo—de hecho, esta ordenación es un orden de secuencialidad para las transacciones. La demostración se deja como ejercicio para el lector (véase el Ejercicio práctico 16.1).

El protocolo de bloqueo de dos fases *no* asegura la ausencia de interbloqueos. Obsérvese que las transacciones T_3 y T_4 son de dos fases pero en la planificación 2 (Figura 16.7) llegan a un interbloqueo.

Recuérdese del Apartado 15.6.2 que las planificaciones, además de ser secuenciables, es aconsejable que sean sin cascada. El retroceso en cascada puede ocurrir en el bloqueo de dos fases. Como ejemplo considérese la planificación parcial de la Figura 16.8. Cada transacción sigue el protocolo de bloqueo de dos fases pero un fallo de T_5 después del paso leer(A) de T_7 lleva a un retroceso en cascada de T_6 y T_7.

Los retrocesos en cascada se pueden evitar por medio de una modificación del protocolo de bloqueo de dos fases que se denomina **protocolo de bloqueo estricto de dos fases**. Este protocolo exige que, además de que el bloqueo sea de dos fases, una transacción deba poseer todos los bloqueos en modo exclusivo que tome hasta que dicha transacción se complete. Este requisito asegura que todo dato que escribe una transacción no comprometida está bloqueado en modo exclusivo hasta que la transacción se completa, evitando que ninguna otra transacción lea el dato.

Otra variante del bloqueo de dos fases es el **protocolo de bloqueo riguroso de dos fases**, el cual exige que se posean todos los bloqueos hasta que se comprometa la transacción. Se puede comprobar fácilmente que con el bloqueo riguroso de dos fases se pueden secuenciar las transacciones en el orden en que se comprometen. Muchos sistemas de bases de datos implementan tanto el bloqueo estricto como el bloqueo estricto de dos fases.

Considérense las dos transacciones siguientes de las que sólo se muestran algunas de las operaciones leer y escribir significativas:

T_5	T_6	T_7
bloquear-X(A)		
leer(A)		
bloquear-C(B)		
leer(B)		
escribir(A)		
desbloquear(A)		
	bloquear-X(A)	
	leer(A)	
	escribir(A)	
	desbloquear(A)	
		bloquear-C(A)
		leer(A)

Figura 16.8 Planificación parcial con bloqueo de dos fases.

T_8: leer(a_1);
leer(a_2);
. . .
leer(a_n);
escribir(a_1).

T_9: leer(a_1);
leer(a_2);
visualizar($a_1 + a_2$).

Si se emplea el protocolo de bloqueo de dos fases entonces T_8 debe bloquear a_1 en modo exclusivo. Por tanto, toda ejecución concurrente de ambas transacciones conduce a una ejecución secuencial. Obsérvese sin embargo que T_8 sólo necesita el bloqueo en modo exclusivo sobre a_1 al final de su ejecución, cuando escribe a_1. Así, si T_8 pudiera bloquear inicialmente a_1 en modo compartido y después pudiera cambiar el bloqueo a modo exclusivo, se obtendría una mayor concurrencia, ya que T_8 y T_9 podrían acceder a a_1 y a_2 simultáneamente.

Esta observación lleva a un refinamiento del protocolo de bloqueo de dos fases básico en el cual se permiten **conversiones de bloqueo**. Se va a proporcionar un mecanismo para cambiar un bloqueo compartido por un bloqueo exclusivo y un bloqueo exclusivo por uno compartido. Se denota la conversión del modo compartido al modo exclusivo como **subir**, y la conversión del modo exclusivo al modo compartido como **bajar**. No se puede permitir la conversión de modos arbitrariamente. Por el contrario, la subida puede tener lugar sólo en la fase de crecimiento, mientras que la bajada puede tener lugar sólo en la fase de decrecimiento.

Volviendo al ejemplo, las transacciones T_8 y T_9 se pueden ejecutar concurrentemente bajo el protocolo de bloqueo de dos fases refinado, como muestra la planificación incompleta de la Figura 16.9, en la cual sólo se muestran algunas de las operaciones de bloqueo.

Obsérvese que se puede forzar a esperar a una transacción que intente subir un bloqueo sobre un elemento Q. Esta espera forzada tiene lugar si Q está bloqueado actualmente por *otra* transacción en modo compartido.

Del mismo modo que el protocolo de bloqueo de dos fases básico, el bloqueo de dos fases con conversión de bloqueos genera sólo planificaciones secuenciables en cuanto a conflictos y se pueden secuenciar las transacciones según sus puntos de bloqueo. Además, si se poseen los bloqueos hasta el final de la transacción, las planificaciones son sin cascada.

Para un conjunto de transacciones puede haber planificaciones secuenciables en cuanto a conflictos que no se puedan obtener por medio del protocolo de bloqueo de dos fases. Sin embargo, para obtener planificaciones secuenciables en cuanto a conflictos por medio de protocolos de bloqueo que no sean de dos fases, es necesario o bien tener información adicional de las transacciones o bien imponer alguna estructura u orden en el conjunto de elementos de datos de la base de datos. A falta de esta información es necesario el bloqueo de dos fases para la secuencialidad en cuanto a conflictos—si T_i no es una

T_8	T_9
bloquear-C(a_1)	
	bloquear-C(a_1)
bloquear-C(a_2)	
	bloquear-C(a_2)
bloquear-C(a_3)	
bloquear-C(a_4)	
	desbloquear(a_1)
	desbloquear(a_2)
bloquear-C(a_n)	
subir(a_1)	

Figura 16.9 Planificación incompleta con conversión de bloqueos.

transacción de dos fases, siempre es posible encontrar otra transacción T_j que sea de dos fases tal que existe una planificación posible para T_i y T_j que no sea secuenciable en cuanto a conflictos.

Los bloqueos estricto de dos fases y riguroso de dos fases (con conversión de bloqueos) se usan ampliamente en sistemas de bases de datos comerciales.

Un esquema simple pero de uso extendido genera automáticamente las instrucciones apropiadas de bloqueo y desbloqueo para una transacción, basándose en peticiones de lectura y escritura desde la transacción:

- Cuando una transacción T_i realiza una operación leer(Q), el sistema genera una instrucción bloquear-C(Q) seguida de una instrucción leer(Q).
- Cuando T_i realiza una operación escribir(Q), el sistema comprueba si T_i posee ya un bloqueo en modo compartido sobre Q. Si es así, entonces el sistema genera una instrucción subir(Q) seguida de la instrucción escribir(Q). En otro caso el sistema genera una instrucción bloquear-X(Q) seguida de la instrucción escribir(Q).
- Todos los bloqueos que obtenga una transacción no se desbloquean hasta que dicha transacción se comprometa o aborte.

16.1.4 Implementación de los bloqueos**

Un **gestor de bloqueos** se puede implementar como un proceso que recibe mensajes de transacciones y envía mensajes como respuesta. El proceso gestor de bloqueos responde a los mensajes de solicitud de bloqueo con mensajes de concesión de bloqueo, o con mensajes que solicitan un retroceso de la transacción (en caso de interbloqueos). Los mensajes de desbloqueo tan sólo requieren un reconocimiento como respuesta, pero pueden dar lugar a un mensaje de concesión para otra transacción que esté esperando.

El gestor de bloqueos utiliza la siguiente estructura de datos: para cada elemento de datos que está actualmente bloqueado, mantiene una lista enlazada de registros, uno para cada solicitud, en el orden en el que llegaron las solicitudes. Utiliza una tabla de asociación, indexada por el nombre del elemento de datos, para encontrar la lista enlazada (si la hay) para cada elemento de datos; esta tabla se denomina **tabla de bloqueos**. Cada registro de la lista enlazada de cada elemento de datos anota qué transacción hizo la solicitud, y qué modo de bloqueo se solicitó. El registro también anota si la solicitud ya se ha concedido.

La Figura 16.10 muestra un ejemplo de una tabla de bloqueos. La tabla contiene bloqueos para cinco elementos de datos distintos: E4, E7, E23, E44 y E912. La tabla de bloqueos utiliza cadenas de desbordamiento, de forma que hay una lista enlazada de elementos de datos por cada entrada de la tabla de bloqueos. También hay una lista de transacciones a las que se les han concedido bloqueos, o que están esperando para bloquear, para cada uno de los elementos de datos. Los bloqueos concedidos se han representado como rectángulos rellenos (negros), mientras que las solicitudes en espera son los rectángulos vacíos. Se ha omitido el modo de bloqueo para que la figura sea más sencilla. Por ejemplo, se puede observar que a T23 se le han concedido bloqueos sobre E912 y E7, y está esperando para bloquear E4.

Aunque la figura no lo muestre, la tabla de bloqueos debería mantener también un índice de identificadores de transacciones, de forma que fuese posible determinar de manera eficiente el conjunto de bloqueos que mantiene una transacción dada.

El gestor de bloqueos procesa las solicitudes de la siguiente forma:

- Cuando llega un mensaje de solicitud, añade un registro al final de la lista enlazada del elemento de datos, si la lista enlazada existe. En otro caso crea una nueva lista enlazada que tan sólo contiene el registro correspondiente a la solicitud.

  Siempre concede la primera solicitud de bloqueo sobre el elemento de datos. Pero si la transacción solicita un bloqueo sobre un elemento sobre el cual ya se ha concedido un bloqueo, el gestor de bloqueos sólo concede la solicitud si es compatible con las solicitudes anteriores, y todas éstas ya se han concedido. En otro caso, la solicitud tiene que esperar.

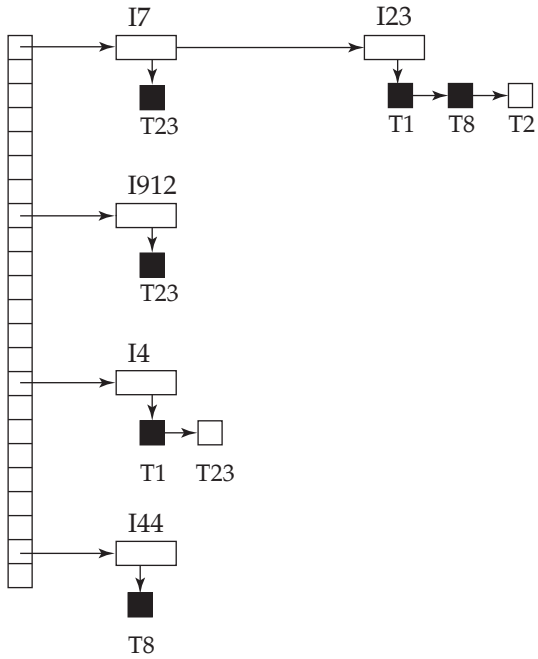


Figura 16.10 Tabla de bloqueos.

- Cuando el gestor de bloqueos recibe un mensaje de desbloqueo de una transacción, borra el registro para ese elemento de datos de la lista enlazada correspondiente a dicha transacción. Prueba el siguiente registro, si lo hay, como se describe en el párrafo anterior, para determinar si ahora se puede conceder dicha solicitud. Si se puede, el gestor de bloqueos concede la solicitud y procesa el siguiente registro, si lo hay, de forma similar, y así sucesivamente.
- Si una transacción se interrumpe, el gestor de bloqueos borra cualquier solicitud en espera realizada por la transacción. Una vez que el sistema de base de datos ha realizado las acciones apropiadas para deshacer la transacción (véase el Apartado 17.3), libera todos los bloqueos que mantenía la transacción abortada.

Este algoritmo garantiza que las solicitudes de bloqueo están libres de inanición, dado que una solicitud nunca se puede conceder mientras no se hayan concedido las solicitudes recibidas anteriormente. Más adelante, en el Apartado 16.6.3, se estudiará cómo detectar y manejar interbloqueos. El Apartado 20.2.1 describe una implementación alternativa—una que utiliza memoria compartida en vez de paso de mensajes para la solicitud y concesión de bloqueos.

16.1.5 Protocolos basados en grafos

Como se ha visto en el Apartado 16.1.3, a falta de información acerca de la forma en que se accede a los elementos de datos, el protocolo de bloqueo de dos fases es necesario y suficiente para asegurar la secuencialidad. Pero, si se desean desarrollar protocolos que no sean de dos fases, es necesario tener información adicional acerca de la forma en que cada transacción accede a la base de datos. Existen varios modelos que pueden ofrecer la información adicional, que difieren en la cantidad de información que proporcionan. El modelo más simple exige que se tenga un conocimiento previo acerca del orden en el cual se accede a los elementos de la base de datos. Dada esta información es posible construir protocolos de bloqueo que no sean de dos fases pero que no obstante aseguren la secuencialidad.

Para adquirir tal conocimiento previo, se impone un orden parcial $\rightarrow$ sobre el conjunto $\mathbf{D} = \{d_1, d_2, \ldots, d_h\}$ de todos los elementos de datos. Si $d_i \rightarrow d_j$ entonces toda transacción que acceda tanto a d_i como a d_j debe acceder a d_i antes de acceder a d_j. Este orden parcial puede ser el resultado de la organización tanto lógica como física de los datos, o se puede imponer únicamente con motivo del control de concurrencia.

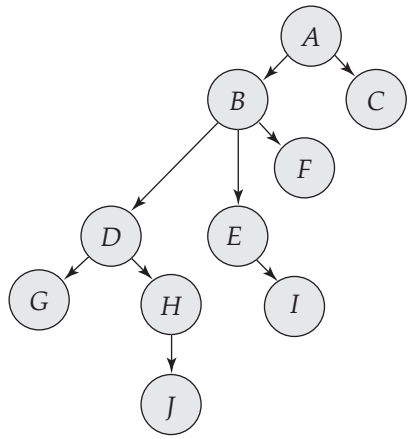


Figura 16.11 Grafo de la base de datos con estructura de árbol.

El orden parcial implica que el conjunto **D** se pueda ver como un grafo dirigido acíclico denominado **grafo de la base de datos**. Para simplificar, este apartado centra su atención sólo en aquellos grafos que son árboles con raíz. Se va a mostrar un protocolo simple llamado *protocolo de árbol*, el cual está restringido a utilizar sólo bloqueos *exclusivos*. En las notas bibliográficas se proporcionan referencias a otros protocolos basados en grafos más complejos.

En el **protocolo de bloqueo de árbol** sólo se permite la instrucción de bloqueo bloquear-C. Cada transacción T_i puede bloquear un elemento de datos al menos una vez y debe seguir las siguientes reglas:

1. El primer bloqueo de T_i puede ser sobre cualquier elemento de datos.
2. Posteriormente, T_i puede bloquear un elemento de datos Q sólo si T_i está bloqueando actualmente al padre de Q.
3. Los elementos de datos bloqueados se pueden desbloquear en cualquier momento.
4. T_i no puede bloquear de nuevo un elemento de datos que ya haya bloqueado y desbloqueado anteriormente.

Todas las planificaciones que sean legales bajo el protocolo de árbol son secuenciables en cuanto a conflictos.

Para ilustrar este protocolo considérese el grafo de la base de datos de la Figura 16.11. Las cuatro transacciones siguientes siguen el protocolo de árbol sobre dicho grafo. Sólo se muestran las instrucciones de bloqueo y desbloqueo:

T_{10}: bloquear-X(B); bloquear-X(E); bloquear-X(D); desbloquear(B); desbloquear(E); bloquear-X(G); desbloquear(D); desbloquear(G).
T_{11}: bloquear-X(D); bloquear-X(H); desbloquear(D); desbloquear(H).
T_{12}: bloquear-X(B); bloquear-X(E); desbloquear(E); desbloquear(B).
T_{13}: bloquear-X(D); bloquear-X(H); desbloquear(D); desbloquear(H).

En la Figura 16.12 se describe una planificación posible en la que participan estas cuatro transacciones. Obsérvese que durante su ejecución, la transacción T_{10} realiza bloqueos sobre dos subárboles *disjuntos*.

Obsérvese que la planificación de la Figura 16.12 es secuenciable en cuanto a conflictos. Se puede demostrar que el protocolo de árbol no sólo asegura la secuencialidad en cuanto a conflictos, sino que también asegura la ausencia de interbloqueos.

El protocolo de árbol de la Figura 16.12 no asegura la recuperabilidad y la ausencia de cascadas. Para asegurar la recuperabilidad y la ausencia de cascadas, el protocolo se puede modificar para que no permita liberar bloqueos exclusivos hasta el final de la transacción. Mantener los bloqueos exclusivos hasta el final de la transacción reduce la concurrencia. Existe una alternativa que mejora la concurrencia, pe-

T_{10}	T_{11}	T_{12}	T_{13}
bloquear-X(B)			
	bloquear-X(D)		
	bloquear-X(H)		
	desbloquear(D)		
bloquear-X(E)			
bloquear-X(D)			
desbloquear(B)			
desbloquear(E)			
		bloquear-X(B)	
		bloquear-X(E)	
	desbloquear(H)		
bloquear-X(G)			
desbloquear(D)			
			bloquear-X(D)
			bloquear-X(H)
			desbloquear(D)
			desbloquear(H)
		desbloquear(E)	
		desbloquear(B)	
desbloquear(G)			

Figura 16.12 Planificación secuenciable bajo el protocolo de árbol.

ro sólo asegura la recuperabilidad: para cada elemento de datos con una escritura no comprometida se registra qué transacción realizó la última escritura sobre el elemento de datos. Cada vez que una transacción T_i realiza una lectura de un elemento de datos no comprometido, se registra una **dependencia de compromiso** de T_i en la transacción que realizó la última escritura sobre el elemento de datos. Entonces, no se permite que la transacción T_i se comprometa hasta que todas las transacciones sobre las que tiene una dependencia de compromiso se comprometan. Si alguna de estas transacciones se interrumpe, también hay que interrumpir T_i.

El protocolo de bloqueo de árbol tiene la ventaja sobre el protocolo de bloqueo de dos fases en que, a diferencia del bloqueo de dos fases, está libre de interbloqueos, así que no se necesitan retrocesos. El protocolo de bloqueo de árbol tiene otra ventaja sobre el protocolo de bloqueo de dos fases en que los desbloqueos se pueden producir antes. El hecho de desbloquear antes puede llevar a unos tiempos de espera menores y a un aumento de la concurrencia.

Sin embargo, el protocolo tiene el inconveniente de que, en algunos casos, una transacción puede que tenga que bloquear elementos de datos a los que no accede. Por ejemplo, una transacción que tenga que acceder a los elementos de datos *A* y *J* en el grafo de la base de datos de la Figura 16.11, debe bloquear no sólo *A* y *J* sino también los elementos de datos *B*, *D* y *H*. Estos bloqueos adicionales producen un aumento del coste de los bloqueos, la posibilidad de tiempos de espera adicionales y un descenso potencial de la concurrencia. Además, sin un conocimiento previo de los elementos de datos que es necesario bloquear, las transacciones tienen que bloquear la raíz del árbol y esto puede reducir considerablemente la concurrencia.

Para un conjunto de transacciones pueden existir planificaciones secuenciables en cuanto a conflictos que no se pueden obtener por medio del protocolo de árbol. De hecho, hay planificaciones que son posibles por medio del protocolo de bloqueo de dos fases que no son posibles por medio del protocolo de árbol, y viceversa. En los ejercicios se exponen ejemplos de tales planificaciones.

16.2 Protocolos basados en marcas temporales

En los protocolos de bloqueo que se han descrito antes se determina el orden entre dos transacciones conflictivas en tiempo de ejecución a través del primer bloqueo que soliciten ambas que traiga consigo modos incompatibles. Otro método para determinar el orden de secuencialidad es seleccionar previa-

mente un orden entre las transacciones. El método más común para hacer esto es utilizar un esquema de *ordenación por marcas temporales*.

16.2.1 Marcas temporales

A toda transacción T_i del sistema se le asocia una única marca temporal fijada, denotada por MT(T_i). El sistema de base de datos asigna esta marca temporal antes de que comience la ejecución de T_i. Si a la transacción T_i se le ha asignado la marca temporal MT(T_i) y una nueva transacción T_j entra en el sistema, entonces MT(T_i) < MT(T_j). Existen dos métodos simples para implementar este esquema:

1. Usar el valor del **reloj del sistema** como marca temporal; es decir, la marca temporal de una transacción es igual al valor del reloj en el momento en el que la transacción entra en el sistema.
2. Usar un **contador lógico** que se incrementa cada vez que se asigna una nueva marca temporal; es decir, la marca temporal de una transacción es igual al valor del contador en el momento en el cual la transacción entra en el sistema.

Las marcas temporales de las transacciones determinan el orden de secuencia. De este modo, si MT(T_i) < MT(T_j) entonces el sistema debe asegurar que toda planificación que produzca sea equivalente a una planificación secuencial en la cual la transacción T_i aparezca antes que la transacción T_j.

Para implementar este esquema se asocia a cada elemento de datos Q dos valores de marca temporal:

- **marca-temporal-E**(Q) denota la mayor marca temporal de todas las transacciones que ejecutan con éxito escribir(Q).
- **marca-temporal-L**(Q) denota la mayor marca temporal de todas las transacciones que ejecutan con éxito leer(Q).

Estas marcas temporales se actualizan cada vez que se ejecuta una nueva operación leer(Q) o escribir(Q).

16.2.2 Protocolo de ordenación por marcas temporales

El **protocolo de ordenación por marcas temporales** asegura que todas las operaciones leer y escribir conflictivas se ejecutan en el orden de las marcas temporales. Este protocolo opera como sigue:

1. Supóngase que la transacción T_i ejecuta leer(Q).
   a. Si MT(T_i) <marca-temporal-E(Q) entonces T_i necesita leer un valor de Q que ya se ha sobrescrito. Por tanto se rechaza la operación leer y T_i retrocede.
   b. Si MT(T_i) $\geq$ marca-temporal-E(Q) entonces se ejecuta la operación leer y marca-temporal-L(Q) se asigna al máximo de marca-temporal-L(Q) y de MT(T_i).
2. Supóngase que la transacción T_i ejecuta escribir(Q).
   a. Si MT(T_i) < marca-temporal-L(Q) entonces el valor de Q que produce T_i se necesita previamente y el sistema asume que dicho valor no se puede producir nunca. Por tanto, se rechaza la operación escribir y T_i retrocede.
   b. Si MT(T_i) < marca-temporal-E(Q) entonces T_i está intentando escribir un valor de Q obsoleto. Por tanto, se rechaza la operación escribir y T_i retrocede.
   c. En otro caso se ejecuta la operación escribir y MT(T_i) se asigna a marca-temporal-E(Q).

A una transacción T_i que el esquema de control de concurrencia haya retrocedido como resultado de la ejecución de una operación leer o escribir se le asigna una nueva marca temporal y se inicia de nuevo.

Para ilustrar este protocolo considérense las transacciones T_{14} y T_{15}. La transacción T_{14} visualiza el contenido de las cuentas A y B:

T_{14}: leer(B);
leer(A);
visualizar(A + B).

T_{14}	T_{15}
leer(B)	
	leer(B)
	B:= B - 50
	escribir(B)
leer(A)	
	leer(A)
visualizar(A + B)	
	A:= A + 50
	escribir(A)
	visualizar(A + B)

Figura 16.13 Planificación 3.

La transacción T_{15} transfiere 50 € de la cuenta B a la A y muestra después el contenido de ambas:

T_{15}: leer(B);
$B := B - 50$;
escribir(B);
leer(A);
$A := A + 50$;
escribir(A);
visualizar(A + B).

En las planificaciones actuales con el protocolo de marcas temporales se asume que a una transacción se le asigna una marca temporal inmediatamente antes de su primera instrucción. Así en la planificación 3 de la Figura 16.13, MT(T_{14}) $<$ MT(T_{15}) y la planificación con el protocolo de marcas temporales es posible.

Obsérvese que la ejecución anterior puede obtenerse también con el protocolo de bloqueo de dos fases. Sin embargo, existen planificaciones que son posibles con el protocolo de bloqueo de dos fases que no lo son con el protocolo de marcas temporales y viceversa (véase el Ejercicio 16.25).

El protocolo de ordenación por marcas temporales asegura la secuencialidad en cuanto a conflictos. Esta afirmación se deduce del hecho de que las operaciones conflictivas se procesan durante la ordenación de las marcas temporales.

El protocolo asegura la ausencia de interbloqueos, ya que ninguna transacción tiene que esperar. Sin embargo, existe una posibilidad de inanición de las transacciones largas si una secuencia de transacciones cortas conflictivas provoca reinicios repetidos de la transacción larga. Si se descubre que una transacción se está reiniciando de forma repetida, es necesario bloquear las transacciones conflictivas de forma temporal para permitir que la transacción termine.

El protocolo puede generar planificaciones no recuperables. Sin embargo, se puede extender para producir planificaciones recuperables de una de las siguientes formas:

- La recuperabilidad y la ausencia de cascadas se pueden asegurar realizando todas las escrituras juntas al final de la transacción. Las escrituras tienen que ser atómicas en el siguiente sentido: mientras que las escrituras están en progreso, no se permite que ninguna transacción acceda a ninguno de los elementos de datos que se han escrito.
- La recuperabilidad y la ausencia de cascadas también se pueden garantizar utilizando una forma limitada de bloqueo, por medio de la cual las lecturas de los elementos no comprometidos se posponen hasta que la transacción haya actualizado el elemento comprometido (véase el Ejercicio 16.26).
- La recuperabilidad sola se puede asegurar realizando un seguimiento de las escrituras no comprometidas y sólo permitiendo que una transacción T_i se comprometa después de que se comprometa cualquier transacción que escribió un valor que leyó T_i . Las dependencias de compromiso, esbozadas en el Apartado 16.1.5, se pueden utilizar para este propósito.

T_{16}	T_{17}
leer(Q)	
	escribir(Q)
escribir(Q)	

Figura 16.14 Planificación 4.

16.2.3 Regla de escritura de Thomas

Ahora se presenta una modificación del protocolo de ordenación por marcas temporales que permite una mayor concurrencia potencial que la que tiene el protocolo del Apartado 16.2.2. Considérese la planificación 4 de la Figura 16.14 y aplíquese el protocolo de ordenación por marcas temporales. Puesto que T_{16} comienza antes que T_{17} se asume que MT(T_{16}) < MT(T_{17}). La operación leer(Q) de T_{16} tiene éxito, así como la operación escribir(Q) de T_{17}. Cuando T_{16} intenta hacer su operación escribir(Q) se encuentra con que MT(T_{16}) < marca-temporal-E(Q), ya que marca-temporal-E(Q) = MT(T_{17}). De este modo se rechaza la operación escribir(Q) de T_{16} y se debe retroceder la transacción.

A pesar de que el protocolo de ordenación por marcas temporales exige el retroceso de la transacción T_{16}, esto no es necesario. Como T_{17} ya ha escrito Q, el valor que intenta escribir T_{16} nunca se va leer. Toda transacción T_i con MT(T_i) < MT(T_{17}) que intente una operación leer(Q) retrocederá, ya que MT(T_i) < marca-temporal-E(Q). Toda transacción T_j con MT(T_j) >MT(T_{17}) debe leer el valor de Q que ha escrito T_{17} en lugar del valor que ha escrito T_{16}.

Esta observación lleva a la versión modificada del protocolo de ordenación por marcas temporales en el cual se pueden ignorar las operaciones escribir obsoletas bajo ciertas circunstancias. Las reglas del protocolo para las operaciones leer no sufren cambios. Sin embargo, las reglas del protocolo para las operaciones escribir son algo diferentes de las del protocolo de ordenación por marcas temporales del Apartado 16.2.2.

La modificación al protocolo de ordenación por marcas temporales, denominada **regla de escritura de Thomas**, es la siguiente. Supóngase que la transacción T_i ejecuta escribir(Q).

1. Si MT(T_i) < marca-temporal-L(Q) entonces el valor de Q que produce T_i se necesita previamente y el sistema asume que dicho valor no se puede producir nunca. Por tanto, se rechaza la operación escribir y T_i retrocede.
2. Si MT(T_i) < marca-temporal-E(Q) entonces T_i está intentando escribir un valor de Q obsoleto. Por tanto se puede ignorar dicha operación escribir.
3. En otro caso se ejecuta la operación escribir y MT(T_i) se asigna a marca-temporal-E(Q).

La diferencia entre las reglas anteriores y las del Apartado 16.2.2 está en la segunda regla. El protocolo de ordenación por marcas temporales exige que T_i retroceda si ejecuta escribir(Q) y MT(T_i) < marca-temporal-E(Q). Sin embargo, ahora se ignora la instrucción escribir obsoleta en aquellos casos en los que MT(T_i) ≥ marca-temporal-L(Q).

La regla de escritura de Thomas emplea la secuencialidad en cuanto a vistas borrando las operaciones escribir obsoletas de las transacciones que las ejecutan. Esta modificación de las transacciones hace posible generar planificaciones que no serían posibles con otros protocolos que se han presentado en este capítulo. Por ejemplo, la planificación 4 de la Figura 16.14 no es secuenciable en cuanto a conflictos y, por tanto, no es posible con el protocolo de bloqueo de dos fases, con el protocolo de árbol o con el protocolo de ordenación por marcas temporales. Con la regla de escritura de Thomas se ignoraría la operación escribir(Q) de T_{16}. El resultado es una planificación que es equivalente en cuanto a vistas a la planificación secuencial < T_{16}, T_{17}>.

16.3 Protocolos basados en validación

En aquellos casos en que la mayoría de las transacciones son de sólo lectura, la tasa de conflictos entre las transacciones puede ser baja. Así, muchas de esas transacciones, si se ejecutasen sin la supervisión de un esquema de control de concurrencia, llevarían no obstante al sistema a un estado consistente.

Un esquema de control de concurrencia impone una sobrecarga en la ejecución de código y un posible retardo en las transacciones. Sería aconsejable utilizar otro esquema alternativo que impusiera menos sobrecarga. La dificultad de reducir la sobrecarga está en que no se conocen de antemano las transacciones que estarán involucradas en un conflicto. Para obtener dicho conocimiento se necesita un esquema para **supervisar** el sistema.

Se asume que cada transacción T_i se ejecuta en dos o tres fases diferentes durante su tiempo de vida dependiendo de si es una transacción de sólo lectura o una de actualización. Las fases son, en orden,

1. **Fase de lectura**. Durante esta fase tiene lugar la ejecución de la transacción T_i. Se leen los valores de varios elementos de datos y se almacenan en variables locales de T_i. Todas las operaciones escribir se realizan sobre las variables locales temporales sin actualizar la base de datos actual.
2. **Fase de validación**. La transacción T_i realiza una prueba de validación para determinar si puede copiar a la base de datos las variables locales temporales que tienen los resultados de las operaciones escribir sin causar una violación de la secuencialidad.
3. **Fase de escritura**. Si la transacción T_i tiene éxito en la validación (paso 2) entonces las actualizaciones reales se aplican a la base de datos. En otro caso, T_i retrocede.

Cada transacción debe pasar por las tres fases y en el orden que se muestra. Sin embargo, se pueden entrelazar las tres fases de la ejecución concurrente de las transacciones.

Para realizar la prueba de validación se necesita conocer el momento en que tienen lugar las distintas fases de las transacciones T_i. Se asociarán por tanto tres marcas temporales distintas a la transacción T_i:

1. **Inicio**(T_i). Momento en el cual T_i comienza su ejecución.
2. **Validación**(T_i). Momento en el cual T_i termina su fase de lectura y comienza su fase de validación.
3. **Fin**(T_i). Momento en el cual T_i termina su fase de escritura.

Se determina el orden de secuencialidad a través de la técnica de la ordenación por marcas temporales utilizando el valor de la marca temporal Validación(T_i). De este modo, el valor MT(T_i) = Validación(T_i) y, si MT(T_j) $<$ MT(T_k), entonces toda planificación que se produzca debe ser equivalente a una planificación secuencial en la cual la transacción T_j aparezca antes que la transacción T_k. La razón por la que se ha elegido Validación(T_i) como marca temporal de la transacción T_i, en lugar de Inicio(T_i), es porque resulta posible esperar un tiempo de respuesta más rápido dado que la tasa de conflictos entre transacciones es realmente bajo.

La **comprobación de validación** para la transacción T_j exige que, para toda transacción T_i con MT(T_i) $<$ MT(T_j), se cumplan una de las dos condiciones siguientes:

1. Fin(T_i) $<$ Inicio(T_j). Puesto que T_i completa su ejecución antes de que comience T_j, el orden de secuencialidad se mantiene realmente.
2. El conjunto de todos los elementos de datos que escribe T_i tiene intersección vacía con el conjunto de elementos de datos que lee T_j, y T_i completa su fase de escritura antes de que T_j comience su fase de validación (Inicio(T_j) $<$ Fin(T_i) $<$ Validación(T_j)). Esta condición asegura que las escrituras de T_i y T_j no se superpongan. Puesto que las escrituras de T_i no afectan a la lectura de T_j, y puesto que T_j no puede afectar a la lectura de T_i, el orden de secuencialidad se mantiene realmente.

Como ejemplo considérense de nuevo las transacciones T_{14} y T_{15}. Supóngase que MT(T_{14}) $<$ MT(T_{15}). Entonces la fase de validación tiene éxito y produce la planificación 5, la cual se describe en la Figura 16.15. Obsérvese que las escrituras a las variables actuales se realizan sólo después de la fase de validación de T_{15}. Así, T_{14} lee los valores anteriores de B y A y la planificación es secuenciable.

El esquema de validación evita automáticamente los retrocesos en cascada, ya que las escrituras reales tienen lugar sólo después de que la transacción que ejecuta la escritura se haya comprometido. Sin embargo, existe una posibilidad de inanición de las transacciones largas debido a una secuencia de

T_{14}	T_{15}
leer(B)	
	leer(B)
	$B := B - 50$
	leer(A)
	$A := A + 50$
leer(A)	
$\langle$ *validar* $\rangle$	
visualizar($A + B$)	
	$\langle$ *validar* $\rangle$
	escribir(B)
	escribir(A)

Figura 16.15 Planificación 5 producida usando validación.

transacciones cortas conflictivas que provoca reinicios repetidos de la transacción larga. Para evitar la inanición es necesario bloquear las transacciones conflictivas de forma temporal para permitir que la transacción larga termine.

Este esquema de validación se denomina esquema **control de concurrencia optimista** dado que las transacciones se ejecutan de forma optimista, asumiendo que serán capaces de finalizar la ejecución y validar al final. En cambio, los bloqueos y la ordenación por marcas temporales son pesimistas en cuanto que fuerzan una espera o un retroceso cada vez que se detecta un conflicto, aun cuando existe una posibilidad de que la planificación sea secuenciable en cuanto a conflictos.

16.4 Granularidad múltiple

En los esquemas de control de concurrencia que se han descrito antes se ha tomado cada elemento de datos individual como la unidad sobre la cual se producía la sincronización.

Sin embargo, hay circunstancias en las que puede ser conveniente agrupar varios elementos de datos y tratarlos como una unidad individual de sincronización. Por ejemplo, si la transacción T_i tiene que acceder a toda la base de datos y se usa un protocolo de bloqueo, entonces T_i debe bloquear cada elemento de la base de datos. Ejecutar estos bloqueos produce claramente un consumo de tiempo. Sería mejor que T_i pudiera realizar una *única* petición de bloqueo para bloquear toda la base de datos. Por otro lado, si la transacción T_j necesita acceder sólo a unos cuantos datos no sería necesario bloquear toda la base de datos, ya que en ese caso se perdería la concurrencia.

Lo que se necesita es un mecanismo que permita al sistema definir múltiples niveles de **granularidad**. Se puede hacer uno permitiendo que los elementos de datos sean de varios tamaños y definiendo una jerarquía de granularidades de los datos, en la cual las granularidades pequeñas están anidadas en otras más grandes. Una jerarquía tal se puede representar gráficamente como un árbol. Obsérvese que el árbol que se usa aquí es significativamente distinto del que se usaba en el protocolo de árbol (Apartado 16.1.5). Un nodo interno del árbol de granularidad múltiple representa los datos que se asocian con sus descendientes. En el protocolo de árbol cada nodo es un elemento de datos independiente.

Como ejemplo considérese el árbol de la Figura 16.16, el cual consiste en cuatro niveles de nodos. El nivel superior representa toda la base de datos. Después de éste están los nodos de tipo *zona*; la base de datos consiste en estas zonas exactamente. Cada zona tiene nodos de tipo *archivo* como hijos. Cada zona contiene exactamente aquellos archivos que sean sus nodos hijos. Ningún archivo está en más de una zona. Finalmente, cada archivo tiene nodos de tipo *registro*. Como antes, un archivo consiste exactamente en aquellos registros que sean sus nodos hijos y ningún registro puede estar presente en más de un archivo.

Se puede bloquear individualmente cada nodo del árbol. Como ya se hizo en el protocolo de bloqueo de dos fases, se van a usar los modos de bloqueo **compartido** y **exclusivo**. Cuando una transacción bloquea un nodo, tanto en modo compartido como exclusivo, también bloquea todos los descendientes de ese nodo con el mismo modo de bloqueo. Por ejemplo, si la transacción T_i **bloquea explícitamente** el archivo A_c de la Figura 16.16 en modo exclusivo, entonces **bloquea implícitamente** en modo exclusivo

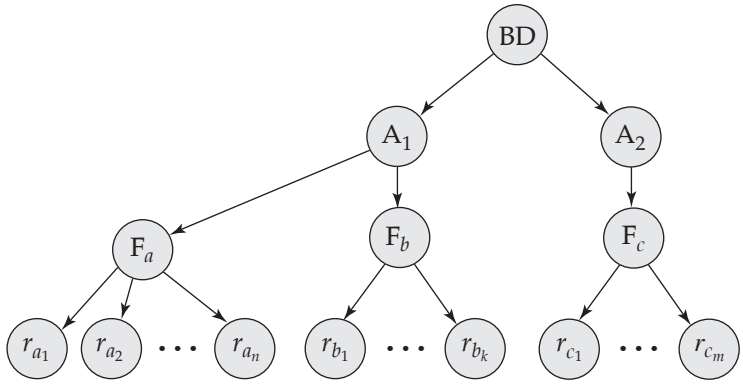


Figura 16.16 Jerarquía de granularidad.

todos los registros que pertenecen a dicho archivo. No es necesario que bloquee explícitamente cada registro individual de F_c.

Supóngase que la transacción T_j quiere bloquear el registro r_{b_6} del archivo A_b. Puesto que T_i ha bloqueado A_b explícitamente, se entiende que también se bloquea r_{b_6} (implícitamente). Pero cuando T_j realiza una petición de bloqueo sobre r_{b_6}, ¡ r_{b_6} no está bloqueado explícitamente! ¿Cómo puede determinar el sistema si T_j puede bloquear r_{b_6} ? T_j debe recorrer el árbol desde la raíz hasta el nodo r_{b_6}. Si cualquier nodo del camino está bloqueado en un modo incompatible entonces hay que retrasar T_j.

Supóngase ahora que la transacción T_k quiere bloquear toda la base de datos. Para hacer esto debe bloquear simplemente la raíz de la jerarquía. Obsérvese, sin embargo, que T_k no tendrá éxito al bloquear el nodo raíz, ya que T_i posee un bloqueo sobre parte del árbol (en concreto sobre el archivo A_b). ¿Pero cómo determina el sistema si se puede bloquear el nodo raíz? Una posibilidad es buscar en todo el árbol. Sin embargo, esta solución anula el propósito del esquema de bloqueo de granularidad múltiple. Un modo más eficiente de obtener este conocimiento es introducir una nueva clase de modo de bloqueo que se denomina **modo de bloqueo intencional**. Si un nodo se bloquea en modo intencional se está haciendo un bloqueo explícito en un nivel inferior del árbol (es decir, en una granularidad más fina). Los bloqueos intencionales se colocan en todos los ascendientes de un nodo antes de bloquearlo explícitamente. Así, no es necesario que una transacción busque en todo el árbol para determinar si puede bloquear un nodo con éxito. Una transacción que quiera bloquear un nodo—por ejemplo Q—debe recorrer el camino en el árbol desde la raíz hasta Q. Durante el recorrido del árbol la transacción bloquea los distintos nodos en un modo intencional.

Hay un modo intencional asociado al modo compartido y hay otro asociado al modo exclusivo. Si un nodo se bloquea en **modo intencional-compartido** (IC) se realiza un bloqueo explícito en un nivel inferior del árbol, pero sólo con bloqueos en modo compartido. De forma similar, si se bloquea un nodo en **modo intencional-exclusivo** (IX) entonces el bloqueo explícito se hace en un nivel inferior con bloqueos en modo exclusivo o en modo compartido. Finalmente, si se bloquea un nodo en **modo intencional-exclusivo y compartido** (IXC) el subárbol cuya raíz es ese nodo se bloquea explícitamente en modo exclusivo, y dicho bloqueo explícito se produce en un nivel inferior con bloqueos en modo exclusivo. La función de compatibilidad para estos modos de bloqueo se presenta en la Figura 16.17.

	IC	IX	C	IXC	X
IC	cierto	cierto	cierto	cierto	falso
IX	cierto	cierto	falso	falso	falso
C	cierto	falso	cierto	falso	falso
IXC	cierto	falso	falso	falso	falso
X	falso	falso	falso	falso	falso

Figura 16.17 Matriz de compatibilidad.

El **protocolo de bloqueo de granularidad múltiple** siguiente asegura la secuencialidad. Cada transacción T_i puede bloquear un nodo Q usando las reglas siguientes:

1. Debe observar la función de compatibilidad de bloqueos de la Figura 16.17.
2. Debe bloquear la raíz del árbol en primer lugar y puede bloquearla en cualquier modo.
3. Puede bloquear un nodo Q en modo C o IC sólo si está bloqueando actualmente al padre de Q en modo IX o IC.
4. Puede bloquear un nodo Q en modo X, IXC o IX sólo si está bloqueando actualmente al padre de Q en modo IX o IXC.
5. Puede bloquear un nodo sólo si no ha desbloqueado previamente ningún nodo (es decir, T_i es de dos fases).
6. Puede desbloquear un nodo Q sólo si no ha bloqueado a ninguno de los hijos de Q.

Obsérvese que en el protocolo de granularidad múltiple es necesario que se adquieran los bloqueos en *orden descendente* (de la raíz a las hojas), y que se liberen en *orden ascendente* (de las hojas a la raíz).

Para ilustrar este protocolo considérese el árbol de la Figura 16.16 y las transacciones siguientes:

- Supóngase que la transacción T_{18} lee el registro r_{a_2} del archivo A_a. Entonces T_{18} necesita bloquear la base de datos, la zona Z_1 y A_a en modo IC (y en este orden) y finalmente debe bloquear r_{a_2} en modo C.
- Supóngase que la transacción T_{19} modifica el registro r_{a_9} del archivo A_a. Entonces T_{19} necesita bloquear la base de datos, la zona Z_1 y el archivo A_a en modo IX y finalmente debe bloquear r_{a_9} en modo X.
- Supóngase que la transacción T_{20} lee todos los registros del archivo A_a. Entonces, T_{20} necesita bloquear la base de datos y la zona Z_1 (en este orden) en modo IC y finalmente debe bloquear A_a en modo C.
- Supóngase que la transacción T_{21} lee toda la base de datos. Puede hacer esto una vez que bloquee la base de datos en modo C.

Se observa que las transacciones T_{18}, T_{20} y T_{21} pueden acceder concurrentemente a la base de datos. La transacción T_{19} se puede ejecutar concurrentemente con T_{18}, pero no con T_{20} ni con T_{21}.

Este protocolo permite la concurrencia y reduce la sobrecarga de bloqueos. Es particularmente útil en las aplicaciones con una mezcla de:

- Transacciones cortas que sólo acceden a algunos elementos de datos.
- Transacciones largas que producen informes a partir de un archivo completo o un conjunto de archivos.

Existe un protocolo de bloqueo similar que es aplicable a sistemas de bases de datos en los cuales las granularidades de los datos se organizan en forma de grafo dirigido acíclico. Véanse las notas bibliográficas para obtener referencias adicionales. En este protocolo son posibles los interbloqueos, al igual que en los protocolos de bloqueo de dos fases. Existen técnicas para reducir la frecuencia de interbloqueos en el protocolo de granularidad múltiple y también para eliminar los interbloqueos completamente. Se hace referencia a estas técnicas en las notas bibliográficas.

16.5 Esquemas multiversión

Los esquemas de control de concurrencia que se han descrito anteriormente aseguran la secuencialidad o bien retrasando una operación o bien abortando la transacción que realiza la operación. Por ejemplo, una operación leer se puede retrasar porque todavía no se haya escrito el valor apropiado; o se puede rechazar (es decir, la transacción que la realiza debe abortarse) porque se haya sobrescrito el valor que

supuestamente se iba a leer. Se podrían evitar estos problemas si se mantuvieran en el sistema copias anteriores de cada elemento de datos.

En los esquemas de **control de concurrencia multiversión**, cada operación escribir(Q) crea una nueva **versión** de Q. Cuando se realiza una operación leer(Q) el gestor de control de concurrencia elige una de las versiones de Q que se va a leer. El esquema de control de concurrencia debe asegurar que la elección de la versión que se va a leer se haga de tal manera que asegure la secuencialidad. Así mismo es crucial, por motivos de rendimiento, que una transacción sea capaz de determinar rápida y fácilmente la versión del elemento de datos que se va a leer.

16.5.1 Ordenación por marcas temporales multiversión

La técnica más frecuentemente utilizada en los esquemas multiversión es la de las marcas temporales. A cada transacción T_i del sistema se le asocia una única marca temporal estática que se denota MT(T_i). El sistema de bases de dates asigna dicha marca temporal antes de que la transacción comience su ejecución, como se ha descrito en el Apartado 16.2.

A cada elemento de datos Q se el asocia una secuencia de versiones $<Q_1, Q_2, \ldots, Q_m>$. Cada versión Q_k contiene tres campos:

- **contenido** es el valor de la versión Q_k.
- **marca-temporal-E**(Q_k) es la marca temporal de la transacción que haya creado la versión Q_k.
- **marca-temporal-L**(Q_k) es la mayor marca temporal de todas las transacciones que hayan leído con éxito la versión Q_k.

Una transacción—por ejemplo, T_i—crea una nueva versión Q_k del elemento de datos Q realizando la operación escribir(Q). El campo contenido de la versión tiene el valor que ha escrito T_i. El sistema inicializa la marca-temporal-E y marca-temporal-L con MT(T_i). El valor marca-temporal-L se actualiza cada vez que una transacción T_j lee el contenido de Q_k y marca-temporal-L(Q_k) $<$ MT(T_j).

El **esquema de marcas temporales multiversión** que se muestra a continuación asegura la secuencialidad. El esquema opera como sigue. Supóngase que la transacción T_i realiza una operación leer(Q) o escribir(Q). Sea Q_k la versión de Q cuya marca temporal de escritura es la mayor marca temporal menor o igual que MT(T_i).

1. Si la transacción T_i ejecuta leer(Q) entonces el valor que se devuelve es el contenido de la versión Q_k.
2. Si la transacción T_i ejecuta escribir(Q) y si MT(T_i) $<$ marca-temporal-L(Q_k) entonces la transacción T_i retrocede. Si no, si MT(T_i) $=$ marca-temporal-E(Q_k) se sobrescribe el contenido de Q_k, y en otro caso se crea una nueva versión de Q.

La justificación de la regla 1 es clara. Una transacción lee la versión más reciente que viene antes de ella en el tiempo. La segunda regla fuerza a que se aborte una transacción que realice una escritura "demasiado tarde". Con más precisión, si T_i intenta escribir una versión que alguna otra transacción haya leído, entonces no se puede permitir que dicha escritura tenga éxito.

La versiones que ya no se necesitan se borran basándose en la regla siguiente. Supóngase que hay dos versiones, Q_k y Q_j, de un elemento de datos y que ambas tienen marca-temporal-E menor que la marca temporal de la transacción más antigua en el sistema. Entonces no se volverá a usar la versión más antigua de Q_k y Q_j y se puede borrar.

El esquema de ordenación por marcas temporales multiversión tiene la propiedad deseable de que nunca falla una petición de lectura y nunca tiene que esperar. En sistemas de bases de datos típicos, en los cuales la lectura es una operación mucho más frecuente que la escritura, esta ventaja puede tener un mayor significado práctico.

Este esquema, sin embargo, tiene dos propiedades no deseables. En primer lugar, la lectura de un elemento de datos necesita también la actualización del campo marca-temporal-L, lo que tiene como resultado dos accesos potenciales al disco en lugar de uno. En segundo lugar, los conflictos entre las

transacciones se resuelven por medio de retrocesos en lugar de esperas. Esta alternativa puede ser costosa. En el Apartado 16.5.2 se describe un algoritmo que evita este problema.

Este esquema de ordenación por marcas temporales multiversión no asegura la recuperabilidad y la ausencia de cascadas. Se puede extender de la misma forma que el esquema de ordenación por marcas temporales básico, para hacerlo recuperable y sin cascadas.

16.5.2 Bloqueo de dos fases multiversión

El **protocolo de bloqueo de dos fases multiversión** intenta combinar las ventajas del control de concurrencia multiversión con las ventajas del bloqueo de dos fases. Este protocolo distingue entre **transacciones de sólo lectura** y **transacciones de actualización**.

Las transacciones de actualización realizan un bloqueo de dos fases riguroso; es decir, mantienen todos los bloqueos hasta el final de la transacción. Así, se pueden secuenciar según su orden de terminación. Cada elemento de datos tiene una única marca temporal. La marca temporal no es en este caso una marca temporal basada en un reloj real, sino que es un contador, que se denomina contador-mt, que se incrementa durante el procesamiento del compromiso.

A las transacciones de sólo lectura se les asigna una marca temporal leyendo el valor actual de contador-mt antes de que comiencen su ejecución; para realizar las lecturas siguen el protocolo de ordenación por marcas temporales multiversión. Así, cuando una transacción de sólo lectura T_i ejecuta leer(Q), el valor que se devuelve es el contenido de la versión con la mayor marca temporal menor o igual que MT(T_i).

Cuando una transacción de actualización lee un elemento obtiene un bloqueo compartido sobre él y lee la versión más reciente de dicho elemento de datos. Cuando una transacción de actualización quiere escribir un elemento, obtiene primero un bloqueo exclusivo sobre el elemento, y luego crea una nueva versión del elemento de datos. La escritura se realiza sobre la nueva versión y la marca temporal de la nueva versión tiene inicialmente el valor ∞, el cual es mayor que el de cualquier marca temporal posible.

Una vez que la transacción T_i ha completado sus acciones realiza el procesamiento del compromiso como sigue: en primer lugar, T_i asigna la marca temporal de todas las versiones que ha creado al valor de contador-mt más 1; después T_i incrementa contador-mt en 1. Sólo se permite que realice el procesamiento del compromiso a una transacción de actualización a la vez.

Como resultado, las transacciones que comiencen después de que T_i incremente contador-mt observarán los valores que T_i ha actualizado, mientras que aquéllas que comiencen antes de que T_i incremente contador-mt observarán los valores anteriores a la actualización de T_i. En ambos casos las transacciones de sólo lectura no tienen que esperar nunca por los bloqueos. El bloqueo de dos fases multiversión también asegura que las planificaciones recuperables y sin cascadas.

Las versiones se borran de forma similar a la utilizada en la ordenación por marcas temporales multiversión. Supóngase que hay dos versiones, Q_k y Q_j, de un elemento de datos y que ambas versiones tienen una marca temporal menor o igual que la de la transacción de sólo lectura más antigua del sistema. Entonces la versión más antigua de Q_k y Q_j no se volverá a utilizar y se puede borrar.

El bloqueo de dos fases multiversión o variaciones de éste se usan en algunos sistemas de bases de datos comerciales.

16.6 Tratamiento de interbloqueos

Un sistema está en estado de interbloqueo si existe un conjunto de transacciones tal que toda transacción del conjunto está esperando a otra transacción del conjunto. Con mayor precisión, existe un conjunto de transacciones en espera $\{T_0, T_1, \ldots, T_n\}$ tal que T_0 está esperando a un elemento de datos que posee T_1, y T_2 está esperando a un elemento de datos que posee T_2, y ..., y T_{n-1} está esperando a un elemento de datos que posee T_n, y T_n está esperando a un elemento que posee T_0. En tal situación ninguna de las transacciones puede progresar.

El único remedio a esta situación no deseada es que el sistema invoque alguna acción drástica, como hacer retroceder alguna de las transacciones involucradas en el interbloqueo. El retroceso de una transac-

ción puede ser parcial: esto es, se puede retroceder una transacción hasta el punto donde obtuvo un bloqueo cuya liberación resuelve el interbloqueo.

Existen dos métodos principales para tratar el problema de los interbloqueos. Se puede utilizar un protocolo de **prevención de interbloqueos** para asegurar que el sistema *nunca* llega a un estado de interbloqueo. De forma alternativa se puede permitir que el sistema llegue a un estado de interbloqueo, y tratar de recuperarse después a través de un esquema de **detección** y **recuperación** de interbloqueos. Como se verá a continuación, ambos pueden provocar retroceso de las transacciones. La prevención se usa normalmente cuando la probabilidad de que el sistema llegue a un estado de interbloqueo es relativamente alta; en otro caso es más eficiente usar la detección y recuperación.

Obsérvese que el esquema de detección y recuperación necesita una sobrecarga que incluye no sólo el coste en tiempo de ejecución para mantener la información necesaria para ejecutar el algoritmo de detección, sino también las pérdidas potenciales inherentes a la recuperación de un interbloqueo.

16.6.1 Prevención de interbloqueos

Existen dos enfoques a la prevención de interbloqueos. Un enfoque asegura que no puede haber esperas cíclicas ordenando las peticiones de bloqueo o exigiendo que todos los bloqueos se adquieran juntos. El otro enfoque es más cercano a la recuperación de interbloqueos y realiza retrocesos de las transacciones en lugar de esperar un bloqueo, siempre que el bloqueo pueda llevar potencialmente a un interbloqueo.

El esquema más simple para la primera aproximación exige que cada transacción bloquee todos sus elementos de datos antes de comenzar su ejecución. Además, o bien se bloquean todos en un paso o ninguno de ellos se bloquea. Este protocolo tiene dos inconvenientes principales: (1) a menudo es difícil predecir, antes de que comience la transacción, cuáles son los elementos de datos que es necesario bloquear; (2) la utilización de elementos de datos puede ser muy baja, ya que muchos de los elementos de datos pueden estar bloqueados pero sin usar durante mucho tiempo.

Otro esquema para prevenir interbloqueos consiste en imponer un orden parcial a todos los elementos de datos y exigir que una transacción bloquee un elemento de datos sólo en el orden que especifica dicho orden parcial. Se ha visto un esquema así en el protocolo de árbol, que usa una ordenación parcial de los elementos de datos.

Una variante a esta aproximación es utilizar un orden total de los elementos de datos, en conjunción con el bloqueo de dos fases. Una vez que una transacción ha bloqueado un elemento en particular, no puede solicitar bloqueos sobre elementos que preceden a dicho elemento en la ordenación. Este esquema es fácil de implementar siempre que el conjunto de los elementos de datos a los que accede una transacción se conozca cuando la transacción comienza la ejecución. No hay necesidad de cambiar el sistema de control de concurrencia subyacente si se utiliza bloqueo de dos fases: todo lo que hace falta es asegurar que los bloqueos se solicitan en el orden adecuado.

La segunda aproximación para prevenir interbloqueos consiste en utilizar expropiación y retrocesos de transacciones. En la expropiación, cuando la transacción T_2 solicita un bloqueo que la transacción T_1 posee, el bloqueo concedido a T_1 debe **expropiarse** retrocediendo T_1 y concediéndolo a continuación a T_2. Para controlar la expropiación se asigna una marca temporal única a cada transacción. El sistema usa estas marcas temporales sólo para decidir si una transacción debe esperar o retroceder. Para el control de concurrencia se sigue usando el bloqueo. Si una transacción retrocede mantiene su marca temporal *antigua* cuando vuelva a comenzar. Se han propuesto dos esquemas de prevención de interbloqueos que usan marcas temporales:

1. El esquema **esperar–morir** está basado en una técnica sin expropiación. Cuando la transacción T_i solicita un elemento de datos que posee actualmente T_j, T_i puede esperar sólo si tiene una marca temporal más pequeña que la de T_j (es decir, T_i es anterior a T_j). En caso contrario T_i retrocede (muere).

   Por ejemplo, supóngase que las transacciones T_{22}, T_{23} y T_{24} tienen marcas temporales 5, 10 y 15 respectivamente. Si T_{22} solicita un elemento de datos que posee T_{23} entonces T_{22} tiene que esperar. Si T_{24} solicita un elemento de datos que posee T_{23} entonces T_{24} retrocede.

2. El esquema **herir–esperar** está basado en una técnica de expropiación. Es el opuesto del esquema esperar–morir. Cuando la transacción T_i solicita un elemento de datos que posee actualmente T_j,

T_i puede esperar sólo si tiene una marca temporal mayor que la de T_j (es decir, T_i es más reciente que T_j). En caso contrario T_j retrocede (T_i *hiere* a T_j).

Volviendo al ejemplo anterior con las transacciones T_{22}, T_{23} y T_{24}, si T_{22} solicita un elemento de datos que posee T_{23} entonces se expropia este elemento de datos y T_{23} retrocede. Si T_{24} solicita el elemento de datos que posee T_{23} entonces T_{24} debe esperar.

Siempre que retrocedan las transacciones, es importante asegurarse de que no exista **inanición**—es decir, ninguna transacción retrocede repetidamente y nunca se le permite progresar.

Tanto el esquema herir–esperar como el esperar–morir impiden la inanición: en todo momento hay una transacción con la menor marca temporal. Dicha transacción *no puede* retroceder en ninguno de los dos esquemas. Puesto que las marcas temporales siempre se incrementan, y puesto que *no* se asignan a las transacciones nuevas marcas temporales cuando retroceden, una transacción que retrocede tendrá finalmente la menor marca temporal. De este modo no retrocederá de nuevo.

Sin embargo, existen algunas diferencias en el modo en que operan estos dos esquemas.

- En el esquema esperar–morir una transacción más antigua debe esperar a que una más reciente libere sus elementos de datos. Así, cuanto más antigua se vuelve una transacción más tiende a esperar. Por el contrario, en el esquema herir–esperar, una transacción más antigua nunca espera a una más reciente.
- En el esquema esperar–morir, si la transacción T_i muere y retrocede a causa de que ha solicitado un elemento de datos que posee la transacción T_j, entonces T_i puede volver a realizar la misma secuencia de peticiones cuando vuelva a comenzar. Si T_j posee todavía el elemento de datos, entonces T_i morirá de nuevo. De este modo T_i morirá varias veces hasta que adquiera el elemento de datos que necesita. Compárese esta secuencia de sucesos con la que ocurre en el esquema herir–esperar. La transacción T_i está herida y retrocede porque T_j solicita un elemento de datos que posee. Cuando vuelve a comenzar T_i y solicita el elemento de datos que ahora posee T_j, T_i espera. De este modo puede haber menos retrocesos en el esquema herir–esperar.

El principal problema de cualquiera de estos dos esquemas es que se pueden dar retrocesos innecesarios.

16.6.2 Esquemas basados en límite de tiempo

Otro enfoque simple al tratamiento de interbloqueos se basa en **bloqueos con límite de tiempo**. En este enfoque una transacción que haya solicitado un bloqueo espera por lo menos durante un intervalo de tiempo especificado. Si no se concede el bloqueo en ese tiempo, se dice que la transacción está fuera de tiempo y ella misma retrocede y vuelve a comenzar. Si de hecho había un interbloqueo, una o varias de las transacciones involucradas en dicho interbloqueo estarán fuera de tiempo y retrocederán, lo que permitirá continuar a las demás. Este esquema se encuentra en un lugar entre la prevención de interbloqueos, donde nunca puede haber un interbloqueo, y la detección y recuperación, las cuales se describen en el Apartado 16.6.3.

El esquema de límite de tiempo es particularmente fácil de implementar y funciona bien si las transacciones son cortas y, si son largas, las esperas deben de ser a causa de los interbloqueos. Sin embargo, es difícil en general decidir cuánto debe esperar una transacción para que esté fuera de tiempo. Esperas demasiado largas provocan retardos innecesarios una vez que se ha producido un interbloqueo. Esperas demasiado cortas producen el retroceso de la transacción incluso si no hay interbloqueo, provocando un gasto de recursos. La inanición es también posible con este esquema. Por tanto el esquema basado en límite de tiempo tiene una aplicación limitada.

16.6.3 Detección y recuperación de interbloqueos

Si el sistema no utiliza algún protocolo que asegure la ausencia de interbloqueos, entonces se debe usar un esquema de detección y recuperación. Periódicamente se invoca a un algoritmo que examina el estado del sistema para determinar si hay un interbloqueo. Si lo hay, entonces el sistema debe intentar recuperarse del interbloqueo. Para ello, el sistema debe:

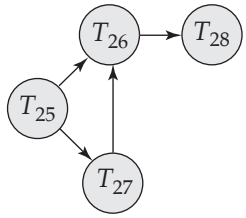


Figura 16.18 Grafo de espera sin ciclos.

- Mantener información sobre la asignación de los elementos de datos a las transacciones, así como de toda petición de elemento de datos pendiente.
- Proporcionar un algoritmo que use esta información para determinar si el sistema ha entrado en un estado de interbloqueo.
- Recuperarse del interbloqueo cuando el algoritmo de detección determine que existe un interbloqueo.

En este apartado se van a desarrollar estos puntos.

16.6.3.1 Detección de interbloqueos

Los interbloqueos se pueden describir con precisión por medio de un grafo dirigido llamado **grafo de espera**. Este grafo consiste en un par *G* = (*V*,*A*), siendo *V* el conjunto de vértices y A el conjunto de arcos. El conjunto de vértices consiste en todas las transacciones del sistema. Cada elemento del conjunto E de arcos es un par ordenado $T_i \rightarrow T_j$. Si $T_i \rightarrow T_j$ pertenece a A entonces hay un arco dirigido de la transacción T_i a T_j, lo cual implica que la transacción T_i está esperando a que la transacción T_j libere un elemento de datos que necesita.

Cuando la transacción T_i solicita un elemento de datos que posee actualmente la transacción T_j, entonces se inserta el arco $T_i \rightarrow T_j$ en el grafo de espera. Este arco sólo se borra cuando la transacción T_j deja de poseer un elemento de datos que necesite la transacción T_i.

Existe un interbloqueo en el sistema si y sólo si el grafo de espera contiene un ciclo. Se dice que toda transacción involucrada en el ciclo está interbloqueada. Para detectar los interbloqueos el sistema debe mantener el grafo de espera y debe invocar periódicamente a un algoritmo que busque un ciclo en el grafo.

Para ilustrar estos conceptos considérese el grafo de espera de la Figura 16.18, el cual describe la situación siguiente:

- La transacción T_{25} está esperando a las transacciones T_{26} y T_{27}.
- La transacción T_{27} está esperando a la transacción T_{26}.
- La transacción T_{26} está esperando a la transacción T_{28}.

Puesto que el grafo no tiene ciclos, el sistema no está en un estado de interbloqueo.

Supóngase ahora que la transacción T_{28} solicita un elemento que posee T_{27}. Se añade el arco $T_{28} \rightarrow T_{27}$ al grafo de espera, lo que resulta en el nuevo estado que se ilustra en la Figura 16.19. Ahora el grafo tiene el ciclo

$$T_{26} \rightarrow T_{28} \rightarrow T_{27} \rightarrow T_{26}$$

lo que implica que las transacciones T_{26}, T_{27} y T_{28} están interbloqueadas.

Por consiguiente surge la pregunta: ¿cuándo se debe invocar al algoritmo de detección? La respuesta depende de dos factores:

1. ¿Cada cuánto tiempo tiene lugar un interbloqueo?
2. ¿Cuántas transacciones se verán afectadas por el interbloqueo?

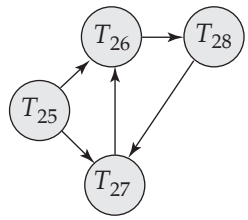


Figura 16.19 Grafo de espera con un ciclo.

Si los interbloqueos ocurren con frecuencia, entonces se debe invocar al algoritmo de detección con mayor frecuencia de la usual. Los elementos de datos asignados a las transacciones interbloqueadas no estarán disponibles para otras transacciones hasta que pueda romperse el interbloqueo. Adicionalmente puede también aumentar el número de ciclos en el grafo. En el peor caso se invoca al algoritmo de detección cada vez que una petición de asignación no se pueda conceder inmediatamente.

16.6.3.2 Recuperación de interbloqueos

Cuando un algoritmo de detección de interbloqueos determina que existe un interbloqueo, el sistema debe **recuperarse** del mismo. La solución más común es retroceder una o más transacciones para romper el interbloqueo. Se deben realizar tres acciones:

1. **Selección de una víctima**. Dado un conjunto de transacciones interbloqueadas se debe determinar la transacción (o transacciones) que se van a retroceder para romper el interbloqueo. Se deben retroceder aquellas transacciones que incurran en un coste mínimo. Desafortunadamente, el término *coste mínimo* no es preciso. Hay muchos factores que determinan el coste de un retroceso, como
   a. Lo que la transacción ha calculado y lo que se calculará hasta completar su tarea designada.
   b. El número de elementos de datos que ha usado la transacción.
   c. La cantidad de elementos de datos que necesita la transacción para que se complete.
   d. El número de transacciones que se verán involucradas en el retroceso.

2. **Retroceso**. Una vez que se ha decidido que retrocederá una transacción en particular, se debe determinar hasta dónde retrocederá dicha transacción.

   La solución más simple consiste en un **retroceso total**: se aborta la transacción y luego vuelve a comenzar. Sin embargo, es más efectivo hacer retroceder la transacción sólo lo necesario para romper el interbloqueo. Dicho **retroceso parcial** requiere que el sistema mantenga información adicional sobre el estado de todas las transacciones que están en ejecución. Concretamente, es necesario registrar la secuencia de solicitudes/concesiones de bloqueos y de actualizaciones realizadas por la transacción. El mecanismo de detección de interbloqueos debería decidir qué bloqueos necesita liberar la transacción seleccionada para romper el interbloqueo. Hay que hacer retroceder la transacción seleccionada hasta el punto donde obtuvo el primero de esos bloqueos, deshaciendo todas las acciones que realizó desde ese punto. El mecanismo de recuperación debe ser capaz de realizar tales retrocesos parciales. Además, las transacciones deben ser capaces de reanudar la ejecución después de un retroceso parcial. Véase el apartado de notas bibliográficas para obtener las referencias pertinentes.

3. **Inanición**. En un sistema en el cual la selección de víctimas esté basada principalmente en factores de coste, puede ocurrir que siempre se elija a la misma transacción como víctima. El resultado es que esta transacción no completa nunca su tarea designada. Dicha situación se denomina **inanición**. Se debe asegurar que una transacción pueda elegirse como víctima sólo un número finito (y pequeño) de veces. La solución más común consiste en incluir en el factor de coste el número de retrocesos.

16.7 Operaciones para insertar y borrar

Hasta ahora se ha centrado la atención en las operaciones leer y escribir. Esta ligadura limita a las transacciones a los elementos de datos que ya están en la base de datos. Algunas transacciones necesitan no sólo acceder a los elementos de datos existentes, sino también poder crear nuevos elementos de datos. Otras necesitan tener la posibilidad de borrar elementos de datos. Para examinar la forma en que tales transacciones afectan al control de concurrencia se introducen las operaciones adicionales siguientes:

- borrar(Q) borra de la base de datos el elemento de datos Q.
- insertar(Q) inserta en la base de datos el nuevo elemento de datos Q y le asigna un valor inicial.

Si la transacción T_i intenta ejecutar una operación leer(Q) después de haberse borrado Q se produce un error lógico en T_i. Igualmente, si la transacción T_i intenta ejecutar una operación leer(Q) antes de que Q se haya insertado produce un error lógico en T_i. También es un error lógico intentar borrar un dato inexistente.

16.7.1 Borrado

Para comprender la manera en que puede afectar la presencia de las instrucciones borrar al control de concurrencia, se debe decidir en qué casos una instrucción borrar está en conflicto con otra instrucción. Sean I_i e I_j instrucciones de T_i y T_j, respectivamente, que están consecutivas en la planificación S. Sea I_i = borrar(Q). Se consideran distintas instrucciones I_j:

- I_j = leer(Q). I_i e I_j están en conflicto. Si I_i está antes de I_j, T_j tendrá un error lógico. Si $I_j$está antes de I_i, T_j puede ejecutar con éxito su operación leer.
- I_j = escribir(Q). I_i e I_j están en conflicto. Si I_i está antes de I_j, T_j tendrá un error lógico. Si $I_j$está antes de I_i, T_j puede ejecutar con éxito su operación escribir.
- I_j = borrar(Q). I_i e $I_j$están en conflicto. Si I_i está antes de I_j, T_j tendrá un error lógico. Si $I_j$está antes de I_i, T_i tendrá un error lógico.
- I_j = insertar(Q). I_i e $I_j$están en conflicto. Supóngase que no existe el elemento de datos Q antes de la ejecución de I_i e I_j. Entonces si I_i está antes de I_j, hay un error lógico en T_i. Si $I_j$está antes de I_i, entonces no hay ningún error lógico. Igualmente si Q existía antes de la ejecución de I_i e I_j, entonces hay un error lógico si $I_j$está antes de I_i, pero no en otro caso.

Se puede concluir lo siguiente:

- En el protocolo de dos fases se necesita un bloqueo exclusivo en un elemento de datos antes de que se borre dicho elemento.
- En el protocolo de ordenación por marcas temporales se debe hacer una prueba similar que la que se hacía con escribir. Supóngase que la transacción T_i ejecuta borrar(Q).
  - ☐ Si MT(T_i) < marca-temporal-L(Q), entonces el valor de Q que va a borrar T_i lo ha leído ya otra transacción T_j con MT(T_j) >MT(T_i). Por tanto se rechaza la operación borrar y T_i retrocede.
  - ☐ Si MT(T_i) < marca-temporal-E(Q), entonces la transacción T_j con MT(T_j) >MT(T_i) ha escrito Q. Por tanto se rechaza esta operación borrar y T_i retrocede.
  - ☐ En otro caso se ejecuta la operación borrar.

16.7.2 Inserción

Ya se ha visto que una operación insertar(Q) está en conflicto con una operación borrar(Q). De forma similar insertar(Q) está en conflicto con las operaciones leer(Q) y escribir(Q). No se pueden realizar operaciones leer o escribir sobre un elemento de datos hasta que este último exista.

Puesto que insertar(Q) asigna un valor al elemento de datos Q, se trata insertar de forma similar a escribir desde el punto de vista del control de concurrencia:

- En el protocolo de bloqueo de dos fases, si T_i realiza un operación insertar(Q), se da a T_i un bloqueo exclusivo sobre el elemento de datos Q recientemente creado.
- En el protocolo de ordenación por marcas temporales, si T_i realiza una operación insertar(Q), se fijan los valores marca-temporal-L(Q) y marca-temporal-E(Q) a MT(T_i).

16.7.3 El fenómeno fantasma

Considérese una transacción T_{29} que ejecuta la siguiente pregunta SQL a la base de datos bancaria:

select sum(*saldo*)
from *cuenta*
where *nombre_sucursal* = 'Pamplona'

La transacción T_{29} necesita acceder a todas las tuplas de la relación *cuenta* que pertenezcan a la sucursal Pamplona.

Sea T_{30} una transacción que ejecuta la siguiente inserción SQL:

insert into *cuenta*
values (C-201, 'Pamplona', 900)

Sea P una planificación que involucra a T_{29} y T_{30}. Se espera que haya un conflicto potencial debido a las razones siguientes:

- Si T_{29} utiliza la tupla que ha insertado recientemente T_{30} al calcular **sum**(*saldo*), entonces T_{29} lee el valor que ha escrito T_{30}. Así, en una planificación secuencial equivalente a S, T_{30} debe ir antes de T_{29}.
- Si T_{29} no utiliza la tupla que ha insertado recientemente T_{30} al calcular **sum**(*saldo*), entonces en una planificación secuencial equivalente a S, T_{29} debe ir antes de T_{30}.

El segundo caso de los dos es curioso. T_{29} y T_{30} no acceden a ninguna tupla común, ¡y sin embargo están en conflicto! En efecto, T_{29} y T_{30} están en conflicto en una tupla fantasma. Si se realiza el control de concurrencia con granularidad de tupla, no se detecta dicho conflicto. Como resultado, el sistema podría no impedir una planificación no secuenciable. Este problema recibe el nombre de **fenómeno fantasma**.

Para evitar el fenómeno fantasma se permite que la transacción T_{29} impida a otras transacciones crear nuevas tuplas en la relación *cuenta* con *nombre_sucursal* = "Pamplona".

Para encontrar todas las tuplas de *cuenta* con *nombre_sucursal* = "Pamplona", T_{29} debe buscar o bien en toda la relación *cuenta*, o al menos en un índice de la relación. Hasta ahora se ha asumido implícitamente que los únicos elementos de datos a los que accede una transacción son tuplas. Sin embargo T_{29} es un ejemplo de transacción que lee información acerca de qué tuplas pertenecen a una relación, y T_{30} es un ejemplo de transacción que actualiza dicha información.

Claramente no es suficiente bloquear las tuplas a las que se accede; se necesita también bloquear la información acerca de qué tuplas pertenecen a la relación.

La solución más simple a este problema consiste en asociar un elemento de datos con la propia relación; el elemento de datos representa la información utilizada para encontrar las tuplas en la relación. Las transacciones como T_{29}, que lean la información acerca de qué tuplas pertenecen a la relación, tendrían que bloquear el elemento de datos correspondientes a la relación en modo compartido. Las transacciones como T_{30}, que actualicen la información acerca de qué tuplas pertenecen a la relación, tendrían que bloquear el elemento de datos en modo exclusivo. De este modo T_{29} y T_{30} tendrían un conflicto en un elemento de datos real, en lugar de tenerlo en uno fantasma.

No se debe confundir el bloqueo de una relación completa, como en el bloqueo de granularidad múltiple, con el bloqueo del elemento de datos correspondiente a la relación. Al bloquear el elemento de datos la transacción sólo evita que otras transacciones actualicen la información acerca de qué tuplas pertenecen a la relación. Sigue siendo necesario un bloqueo de tuplas. Una transacción que acceda directamente a una tupla puede obtener un bloqueo sobre las tuplas incluso si otra transacción posee un bloqueo exclusivo sobre el elemento de datos correspondiente a la propia relación.

El inconveniente principal de bloquear un elemento de datos correspondiente a la relación es el bajo grado de concurrencia—se impide que dos transacciones que inserten distintas tuplas en la relación se ejecuten concurrentemente.

Una solución mejor es la técnica de **bloqueo del índice**. Toda transacción que inserte una tupla en una relación debe insertar información en cada uno de los índices que se mantengan en la relación. Se elimina el fenómeno fantasma al imponer un protocolo para los índices. Para simplificar, sólo se van a considerar los índices del árbol B^+.

Como se vio en el Capítulo 12, todo valor de la clave de búsqueda se asocia a un nodo hoja índice. Una consulta usará normalmente uno o más índices para acceder a la relación. Una inserción debe insertar una nueva tupla en todos los índices de la relación. En el ejemplo se asume que hay un índice para *cuenta* en *nombre_sucursal*. Entonces T_{30} debe modificar la hoja que contiene la clave Pamplona. Si T_{29} lee el mismo nodo hoja para localizar todas las tuplas que pertenecen a la sucursal Pamplona, entonces T_{29} y T_{30} tienen un conflicto en dicho nodo hoja.

El **protocolo de bloqueo del índice** toma las ventajas de la disponibilidad de índices en una relación convirtiendo las apariciones del fenómeno fantasma en conflictos en los bloqueos sobre los nodos hoja índice. El protocolo opera de la siguiente manera:

- Toda relación debe tener al menos un índice.
- Una transacción T_i puede acceder a las tuplas de una relación únicamente después de haberlas encontrado primero a través de uno o más índices de la relación.
- Una transacción T_i que realiza una búsqueda (o bien una búsqueda de rango o una búsqueda concreta) debe bloquear en modo compartido todos los nodos hoja índice a los que accede.
- Una transacción T_i no puede insertar, borrar o actualizar una tupla t_i en una relación r sin actualizar todos los índices de r. La transacción debe obtener bloqueos en modo exclusivo sobre todos los nodos hoja índice que están afectados por la inserción, el borrado o la actualización. Para la inserción y el borrado, los nodos hoja afectados son aquellos que contienen (después de la inserción) o han contenido (antes de la modificación) el valor de la clave de búsqueda en la tupla. Para las actualizaciones, los nodos hoja afectados son los que (antes de la modificación) contienen el valor antiguo de la clave de búsqueda y los nodos que (después de la modificación) contienen el nuevo valor de la clave de búsqueda.
- Hay que cumplir las reglas del protocolo de bloqueo de dos fases.

Existen variantes de la técnica de bloqueo del índice para eliminar el fenómeno fantasma con otros protocolos de control de concurrencia que se han presentado en este capítulo.

16.8 Niveles débiles de consistencia

La secuencialidad es un concepto útil porque permite a los programadores ignorar los problemas relacionados con la concurrencia cuando codifican las transacciones. Si todas las transacciones tienen la propiedad de mantener la consistencia de la base de datos si se ejecutan por separado, la secuencialidad asegura que las ejecuciones concurrentes mantienen la consistencia. Sin embargo, puede que los protocolos necesarios para asegurar la secuencialidad permitan muy poca concurrencia para algunas aplicaciones. En estos casos se utilizan los niveles más débiles de consistencia. El uso de niveles más débiles de consistencia añade una nueva carga a los programadores para asegurar la corrección de las bases de datos.

16.8.1 Consistencia de grado dos

El objetivo de la **consistencia de grado dos** es evitar abortar en cascada sin asegurar necesariamente la secuencialidad. El protocolo de bloqueo para la consistencia de grado dos utiliza los mismos dos modos de bloqueo que se utilizan para el protocolo de bloqueo de dos fases: compartido (C) y exclusivo (X). Las transacciones deben mantener el modo de bloqueo adecuado cuando tengan acceso a un elemento de datos.

T_3	T_4
bloquear-C(Q)	
leer(Q)	
desbloquear(Q)	
	bloquear-X(Q)
	leer(Q)
	escribir(Q)
	desbloquear(Q)
bloquear-C(Q)	
leer(Q)	
desbloquear(Q)	

Figura 16.20 Planificación no secuenciable con consistencia de grado dos.

A diferencia de la situación en los bloqueos de dos fases, los bloqueos C pueden liberarse en cualquier momento y también se pueden establecer bloqueos en cualquier momento. Los bloqueos exclusivos no se pueden liberar hasta que la transacción se comprometa o se aborte. La secuencialidad no queda asegurada por este protocolo. En realidad, una transacción puede leer dos veces el mismo elemento de datos y obtener resultados diferentes. En la Figura 16.20, T_3 lee el valor de Q antes y después de que T_4 escriba su valor.

La posibilidad de que se produzca inconsistencia con la consistencia de grado dos hace que este enfoque no sea conveniente para muchas aplicaciones.

16.8.2 Estabilidad del cursor

La **estabilidad del cursor** es una forma de consistencia de grado dos diseñada para programas escritos en lenguajes de propósito general, los cuales iteran sobre las tuplas de una relación utilizando cursores. En vez de bloquear toda la relación, la estabilidad del cursor asegura que:

- La tupla que está procesando la iteración esté bloqueada en modo compartido.
- Todas las tuplas modificadas estén bloqueadas en modo exclusivo hasta que se comprometa la transacción.

Estas reglas aseguran que se obtiene consistencia de grado dos. No se requiere bloqueo de dos fases. No se garantiza la secuencialidad. La estabilidad del cursor se utiliza en la práctica sobre relaciones a las que se accede muchas veces como una forma de incrementar la concurrencia y mejorar el rendimiento del sistema. Las aplicaciones que utilizan estabilidad del cursor deben ser desarrolladas de forma que aseguren la consistencia de la base de datos a pesar de la posibilidad de planificaciones no secuenciables. Por tanto, el uso de estabilidad del cursor está limitado a determinadas situaciones con restricciones simples de consistencia.

16.8.3 Niveles débiles de consistencia en SQL

La norma SQL también permite que una transacción especifique si puede ser ejecutada de tal forma que se convierta en no secuenciable con respecto a otras transacciones. Por ejemplo, una transacción puede operar en el nivel **sin compromiso de lectura**, lo que permite que la transacción lea registros incluso si éstos no se han comprometido. SQL proporciona tales características para transacciones largas cuyos resultados no necesitan ser precisos. Por ejemplo, una información aproximada suele ser suficiente para las estadísticas utilizadas en la optimización de consultas. Si estas transacciones se ejecutaran en modo secuencial, podrían interferir con otras transacciones, provocando que la ejecución de las otras se retrasara.

Los niveles de consistencia especificados por SQL-92son los siguientes:

- **Secuenciable**. Es el predeterminado.

- **Lectura repetible**. Sólo permite leer registros comprometidos, y además requiere que, entre dos lecturas de un registro realizadas por una transacción, no se permita que ninguna otra transacción actualice el registro. Sin embargo, la transacción puede no ser secuenciable con respecto a otras transacciones. Por ejemplo, cuando está buscando registros que satisfagan algunas condiciones, una transacción podría encontrar algunos de los registros que ha insertado una transacción comprometida, pero podría no encontrar otros.
- **Compromiso de lectura**. Sólo permite leer registros comprometidos, pero ni siquiera requiere lecturas repetibles. Por ejemplo, entre dos lecturas de un registro realizadas por una transacción, los registros deben ser actualizados por otras transacciones comprometidas. Esto es básicamente lo mismo que la consistencia de grado dos; la mayoría de los sistemas que soportan este nivel de consistencia debería implementar en realidad estabilidad del cursor, que es un caso especial de consistencia de grado dos.
- **Sin compromiso de lectura**. Permite incluso leer registros no comprometidos. Éste es el nivel de consistencia más bajo que permite SQL-92.

16.9 Concurrencia en los índices**

Es posible tratar el acceso a los índice como el de otras estructuras de base de datos y aplicar las técnicas de control de concurrencia que se han descrito anteriormente. Sin embargo, puesto que se accede frecuentemente a los índices, se pueden convertir en un punto con mucho bloqueo, lo que produce un bajo grado de concurrencia. Por suerte, no es necesario tratar los índices como a las demás estructuras de base de datos. Es perfectamente aceptable que una transacción busque en un índice dos veces y se encuentre con que la estructura del índice ha cambiado entre ambas búsquedas, mientras la búsqueda devuelva el conjunto correcto de tuplas. De este modo se acepta tener un acceso no secuenciable a un índice mientras siga siendo correcto dicho índice.

A continuación se mostrarán dos técnicas para tratar los accesos concurrentes a árboles B^+. En las notas bibliográficas se hace referencia a otras técnicas para árboles B^+, así como a técnicas para otras estructuras de índice.

Las técnicas que se presentan para el control de concurrencia en los árboles B^+ se basan en el bloqueo, pero no se emplean ni el bloqueo de dos fases ni el protocolo de árbol. Los algoritmos de búsqueda, inserción y borrado son los mismos que se usan en el Capítulo 12 con sólo algunas pequeñas modificaciones.

La primera técnica se denomina **protocolo del cangrejo**:

- Cuando se busca un valor clave, el protocolo del cangrejo bloquea primero el nodo raíz en modo compartido. Cuando se recorre el árbol hacia abajo, adquiere un bloqueo compartido sobre el siguiente nodo hijo. Después de adquirir el bloqueo sobre el nodo hijo, libera el bloqueo sobre el nodo padre. Repite este proceso hasta que alcanza un nodo hoja.
- Cuando se inserta o se borra un valor clave, el protocolo del cangrejo realiza estas acciones:
  - Sigue el mismo protocolo que para la búsqueda hasta que alcanza el nodo hoja deseado. Hasta este punto, tan sólo obtiene (y libera) bloqueos compartidos.
  - Bloquea el nodo hoja en modo exclusivo e inserta o borra el valor clave.
  - Si necesita dividir un nodo o fusionarlo con sus hermanos, o redistribuir los valores claves entre hermanos, el protocolo del cangrejo bloquea al padre del nodo en modo exclusivo. Después de realizar estas acciones, libera los bloqueos sobre el nodo y los hermanos.

    Si el padre requiere división, fusión o redistribución de valores clave, el protocolo mantiene el bloqueo sobre el padre, y la división, la fusión o la redistribución se sigue propagando de la misma manera. En otro caso, libera el bloqueo sobre el padre.

El protocolo obtiene su nombre de la forma en que los cangrejos avanzan moviéndose de lado, moviendo las patas de un lado, después las patas del otro, y así alternando sucesivamente. El avance de los bloqueos mientras el protocolo baja por el árbol o sube de nuevo (en el caso de divisiones, fusiones o redistribuciones) actúa de forma similar a la del cangrejo.

Una vez que una operación particular libera un bloqueo sobre un nodo, otras operaciones pueden acceder a ese nodo. Existe una posibilidad de interbloqueos entre las operaciones de búsqueda que bajan por el árbol, y las divisiones, fusiones y redistribuciones que se propagan hacia arriba por el árbol. El sistema puede manejar con facilidad tales interbloqueos reiniciando la operación de búsqueda desde la raíz, después de liberar los bloqueos mantenidos por la operación.

La segunda técnica consigue incluso más concurrencia, impidiendo incluso que se mantenga un bloqueo sobre un nodo mientras se está adquiriendo el bloqueo sobre otro nodo, utilizando una versión modificada de los árboles B^+ llamados **árboles B enlazados**; los árboles B enlazados requieren que todo nodo (incluyendo los nodos internos, no sólo las hojas) mantenga un puntero a su hermano derecho. Se necesita este puntero porque una búsqueda que tenga lugar mientras se divide un nodo puede que tenga que buscar no sólo ese nodo sino también el hermano derecho de ese nodo (si existe alguno). Esta técnica se va a ilustrar con un ejemplo después de presentar los procedimientos modificados del **protocolo de bloqueo con árboles B enlazados**.

- **Búsqueda**. Se debe bloquear en modo compartido cada nodo del árbol B^+ antes de que se acceda a él. Dicho bloqueo se libera antes de que se solicite algún otro bloqueo sobre algún nodo del árbol B^+. Si tiene lugar una división de forma concurrente con una búsqueda, el valor de la clave de búsqueda deseado puede dejar de aparecer dentro del rango de valores que representa un nodo que se ha accedido en la búsqueda. En tal caso se representa el valor de la clave de búsqueda por medio de un nodo hermano, el cual coloca el sistema siguiendo el puntero al hermano derecho. Sin embargo, el sistema bloquea los nodos hoja siguiendo el protocolo de bloqueo de dos fases, como se describe en el Apartado 16.7.3, para evitar el fenómeno fantasma.
- **Inserción y borrado**. El sistema sigue las reglas de la búsqueda para localizar el nodo sobre el cual se va a realizar la inserción o el borrado. Se modifica el bloqueo en modo compartido sobre ese nodo a modo exclusivo y se realiza la inserción o el borrado. Se bloquean los nodos hoja afectados por la inserción o el borrado siguiendo el protocolo de bloqueo de dos fases, como se describe en el Apartado 16.7.3, para evitar el fenómeno fantasma.
- **División**. Si se divide un nodo se crea otro nuevo siguiendo el algoritmo del Apartado 12.3 y se convierte en el hermano derecho del nodo original. Se fijan los punteros al hermano derecho del nodo original y del nuevo nodo. Seguidamente se libera el bloqueo en modo exclusivo sobre el nodo original (dado que es un nodo interno, los nodos hoja están bloqueados en dos fases) y se solicita un bloqueo sobre el padre para que se pueda insertar un nuevo nodo (no es necesario bloquear o desbloquear el nuevo nodo).
- **Fusión**. Si un nodo tiene muy pocos valores de clave de búsqueda después de un borrado, se debe bloquear en modo exclusivo el nodo con el que se debe fusionar. Una vez que se fusionen estos nodos se solicita un bloqueo en modo exclusivo sobre el padre para que se pueda eliminar el nodo borrado. En ese momento se libera el bloqueo sobre los nodos fusionados. Se libera el bloqueo sobre el nodo padre a no ser que también se tenga que fusionar.

Es importante observar que: una inserción o un borrado pueden bloquear un nodo, desbloquearlo y posteriormente volverlo a bloquear. Además, una búsqueda que se ejecute concurrentemente con operaciones de división o de fusión puede observar que la clave de búsqueda deseada se ha trasladado al nodo hermano derecho debido a la división o a la fusión.

Como ejemplo considérese el árbol B^+ de la Figura 16.21. Supóngase que hay dos operaciones concurrentes sobre dicho árbol B^+:

1. Insertar “Cádiz”.
2. Buscar “Daimiel”.

Considérese que la operación inserción comienza en primer lugar. Realiza una búsqueda de “Cádiz” y encuentra que el nodo en el cual se debe insertar “Cádiz” está vacío. Por tanto convierte el bloqueo compartido sobre el nodo en un bloqueo exclusivo y crea un nuevo nodo. El nodo original contiene

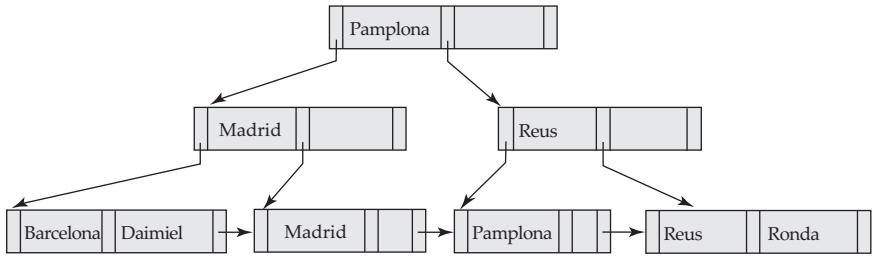


Figura 16.21 Árbol B^+ para el archivo *cuenta* con $n = 3$.

ahora los valores de clave de búsqueda "Barcelona" y "Cádiz". El nuevo nodo contiene el valor de clave de búsqueda "Daimiel".

Supóngase ahora que se produce un cambio de contexto que le pasa el control a la operación búsqueda. Dicha operación búsqueda accede a la raíz y sigue el puntero al hijo izquierdo de la raíz. Accede entonces a ese nodo y obtiene el puntero al hijo izquierdo. El hijo izquierdo contenía originalmente los valores de clave de búsqueda "Barcelona" y "Daimiel". Puesto que la operación inserción está bloqueando actualmente en modo exclusivo dicho nodo, la operación búsqueda debe esperar. Obsérvese que, en este punto, ¡la operación búsqueda no posee ningún bloqueo!

Ahora la operación inserción desbloquea el nodo hoja y vuelve a bloquear a su padre en esta ocasión en modo exclusivo. Se completa la inserción, lo que deja al árbol B^+ como se muestra en la Figura 16.22. Continúa la operación de búsqueda. Sin embargo tiene un puntero a un nodo hoja incorrecto. Sigue por tanto el puntero al hermano derecho para encontrar el nodo siguiente. Si este nodo es también incorrecto, se sigue también el puntero al hermano derecho. Se puede demostrar que, si una búsqueda tiene un puntero a un nodo incorrecto entonces, al seguir los punteros al hermano derecho, la búsqueda llega finalmente al nodo correcto.

Las operaciones de búsqueda y de inserción no pueden llevar a un interbloqueo. La fusión de nodos durante el borrado puede provocar inconsistencias, dado que una búsqueda puede tener que leer un puntero a un nodo borrado desde su padre, antes de que el nodo padre sea actualizado y, entonces, puede intentar acceder al nodo borrado. La búsqueda se tendría que reiniciar entonces desde la raíz. Dejar los nodos sin fusionar evita tales inconsistencias. Esta solución genera nodos que contienen muy pocos valores clave de búsqueda y que violan algunas propiedades de los árboles B^+. Sin embargo, en la mayoría de las bases de datos las inserciones son más frecuentes que los borrados, por lo que es probable que los nodos que tienen muy pocos valores claves de búsqueda ganen valores adicionales de forma relativamente rápida.

En lugar de bloquear nodos hoja índice en dos fases, algunos esquemas de control de concurrencia de índices utilizan **bloqueo de valores clave** sobre valores claves individuales, permitiendo que se inserten o se borren otros valores clave de la misma hoja. Por lo tanto, el bloqueo de valores clave proporciona una concurrencia mejorada. Sin embargo, utilizar el bloqueo de valores clave ingenuamente, podría permitir que se produjera el fenómeno fantasma; para prevenir el fenómeno fantasma se utiliza la técnica **bloqueo de la siguiente clave**. En esta técnica, cada búsqueda por índice debe bloquear no sólo las claves encontradas dentro del rango (o la única clave, en caso de una búsqueda concreta) sino también el siguiente valor clave—esto es, el valor clave que es justo mayor que el último valor clave que estaba

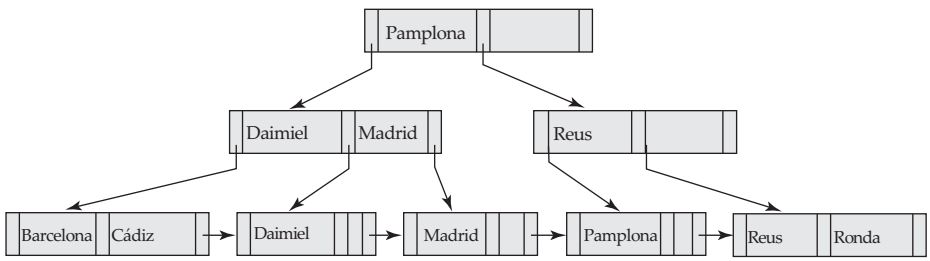


Figura 16.22 Inserción de "Cádiz" en el árbol B^+ de la Figura 16.21.

dentro del rango. Además, cada inserción no sólo debe bloquear el valor que se inserta, sino también el siguiente valor clave. Así, si una transacción intenta insertar un valor que estaba dentro del rango de la búsqueda por índice de otra transacción, las dos transacciones entrarán en conflicto en el valor clave que sigue al valor clave insertado. De forma similar, los borrados también deben bloquear el siguiente valor clave al valor que se ha borrado, para asegurar que se detecten los conflictos con las subsiguientes búsquedas de rango de otras consultas.

16.10 Resumen

- Cuando se ejecutan concurrentemente varias transacciones en la base de datos, puede dejar de conservarse la consistencia de los datos. Es necesario que el sistema controle la interacción entre las transacciones concurrentes, y dicho control se lleva a cabo mediante uno de los muchos mecanismos llamados esquemas de *control de concurrencia*.
- Se pueden usar varios esquemas de control de concurrencia para asegurar la secuencialidad. Todos estos esquemas o bien retrasan una operación o bien abortan la transacción que ha realizado la operación. Los más comunes son los protocolos de bloqueo, los esquemas de ordenación por marcas temporales, las técnicas de validación y los esquemas multiversión.
- Un protocolo de bloqueo es un conjunto de reglas, las cuales indican el momento en el que una transacción puede bloquear o desbloquear un elemento de datos de la base de datos.
- El protocolo de bloqueo de dos fases permite que una transacción bloquee un nuevo elemento de datos sólo si todavía no ha desbloqueado ningún otro elemento de datos. Este protocolo asegura la secuencialidad pero no la ausencia de interbloqueos. A falta de información acerca de la forma en que se accede a los elementos de datos, el protocolo de bloqueo de dos fases es necesario y suficiente para asegurar la secuencialidad.
- El protocolo de bloqueo estricto de dos fases permite liberar bloqueos exclusivos sólo al final de la transacción, para asegurar la recuperabilidad y la ausencia de cascadas en las planificaciones resultantes. El protocolo de bloqueo riguroso de dos fases libera todos los bloqueos sólo al final de la transacción.
- Los protocolos de bloqueo basados en grafos imponen restricciones sobre el orden de acceso a los elementos, y pueden por tanto asegurar la secuencialidad sin requerir el bloqueo de dos fases, y pueden además asegurar la ausencia de interbloqueos.
- El esquema de ordenación por marcas temporales asegura la secuencialidad seleccionando previamente un orden entre todo par de transacciones. Se asocia una única marca temporal fija a cada transacción del sistema. Las marcas temporales de las transacciones determinan el orden de secuencialidad. De este modo, si la marca temporal de la transacción T_i es más pequeña que la de la transacción T_j, entonces el esquema asegura que la planificación que ha producido es equivalente a una planificación secuencial en la cual la transacción T_i aparece antes de la transacción T_j. Lo asegura retrocediendo una transacción siempre que se viole dicho orden.
- Un esquema de validación es un método de control de concurrencia adecuado en aquellos casos en los que la mayoría de las transacciones son de sólo lectura, y por tanto la tasa de conflictos entre dichas transacciones es baja. Se asocia una única marca temporal fija a cada transacción del sistema. Se determina el orden de secuencialidad por medio de la marca temporal. Nunca se retrasa una transacción en dicho esquema. Debe pasar, sin embargo, una comprobación de validación para poder completarse. Si no pasa la comprobación de validación retrocede a su estado inicial.
- Hay circunstancias bajo las cuales puede ser conveniente agrupar varios elementos de datos y tratarlos como un conjunto de elementos de datos por motivos del trabajo, lo que da lugar a varios niveles de *granularidad*. Se permiten elementos de datos de varios tamaños y se define una jerarquía de elementos de datos en la cual los elementos más pequeños están anidados dentro de otros más grandes. Dicha jerarquía se puede representar de forma gráfica como un árbol. El orden de obtención de los bloqueos es desde la raíz hasta la hojas; se liberan desde las hojas hasta la raíz. Este protocolo asegura la secuencialidad pero no la ausencia de interbloqueos.

- Un esquema de control de concurrencia multiversión se basa en crear una nueva versión de un elemento de datos cada vez que una transacción va a escribir dicho elemento. Cuando se realiza una operación de lectura, el sistema elige una de las versiones para que se lea. El esquema de control de concurrencia asegura que la versión que se va a leer se elige de forma que asegure la secuencialidad usando las marcas temporales. Una operación de lectura tiene éxito siempre.
  - En la ordenación por marcas temporales multiversión, una operación de escritura puede provocar el retroceso de una transacción.
  - En el bloqueo de dos fases multiversión las operaciones de escritura pueden provocar una espera con bloqueo o posiblemente un interbloqueo.
- Algunos de los protocolos de bloqueo no evitan los interbloqueos. Una forma de prevenir los interbloqueos es utilizar una ordenación de los elementos de datos, y solicitar los bloqueos en una secuencia consistente con la ordenación.
- Otra forma de prevenir los interbloqueos es utilizar expropiación y retroceso de transacciones. Para controlar la expropiación se asigna una única marca temporal a cada transacción. El sistema utiliza estas marcas temporales para decidir si una transacción debe esperar o retroceder. Si una transacción retrocede conserva su marca temporal *anterior* cuando vuelve a comenzar. El esquema herir–esperar es un esquema de expropiación.
- Si no se pueden prevenir los interbloqueos, el sistema debe ocuparse de ellos utilizando el esquema de detección y recuperación de interbloqueos. Para hacer esto, el sistema construye un grafo de espera. Un sistema está en estado de interbloqueo si y sólo si contiene un ciclo en el grafo de espera. Cuando el algoritmo de detección de interbloqueos determina que existe un interbloqueo, el sistema debe recuperarse del interbloqueo. Esto se lleva a cabo retrocediendo una o más transacciones para romper el interbloqueo.
- Se puede realizar una operación borrar sólo si la transacción que borra la tupla tiene un bloqueo en modo exclusivo sobre dicha tupla. A la transacción que inserta una nueva tupla se le concede un bloqueo en modo exclusivo sobre dicha tupla.
- Las inserciones pueden provocar el fenómeno fantasma, en el cual hay un conflicto entre una inserción y una pregunta incluso si las dos transacciones no acceden a tuplas comunes. Tales conflictos no se pueden detectar si el bloqueo se ha hecho sólo sobre tuplas a las que han accedido transacciones. Es necesario bloquear los datos utilizados para encontrar las tuplas en la relación. La técnica del bloqueo del índice resuelve este problema al exigir bloqueos sobre ciertos cajones de índices. Estos bloqueos aseguran que todas las transacciones conflictivas están en conflicto por un elemento de datos real en lugar de por uno fantasma.
- Los niveles débiles de consistencia se utilizan en algunas aplicaciones cuando la consistencia de los resultados de la consulta no es crítica y, utilizar secuencialidad podría dar lugar a consultas que afectaran desfavorablemente al procesamiento de transacciones. La consistencia de grado dos es uno de los niveles de consistencia débiles; la estabilidad de cursor es un caso especial de consistencia de grado dos y se utiliza bastante. SQL:1999 permite a las consultas especificar el nivel de consistencia requerido.
- Se pueden desarrollar técnicas de control de concurrencia para estructuras especiales. A menudo se aplican técnicas especiales en los árboles B^+ para permitir una mayor concurrencia. Estas técnicas permiten accesos no secuenciables al árbol B^+, pero aseguran que la estructura del árbol B^+ es correcta y que los accesos a la base de datos son secuenciables.

Términos de repaso

- Control de concurrencia.
- Tipos de bloqueo:
  - Bloqueo en modo compartido (C).
  - Bloqueo en modo exclusivo (X).
- Bloqueo
  - Compatibilidad.
  - Solicitud.
  - Espera.
  - Concesión.
- Interbloqueo.

- Inanición.
- Protocolo de bloqueo.
- Planificación legal.
- Protocolo de bloqueo de dos fases.
  - ☐ Fase de crecimiento.
  - ☐ Fase de decrecimiento.
  - ☐ Punto de bloqueo.
  - ☐ Bloqueo estricto de dos fases.
  - ☐ Bloqueo riguroso de dos fases.
- Conversión de bloqueos.
  - ☐ Subir.
  - ☐ Bajar.
- Protocolos basados en grafos.
  - ☐ Protocolo de árbol.
  - ☐ Dependencia de compromiso.
- Protocolos basados en marcas temporales.
- Marca temporal.
  - ☐ Reloj del sistema.
  - ☐ Contador lógico.
  - ☐ marca-temporal-E(Q).
  - ☐ marca-temporal-L(Q).
- Protocolo de ordenación por marcas temporales.
  - ☐ Regla de escritura de Thomas.
- Protocolos basados en validación.
  - ☐ Fase de lectura.
  - ☐ Fase de validación.
  - ☐ Fase de escritura.
  - ☐ Comprobación de validación.
- Granularidad múltiple.
  - ☐ Bloqueos explícitos.
  - ☐ Bloqueos implícitos.
  - ☐ Bloqueos intencionales.
- Modos de bloqueo intencionales:
  - ☐ Intencional-compartido (IC).
  - ☐ Intencional-exclusivo (IX).
  - ☐ Intencional-exclusivo y compartido (IXC).
- Protocolo de bloqueo de granularidad múltiple.
- Control de concurrencia multiversión.
- Versiones.
- Ordenación por marcas temporales multiversión.
- Bloqueo de dos fases multiversión.
  - ☐ Transacciones de sólo lectura.
  - ☐ Transacciones de actualización.
- Tratamiento de interbloqueos.
  - ☐ Prevención.
  - ☐ Detección.
  - ☐ Recuperación.
- Prevención de interbloqueos.
  - ☐ Bloqueos ordenados.
  - ☐ Expropiación de bloqueos.
  - ☐ Esquema esperar–morir.
  - ☐ Esquema herir–esperar.
  - ☐ Esquemas basados en tiempo límite.
- Detección de interbloqueos.
  - ☐ Grafo de espera.
- Recuperación de interbloqueos.
  - ☐ Retroceso total.
  - ☐ Retroceso parcial.
- Operaciones para insertar y borrar.
- Fenómeno fantasma.
  - ☐ Protocolo de bloqueo del índice.
- Niveles débiles de consistencia:
  - ☐ Consistencia de grado dos.
  - ☐ Estabilidad del cursor.
  - ☐ Lectura repetible.
  - ☐ Compromiso de lectura.
  - ☐ Sin compromiso de lectura.
- Concurrencia en índices.
  - ☐ Cangrejo.
  - ☐ Árboles B enlazados.
  - ☐ Protocolo de bloqueo con árboles B enlazados.
  - ☐ Bloqueo de la siguiente clave.

Ejercicios prácticos

16.1 Demuéstrese que el protocolo de bloqueo de dos fases asegura la secuencialidad en cuanto a conflictos y que se pueden secuenciar las transacciones a través de sus puntos de bloqueo.

16.2 Considérense las dos transacciones siguientes:

T_{31}: leer(A);
leer(B);
si $A = 0$ **entonces** $B := B + 1$;
escribir(B).

T_{32}: leer(B);
leer(A);
si $B = 0$ **entonces** $A := A + 1$;
escribir(A).

Añádanse a las transacciones T_{31} y T_{32} las instrucciones de bloqueo y desbloqueo para que sigan el protocolo de dos fases. ¿Puede producir la ejecución de estas transacciones un interbloqueo?

16.3 ¿Qué beneficio proporciona el bloqueo riguroso de dos fases? Compárese con otras formas de bloqueo de dos fases.

16.4 Considérese una base de datos organizada como un árbol con raíz. Supóngase que se inserta un nodo ficticio entre cada par de nodos. Demuéstrese que, si se sigue el protocolo de árbol con este nuevo árbol, se obtiene mayor concurrencia que con el árbol original.

16.5 Demuéstrese con un ejemplo que hay planificaciones que son posibles con el protocolo de árbol que no lo son con otros protocolos de bloqueo de dos fases y viceversa.

16.6 Considérese la siguiente extensión del protocolo de bloqueo de árbol que permite bloqueos compartidos y exclusivos:

- Una transacción puede ser de sólo lectura, en cuyo caso sólo puede solicitar bloqueos compartidos, o bien puede ser de actualización, en cuyo caso sólo puede solicitar bloqueos exclusivos.
- Cada transacción debe seguir las reglas del protocolo de árbol. Las transacciones de sólo lectura deben bloquear primero cualquier elemento de datos, mientras que las transacciones de actualización deben bloquear primero la raíz.

Demuéstrese que este protocolo asegura la secuencialidad y la ausencia de interbloqueos.

16.7 Considérese el siguiente protocolo de bloqueo basado en grafo, el cual sólo permite bloqueos exclusivos y que funciona con grafos de datos con forma de grafo dirigido acíclico con raíz.

- Una transacción puede bloquear en primer lugar cualquier nodo.
- Para bloquear cualquier otro nodo, la transacción debe poseer un bloqueo sobre la mayoría de los padres de dicho nodo.

Demuéstrese que este protocolo asegura la secuencialidad y la ausencia de interbloqueos.

16.8 Considérese el siguiente protocolo de bloqueo basado en grafos que sólo permite bloqueos exclusivos y que funciona con grafos de datos con forma de grafo dirigido acíclico con raíz.

- Una transacción puede bloquear en primer lugar cualquier nodo.
- Para bloquear cualquier otro nodo, la transacción debe haber visitado a todos los padres de dicho nodo y debe poseer un bloqueo sobre uno de los padres del vértice.

Demuéstrese que este protocolo asegura la secuencialidad y la ausencia de interbloqueos.

16.9 El bloqueo no se hace explícitamente en lenguajes de programación persistentes. En vez de esto se deben bloquear los objetos (o sus páginas correspondientes) cuando se accede a dichos objetos. Muchos de los sistemas operativos más modernos permiten al usuario definir protecciones de acceso (sin acceso, lectura, escritura) para las páginas, y aquellos accesos a memoria que violen las protecciones de acceso dan como resultado una violación de protección (véase la orden `mprotect` de Unix, por ejemplo). Descríbase la forma en que se puede usar el mecanismo de protección de acceso para bloqueos a nivel de página en lenguajes de programación persistentes.

16.10 Considérese una base de datos que tiene la operación atómica **incrementar** además de las operaciones leer y escribir. Sea V el valor del elemento de datos X.

	C	X	I
C	cierto	falso	falso
X	falso	falso	falso
I	falso	falso	cierto

Figura 16.23 Matriz de compatibilidad de bloqueos.

La operación

incrementar(X) en C

asigna el valor $V + C$ a X en un paso atómico. El valor de X no está disponible hasta que no se ejecute posteriormente una operación leer(X). En la Figura 16.23 se muestra una matriz de compatibilidad de bloqueos para tres tipos de bloqueo: modo compartido, exclusivo y de incremento.

a. Demuéstrese que, si todas las transacciones bloquean el dato al que acceden en el modo correspondiente, entonces el bloqueo de dos fases asegura la secuencialidad.

b. Demuéstrese que la inclusión del bloqueo en modo **incrementar** permite una mayor concurrencia. *Sugerencia*: considérense las transacciones de transferencia de fondos del ejemplo bancario.

16.11 En la ordenación por marcas temporales, **marca-temporal-E**(Q) indica la mayor marca temporal de todas las transacciones que hayan ejecutado escribir(Q) con éxito. Supóngase que en lugar de ello, **marca-temporal-E**(Q) se define como la marca temporal de la transacción más reciente que haya ejecutado escribir(Q) con éxito. ¿Hay alguna diferencia al cambiar esta definición? Razónese la respuesta.

16.12 La utilización de un bloqueo de granularidad múltiple puede necesitar más o menos bloqueos que en un sistema equivalente con una granularidad simple de bloqueo. Proporciónense ejemplos de ambas situaciones y compárese el aumento relativo de la concurrencia que se permite.

16.13 Considérese el esquema de control de concurrencia basado en la validación del Apartado 16.3. Demuéstrese que si se elige Validación(T_i) en lugar de Inicio(T_i) como marca temporal de la transacción T_i, se puede esperar una mejor respuesta en tiempo debido a que la tasa de conflictos entre las transacciones es realmente baja.

16.14 Para cada uno de los protocolos siguientes, descríbanse los aspectos de aplicación práctica que sugieran utilizar el protocolo y aspectos que sugieran no usarlo:

- Bloqueo de dos fases.
- Bloqueo de dos fases con granularidad múltiple.
- Protocolo de árbol.
- Ordenación por marcas temporales.
- Validación.
- Ordenación por marcas temporales multiversión.
- Bloqueo de dos fases multiversión.

16.15 Explíquese por qué la siguiente técnica de ejecución de transacciones puede proporcionar mayor rendimiento que la utilización del bloqueo estricto de dos fases: primero se ejecuta la transacción sin adquirir ningún bloqueo y sin realizar ninguna escritura en la base de datos como en las técnicas basadas en validación, pero a diferencia de las técnicas de validación no se realiza otra validación o escritura en la base de datos. En cambio, se vuelve a ejecutar la transacción utilizando bloqueo estricto de dos fases. *Sugerencia*: considérense esperas para la E/S de disco.

16.16 Considérese el protocolo de ordenación por marcas temporales, y dos transacciones, una que escribe dos elementos de datos p y q, y otra que lee los mismos dos elementos de datos. Obténgase una planificación por medio de la cual la comprobación por marcas temporales para una operación escribir falle y provoque el reinicio de la primera transacción, provocando a su vez una cancelación en cascada de la otra transacción. Muéstrese cómo esto podría acabar en inanición

de las dos transacciones. (Tal situación, donde dos o más procesos realizan acciones, pero no se puede completar la tarea porque se interacciona con otros procesos, se denomina **interbloqueo**).

16.17 Diséñese un protocolo basado en marcas temporales que evite el fenómeno fantasma.

16.18 Supóngase que se utiliza el protocolo de árbol del Apartado 16.1.5 para administrar el acceso concurrente a un árbol B^+. Puesto que puede haber una división en una inserción que afecte a la raíz, se deduce que una operación inserción no puede liberar ningún bloqueo hasta que no se complete la operación entera. ¿Bajo qué circunstancias es posible liberar antes un bloqueo?

Ejercicios

16.19 ¿Qué beneficio proporciona el bloqueo estricto de dos fases? ¿Qué inconvenientes tiene?

16.20 Muchas implementaciones de sistemas de bases de datos utilizan el bloqueo estricto de dos fases. Indíquense tres razones que expliquen la popularidad de este protocolo.

16.21 Considérese una variante del protocolo de árbol llamada protocolo de *bosque*. La base de datos está organizada como un bosque de árboles con raíz. Cada transacción T_i debe seguir las reglas siguientes:

- El primer bloqueo en un árbol puede hacerse sobre cualquier elemento de datos.
- Se pueden solicitar el segundo y posteriores bloqueos sólo si el padre del nodo solicitado está bloqueado actualmente.
- Se pueden desbloquear los elementos de datos en cualquier momento.
- T_i no puede volver a bloquear un elemento de datos después de haberlo desbloqueado.

Demuéstrese que el protocolo de bosque *no* asegura la secuencialidad.

16.22 Cuando una transacción retrocede en el protocolo de ordenación por marcas temporales se le asigna una nueva marca temporal. ¿Por qué no puede conservar simplemente su antigua marca temporal?

16.23 En el protocolo de granularidad múltiple, ¿qué diferencia hay entre bloqueo implícito y explícito?

16.24 Aunque el modo IXC es útil para el bloqueo de granularidad múltiple, no se usa un modo exclusivo e intencional-compartido (ICX). ¿Por qué no es útil?

16.25 Demuéstrese que hay planificaciones que son posibles con el protocolo de bloqueo de dos fases que no lo son con el protocolo de marcas temporales y viceversa.

16.26 En una versión modificada del protocolo de marcas temporales se necesita comprobar un bit de compromiso para saber si una petición de lectura debe esperar o no. Explíquese cómo puede evitar el bit de compromiso que aborten en cascada. ¿Por qué no se necesita hacer esta comprobación con las peticiones de escritura?

16.27 ¿Bajo qué condiciones es menos costoso evitar los interbloqueos que permitirlos y luego detectarlos?

16.28 Si se evitan los interbloqueos, ¿sigue siendo posible que haya inanición? Razónese la respuesta.

16.29 Explíquese el fenómeno fantasma. ¿Por qué produce este fenómeno una ejecución concurrente incorrecta a pesar de utilizar el protocolo de bloqueo de dos fases?

16.30 Explíquese la razón por la cual se utiliza la consistencia de grado dos. ¿Qué inconvenientes tiene esta técnica?

16.31 Dense ejemplos de planificaciones para mostrar que si, con el bloqueo de valores clave, cualquier búsqueda, inserción o borrado no bloquea el siguiente valor clave, el fenómeno fantasma podría ser indetectable.

16.32 Si varias transacciones modifican un elemento común (por ejemplo, el saldo de una sucursal) y elementos privados (por ejemplo, saldos de cuentas en concreto), explíquese cómo se puede aumentar el grado de concurrencia (y, por tanto, de la productividad) ordenando las operaciones de la transacción.

16.33 Considérese el siguiente protocolo de bloqueo: todos los elementos se numeran y, cada vez que un elemento se desbloquea, sólo se pueden bloquear elementos con un número mayor. Los bloqueos se pueden liberar en cualquier momento. Sólo se usan bloqueos X.

Muéstrese mediante un ejemplo que este bloqueo no garantiza la secuencialidad.

Notas bibliográficas

Gray y Reuter [1993] proporcionan un libro de texto detallado que cubre conceptos de procesamiento de transacciones, incluyendo conceptos de control de concurrencia y detalles de implementación. Bernstein y Newcomer [1997] proporcionan un libro de texto que trata varios aspectos del procesamiento de transacciones incluyendo el control de concurrencia.

Entre los primeros libros de texto que incluyen estudios sobre el control de concurrencia y la recuperación se incluyen los de Papadimitriou [1986] y Bernstein et al. [1987]. Gray [1978] presenta uno de los primeros estudios sobre aspectos de la implementación del control de concurrencia y la recuperación.

Eswaran et al. [1976] introdujo el protocolo de bloqueo de dos fases. El protocolo de bloqueo de árbol es de Silberschatz y Kedem [1980]. Yannakakis et al. [1979], Kedem y Silberschatz [1983] y Buckley y Silberschatz [1985] desarrollaron otros protocolos de bloqueo que no son de dos fases y que operan con grafos más generales. Korth [1983] explora varios modos de bloqueo que se pueden obtener a partir de los modos básicos, compartido y exclusivo.

El Ejercicio práctico 16.4 es de Buckley y Silberschatz [1984]. El Ejercicio 16.8 es de Kedem y Silberschatz [1983]. El Ejercicio 16.7 es de Kedem y Silberschatz [1979]. El Ejercicio práctico 16.8 es de Yannakakis et al. [1979]]. El Ejercicio práctico 16.10 es de Korth [1983].

El esquema de control de concurrencia basado en marcas temporales es de Reed [1983]. Bernstein y Goodman [1980] presentan una exposición de varios algoritmos de control de concurrencia basados en marcas temporales. Buckley y Silberschatz [1983] presentan un algoritmo de marcas temporales que no necesita retroceso para asegurar la secuencialidad. El esquema de control de concurrencia basado en validación es de Kung y Robinson [1981].

El protocolo de bloqueo para elementos de datos de granularidad múltiple es de Gray et al. [1975]. Gray et al. [1976] presentan una descripción detallada. Korth [1983] formaliza el bloqueo con granularidad múltiple para una colección arbitraria de modos de bloqueo (que permite más funcionalidades además de simplemente leer y escribir). Esta aproximación incluye una clase de modos de bloqueo llamados modos de *actualización* para permitir conversión de bloqueos. Carey [1983] extiende la idea de granularidad múltiple a la de control de concurrencia basado en marcas temporales. Korth [1982] presenta una extensión del protocolo que asegura la ausencia de interbloqueos.

Bernstein et al. [1983] ofrecen estudios acerca del control de concurrencia multiversión. En Silberschatz [1982] aparece un algoritmo de bloqueo de árbol multiversión. Reed [1978] y Reed [1983] introdujeron la ordenación por marcas temporales multiversión. Lai y Wilkinson [1984] describen un certificador de bloqueo de dos fases multiversión.

En Gray et al. [1975] se introduce la consistencia de grado dos. Los niveles de consistencia—o aislamiento—que ofrece SQL se explican y comentan en Berenson et al. [1995]. Muchos sistemas comerciales de bases de datos usan enfoques basados en versiones en combinación con el bloqueo. Oracle usa una forma de aislamiento basada en instantáneas basada en el protocolo descrito en el Apartado 16.5.2. Los detalles se encuentran en el Capítulo 27 y en Fekete et al. [2005]. El enfoque multiversión de PostgreSQL se trata en el Capítulo 26 y el de SQL Server (instantáneas) en el 29.

Bayer y Schkolnick [1977] y Johnson y Shasha [1993] estudiaron la concurrencia en árboles B^+. La técnica que se ha presentado en el Apartado 16.9 está basada en Kung y Lehman [1980], y en Lehman y Yao [1981]. En Mohan [1990a] y Mohan y Levine [1992] se describe la técnica del bloqueo del valor clave utilizada en ARIES que proporciona una gran concurrencia en el acceso a los árboles B^+. Ellis [1987] presenta una técnica de control de concurrencia para asociación lineal.

CAPÍTULO 17

Sistema de recuperación

Las computadoras, al igual que cualquier otro dispositivo eléctrico o mecánico, están sujetas a fallos. Éstos se producen por diferentes motivos como: fallos de disco, cortes de corriente, errores en el software, un incendio en la habitación de la computadora o incluso sabotaje. En cada uno de estos casos puede perderse información. Por tanto, el sistema de bases de datos debe realizar con anticipación acciones que garanticen que las propiedades de atomicidad y durabilidad de las transacciones, presentadas en el Capítulo 15, se preservan a pesar de tales fallos. Una parte integral de un sistema de bases de datos es un **esquema de recuperación**, el cual es responsable de la restauración de la base de datos al estado consistente previo al fallo. El esquema de recuperación también debe proporcionar **alta disponibilidad**; esto es, debe minimizar el tiempo durante el que la base de datos no se puede usar después de un fallo.

17.1 Clasificación de los fallos

En un sistema pueden producirse varios tipos de fallos, cada uno de los cuales requiere un tratamiento diferente. El tipo de fallo más fácil de tratar es el que no conduce a una pérdida de información en el sistema. Los fallos más difíciles de tratar son aquellos que provocan una pérdida de información. En este capítulo consideraremos sólo los siguientes tipos de fallos:

- **Fallo en la transacción**. Hay dos tipos de errores que pueden hacer que una transacción falle:
  - **Error lógico**. La transacción no puede continuar con su ejecución normal a causa de alguna condición interna, como una entrada incorrecta, datos no encontrados, desbordamiento o exceso del límite de recursos.
  - **Error del sistema**. El sistema se encuentra en un estado no deseado (por ejemplo, de interbloqueo) como consecuencia del cual una transacción no puede continuar con su ejecución normal. La transacción, sin embargo, se puede volver a ejecutar más tarde.
- **Caída del sistema**. Un mal funcionamiento del hardware o un error en el software de la base de datos o del sistema operativo causa la pérdida del contenido de la memoria volátil y aborta el procesamiento de una transacción. El contenido de la memoria no volátil permanece intacto y no se corrompe.

  La suposición de que los errores de hardware o software fuercen una parada del sistema, pero no corrompan el contenido de la memoria no volátil, se conoce como **supuesto de fallo-parada**. Los sistemas bien diseñados tienen numerosas comprobaciones internas, al nivel de hardware y de software, que abortan el sistema cuando existe un error. De aquí que el supuesto de fallo-parada sea razonable.
- **Fallo de disco**. Un bloque del disco pierde su contenido como resultado de una colisión de la cabeza lectora, o de un fallo durante una operación de transferencia de datos. Las copias de los

datos que se encuentran en otros discos o en archivos de seguridad en medios de almacenamiento secundarios, como cintas, se utilizan para recuperarse del fallo.

Para determinar el medio por el que el sistema debe recuperarse de los fallos, es necesario identificar los modos de fallo de los dispositivos de almacenamiento. A continuación se verá cómo afectan estos modos de fallo al contenido de la base de datos. Por tanto, se pueden proponer algoritmos para garantizar la consistencia de la base de datos y la atomicidad de las transacciones a pesar de los fallos. Estos algoritmos, conocidos como algoritmos de recuperación, constan de dos partes:

1. Acciones llevadas a cabo durante el procesamiento normal de transacciones para asegurar que existe información suficiente para permitir la recuperación frente a fallos.
2. Acciones llevadas a cabo después de ocurrir un fallo para restablecer el contenido de la base de datos a un estado que asegure la consistencia de la base de datos, la atomicidad de la transacción y la durabilidad.

17.2 Estructura del almacenamiento

Como vimos en el Capítulo 10, los diferentes elementos que componen una base de datos pueden ser almacenados y se puede acceder a ellos en diferentes medios de almacenamiento. Para entender cómo se pueden garantizar las propiedades de atomicidad y durabilidad de una transacción, se deben comprender mejor estos medios de almacenamiento y sus métodos de acceso.

17.2.1 Tipos de almacenamiento

En el Capítulo 11 se vio que los medios de almacenamiento se pueden distinguir según su velocidad relativa, capacidad y resistencia a fallos, y se pueden clasificar como almacenamiento volátil o no volátil. Se repasarán estos términos y se introducirá otra clase de almacenamiento, denominada **almacenamiento estable**.

- **Almacenamiento volátil**. La información que reside en almacenamiento volátil no suele sobrevivir a las caídas del sistema. La memoria principal y la memoria caché son ejemplos de este almacenamiento. El acceso al almacenamiento volátil es muy rápido, tanto por la propia velocidad de acceso a la memoria, como porque es posible acceder directamente a cualquier elemento de datos.
- **Almacenamiento no volátil**. La información que reside en almacenamiento no volátil sobrevive a las caídas del sistema. Los discos y las cintas magnéticas son ejemplos de este almacenamiento. Los discos se utilizan para almacenamiento en conexión, mientras que las cintas se usan para almacenamiento permanente. Ambos, sin embargo, pueden fallar (por ejemplo, colisión de la cabeza lectora) lo que puede conducir a una pérdida de información. En el estado actual de la tecnología, el almacenamiento no volátil es más lento en varios órdenes de magnitud que el almacenamiento volátil. Esta diferencia de velocidad es consecuencia de que los dispositivos de disco y de cinta sean electromecánicos, mientras que el almacenamiento volátil se basa por completo en circuitos integrados, como el almacenamiento volátil. En los sistemas de bases de datos los discos se utilizan fundamentalmente para el almacenamiento no volátil. Otros medios de almacenamiento no volátil sólo se usan normalmente para copias de seguridad de los datos. El almacenamiento flash (véase el Apartado 11.1), aunque no volátil, tiene capacidad insuficiente para la mayoría de los sistemas de bases de datos.
- **Almacenamiento estable**. La información que reside en almacenamiento estable *nunca* se pierde (bueno, *nunca digas nunca jamás* porque, teóricamente el *nunca* no puede garantizarse—por ejemplo, es posible, aunque extremadamente improbable, que un agujero negro se trague la tierra y destruya para siempre todos los datos). A pesar de que el almacenamiento estable es teóricamente imposible de conseguir, puede obtenerse una buena aproximación usando técnicas que hagan que la pérdida de información sea una posibilidad muy remota. La implementación del almacenamiento estable se estudia en el Apartado 17.2.2.

Las diferencias entre los distintos tipos de almacenamiento son, con frecuencia, menos claras en la práctica que en la presentación anterior. Ciertos sistemas están provistos de una fuente de alimentación de seguridad, por lo que determinada memoria principal puede sobrevivir a las caídas del sistema y a cortes de corriente. Otras formas alternativas de almacenamiento no volátil, como los medios ópticos, ofrecen un grado de confianza incluso más alto que el de los discos.

17.2.2 Implementación del almacenamiento estable

Para implementar almacenamiento estable se debe replicar la información necesaria en varios medios de almacenamiento no volátil (normalmente discos) con modos de fallo independientes, y actualizar esa información de manera controlada para asegurar que un fallo durante una transferencia de datos no dañará la información necesaria.

Recuérdese (del Capítulo 10) que los sistemas RAID (disposición redundante de discos independientes) garantizan que el fallo de un solo disco (incluso durante una transferencia de datos) no conduce a la pérdida de los datos. La variante más sencilla y rápida de RAID es el disco con imagen, que guarda dos copias de cada bloque en distintos discos. Otras formas de RAID ofrecen menores costes a expensas de un rendimiento inferior.

Los sistemas RAID, sin embargo, no pueden proteger contra las pérdidas de datos debidas a desastres tales como un incendio o una inundación. Muchos sistemas de almacenamiento guardan copias de seguridad de las cintas en otro lugar como protección frente a tales desastres. No obstante, como las cintas no pueden ser trasladadas a otro lugar continuamente, los cambios que se hayan realizado después del último traslado de las cintas se perderán en caso de un desastre tal. Los sistemas más seguros guardan una copia de cada bloque de almacenamiento estable en un lugar remoto, escribiéndola a través de una red de computadoras, además de almacenar el bloque en un sistema de discos locales. Como los bloques se envían al sistema remoto al mismo tiempo y de la misma forma que se guardan en almacenamiento local, una vez que una operación de este tipo se completa los bloques copiados no pueden perderse, incluso en el caso de que ocurriese un desastre como un incendio o una inundación. En el Apartado 17.8 se estudian estos sistemas de *copia de seguridad remota*.

En el resto de este apartado se estudia la manera de proteger los medios de almacenamiento de los errores durante una transferencia de datos. Las transferencias de bloques entre la memoria y el disco pueden acabar de diferentes formas:

- **Éxito**. La información transferida llega a su destino con seguridad.
- **Fallo parcial**. Ocurre un fallo en medio de la transferencia y el bloque de destino contiene información incorrecta.
- **Fallo total**. El fallo ocurre tan al principio de la transferencia que el bloque de destino permanece intacto.

Es necesario que, si se produce un **fallo durante una transferencia de datos**, el sistema lo detecte e invoque a un procedimiento de recuperación para restaurar el bloque a un estado estable. Para hacer esto, el sistema debe mantener dos bloques físicos por cada bloque lógico de la base de datos; en el caso de los discos con imagen, ambos bloques están en el mismo lugar; en el caso de copia de seguridad remota, uno de los bloques es local mientras que el otro está en un lugar remoto. Una operación de salida se ejecuta de la siguiente manera:

1. Se escribe la información en el primer bloque físico.
2. Cuando la primera escritura se completa con éxito, se escribe la misma información en el segundo bloque físico.
3. La salida está completada sólo después de que la segunda escritura finalice con éxito.

Durante la recuperación se examina cada par de bloques físicos. Si ambos coinciden y no existe ningún error detectable, entonces no son necesarias más acciones (recuérdese que los errores de los bloques del disco, como puede ser una escritura parcial del bloque, se detectan mediante el almacenamiento de

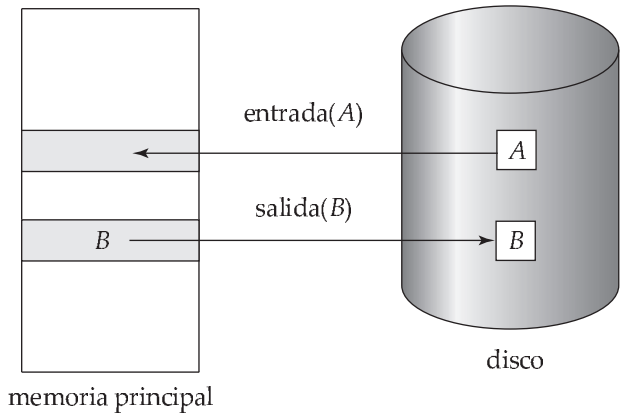


Figura 17.1 Operaciones de almacenamiento de bloques.

una suma de control en cada bloque). Si un bloque contiene un error detectable, se reemplaza su contenido por el del segundo bloque. Si ninguno de los dos bloques contiene errores detectables, pero su contenido es diferente, el sistema sustituye el contenido del primer bloque por el valor del segundo. Este procedimiento de recuperación garantiza que la escritura en almacenamiento estable o bien se completa con éxito (esto es, se actualizan todas las copias) o bien no produce ningún cambio.

El requisito de comparar cada par correspondiente de bloques durante la recuperación es bastante costoso. Puede reducirse considerablemente ese coste si se registran las escrituras de bloques que están en progreso utilizando una pequeña cantidad de RAM no volátil. En la recuperación solamente es necesario comparar aquellos bloques en los que se estuviera escribiendo.

Los protocolos para escribir un bloque en un lugar remoto son similares a los utilizados para escribir bloques en un sistema de disco con imagen, que fueron examinados en el Capítulo 11 y, en particular, en el Ejercicio práctico 11.2.

Este procedimiento puede extenderse fácilmente para permitir el uso de un número arbitrariamente alto de copias de cada bloque de almacenamiento estable. Aunque un elevado número de copias reduce la probabilidad de fallo incluso por debajo de la conseguida con dos copias, habitualmente es razonable la simulación de almacenamiento estable con sólo dos copias.

17.2.3 Acceso a los datos

Como se vio en el Capítulo 11, el sistema de bases de datos reside permanentemente en almacenamiento no volátil (normalmente discos) y se divide en unidades de almacenamiento de longitud fija denominadas **bloques**. Los bloques son las unidades de datos que se transfieren desde y hacia el disco y pueden contener varios elementos de datos. Supondremos que ningún elemento de datos mide dos o más bloques. Esta suposición es realista para la mayoría de las aplicaciones de procesamiento de datos tales como el ejemplo bancario.

Las transacciones llevan información del disco hacia la memoria principal y luego devuelven la información al disco. Las operaciones de entrada y salida se realizan en unidades de bloque. Nos referiremos a los bloques que residen en el disco como **bloques físicos**, y a los que residen temporalmente en la memoria principal como **bloques de memoria intermedia**. El área de memoria en donde los bloques residen temporalmente se denomina **memoria intermedia de disco**.

Las transferencias de un bloque entre disco y memoria principal se comienzan a través de las dos operaciones siguientes:

1. entrada(B) transfiere el bloque físico B a la memoria principal.
2. salida(B) transfiere el bloque de memoria intermedia B al disco y reemplaza allí al correspondiente bloque físico.

Este esquema se ilustra en la Figura 17.1.

Cada transacción T_i posee un área de trabajo privada en la cual se guardan copias de todos los elementos de datos a los que ha accedido y actualizado. Este área de trabajo se crea cuando se comienza una transacción y se elimina cuando la transacción o bien se compromete o bien aborta. Cada elemento de datos X almacenado en el área de trabajo de la transacción T_i se denotará como x_i. La transacción T_i interactúa con el sistema de bases de datos por medio de transferencias de datos desde su área de trabajo hacia la memoria intermedia del sistema y viceversa. Nosotros realizaremos transferencias de datos utilizando las dos operaciones siguientes:

1. leer(X) asigna el valor del elemento de datos X a la variable local x_i. Esta operación se ejecuta como sigue:
   a. Si el bloque B_X en el que reside X no está en la memoria principal, entonces se ejecuta entrada(B_X).
   b. Asigna a x_i el valor de X en el bloque de memoria intermedia.
2. escribir(X) asigna el valor de la variable local x_i al elemento de datos X en el bloque de memoria intermedia. Esta operación se ejecuta como sigue:
   a. Si el bloque B_X en el que reside X no está en la memoria principal, entonces se ejecuta entrada(B_X).
   b. Asigna el valor de x_i a X en la memoria intermedia B_X.

Obsérvese que ambas operaciones pueden requerir la transferencia de un bloque desde disco a la memoria principal. En cambio, ninguna de ellas requiere específicamente la transferencia de un bloque desde la memoria principal al disco.

El bloque de memoria intermedia B se escribe finalmente en el disco bien porque el gestor de la memoria intermedia necesita espacio en memoria para otros propósitos, o bien porque el sistema de base de datos desea reflejar en el disco los cambios sufridos por B. Se dice que el sistema de bases de datos **fuerza la salida** de la memoria intermedia B si ejecuta el comando salida(B).

Cuando una transacción necesita acceder a un elemento de datos X por primera vez, debe ejecutar leer(X). Todas las actualizaciones de X se llevan a cabo sobre x_i. Después de que la transacción acceda a X por última vez, se debe ejecutar escribir(X) para reflejar en la propia base de datos los cambios sufridos por X.

La operación salida(B_X) sobre el bloque de memoria intermedia B_X en el que reside X no tiene por qué tener efecto inmediatamente después de ejecutar escribir(X), ya que el bloque B_X puede contener otros elementos de datos que a los que aún se esté accediendo. Así, la salida real tiene lugar más tarde. Obsérvese que, si el sistema se bloquea después de ejecutar la operación escribir(X), pero antes de ejecutar salida(B_X), no se escribe nunca en el disco el nuevo valor de X y, por tanto, se pierde.

17.3 Recuperación y atomicidad

Considérese de nuevo el sistema bancario simplificado y una transacción T_i que transfiere 50 € desde la cuenta A a la cuenta B, siendo los saldos iniciales de A y de B de 1.000 € y de 2.000 €, respectivamente. Supóngase que el sistema cae durante la ejecución de T_i después de haberse ejecutado salida(B_A), pero antes de la ejecución de salida(B_B), donde B_A y B_B denotan los bloques de memoria intermedia en los que residen B_A y B_B. Al perderse el contenido de la memoria no se sabe la suerte de la transacción; así, podríamos invocar uno de los dos procedimientos posibles de recuperación.

- **Volver a ejecutar** T_i. Este procedimiento hará que el saldo de A se quede en 900 € en vez de en 950 €. De este modo el sistema entra en un estado inconsistente.
- **No volver a ejecutar** T_i. El estado actual del sistema otorga los valores de 950 € y 2.000 € para A y B respectivamente. Por tanto, el sistema entra en un estado inconsistente.

En cualquier caso se deja a la base de datos en un estado inconsistente y, por tanto, este esquema simple de recuperación de datos no funciona. El motivo de este mal funcionamiento es que se ha modificado la base de datos sin tener la seguridad de que la transacción se comprometa realmente. El objetivo es realizar todos los cambios inducidos por T_i o no llevar a cabo ninguno. Sin embargo, si T_i realiza varias

modificaciones en la base de datos, pueden necesitarse varias operaciones de salida y puede ocurrir un fallo después de haberse concluido alguna de estas modificaciones, pero antes de haber terminado todas.

Para conseguir el objetivo de la atomicidad se debe efectuar primero la operación de salida de la información que describe las modificaciones en el almacenamiento estable sin modificar todavía la base de datos. Como se verá, este procedimiento permitirá realizar la salida de todas las modificaciones realizadas por una transacción comprometida aunque se produzcan fallos. Supondremos que *las transacciones se ejecutan secuencialmente*; es decir, sólo una transacción está activa en cada momento. Se describirá la forma de manejar la ejecución concurrente de transacciones más adelante, en el Apartado 17.5.

17.4 Recuperación basada en el registro histórico

La estructura más ampliamente utilizada para guardar las modificaciones de una base de datos es el **registro histórico**. El registro histórico es una secuencia de **registros** que almacena todas las actividades de actualización de la base de datos. Existen varios tipos de registros del registro histórico. Un **registro de actualización del registro histórico** describe una única escritura en la base de datos y tiene los siguientes campos:

- El **identificador de la transacción** es un identificador único de la transacción que realiza la operación escribir.
- El **identificador del elemento de datos** es un identificador único del elemento de datos que se escribe. Normalmente suele coincidir con la ubicación del elemento de datos en el disco.
- El **valor anterior** es el valor que tenía el elemento de datos antes de la escritura.
- El **valor nuevo** es el valor que tendrá el elemento de datos después de la escritura.

Existen otros registros del registro histórico especiales para registrar sucesos significativos durante el procesamiento de una transacción, tales como el comienzo de una transacción y el éxito o aborto de la misma. Denotaremos como sigue los diferentes tipos de registros del registro histórico:

- $<T_i$ iniciada$>$. La transacción T_i ha comenzado.
- $<T_i,\ X_j,\ V_1,\ V_2>$. La transacción T_i ha realizado una escritura sobre el elemento de datos X_j. X_j tenía el valor V_1 antes de la escritura y tendrá el valor V_2 después de la escritura.
- $<T_i$ comprometida$>$. La transacción T_i se ha comprometido.
- $<T_i$ abortada$>$. La transacción T_i ha sido abortada.

Cuando una transacción realiza una escritura es fundamental que se cree el registro del registro histórico correspondiente a esa escritura antes de modificar la base de datos. Una vez que el registro del registro histórico existe, se puede realizar la salida de la modificación a la base de datos si se desea. Además, es posible *deshacer* una modificación que ya haya salido a la base de datos. Se deshará utilizando el campo valor-anterior de los registros del registro histórico.

Para que los registros del registro histórico sean útiles para recuperarse frente a errores del disco o del sistema, el registro histórico debe residir en almacenamiento estable. Por ahora supóngase que cada registro del registro histórico se escribe, tan pronto como se crea, al final del registro histórico en almacenamiento estable. En el Apartado 17.7 se verán las condiciones necesarias para poder relajar este requisito de forma segura de modo que se reduzca la sobrecarga impuesta por el registro histórico. En los Apartados 17.4.1 y 17.4.2 se presentarán dos técnicas de utilización del registro histórico para garantizar la atomicidad frente a fallos. Obsérvese que en el registro histórico se tiene constancia de todas las actividades de la base de datos. Como consecuencia, el tamaño de los datos almacenados en el registro histórico puede llegar a ser extremadamente grande. En el Apartado 17.4.3 se mostrará bajo qué condiciones se puede borrar información del registro histórico de manera segura.

<T_0 iniciada>
<T_0, A, 950>
<T_0, B, 2050>
<T_0 comprometida>
<T_1 iniciada>
<T_1, C, 600>
<T_1 comprometida>

Figura 17.2 Fragmento del registro histórico de la base de datos correspondiente a T_0 y T_1.

17.4.1 Modificación diferida de la base de datos

La **técnica de modificación diferida** garantiza la atomicidad de las transacciones mediante el almacenamiento de todas las modificaciones de la base de datos en el registro histórico, pero retardando la ejecución de todas las operaciones escribir de una transacción hasta que la transacción se compromete parcialmente. Recuérdese que se dice que una transacción se compromete parcialmente una vez que se ejecuta la acción final de la transacción. En la versión de la técnica de modificación diferida que se describe en este apartado se supone que las transacciones se ejecutan secuencialmente.

Cuando una transacción se compromete parcialmente, la información del registro histórico asociada a esa transacción se utiliza para la ejecución de las escrituras diferidas. Si el sistema cae antes de que la transacción complete su ejecución o si la transacción aborta, la información del registro histórico simplemente se ignora.

La ejecución de una transacción T_i opera de esta manera: antes de que T_i comience su ejecución se escribe en el registro histórico un registro <T_i iniciada>. La operación escribir(X) realizada por T_i se traduce en la escritura de un nuevo registro en el registro histórico. Finalmente, cuando T_i se ha comprometido parcialmente, se escribe en el registro histórico un registro <T_i comprometida>.

Cuando T_i se compromete parcialmente, los registros asociados a ella en el registro histórico se utilizan para la ejecución de las escrituras diferidas. Como puede ocurrir un fallo mientras se lleva a cabo esta actualización, hay que asegurarse de que, antes del comienzo de estas actualizaciones, todos los registros del registro histórico se guardan en almacenamiento estable. Una vez que se ha hecho esto, la actualización real tiene lugar y la transacción pasa al estado comprometido.

Obsérvese que la técnica de modificación diferida sólo requiere el nuevo valor de los elementos de datos. Así, se puede simplificar la estructura general de los registros de actualización del registro histórico que se vieron en el apartado anterior omitiendo el campo para el valor anterior.

Para ilustrar esto reconsidérese el sistema bancario simplificado. Sea T_0 una transacción que transfiere 50 € desde la cuenta A a la cuenta B:

T_0: leer(A);
$A := A - 50$;
escribir(A);
leer(B);
$B := B + 50$;
escribir(B).

Sea T_1 una transacción que retira 100 € de la cuenta C:

T_1: leer(C);
$C := C - 100$;
escribir(C).

Supóngase que estas transacciones se ejecutan secuencialmente, primero T_0 y después T_1 , y que los saldos de las cuentas A, B y C antes de producirse la ejecución eran 1.000, 2.000 y 700 € respectivamente. El fragmento del registro histórico que contiene la información relevante sobre estas dos transacciones se muestra en la Figura 17.2.

Registro histórico	Base de datos
<T_0 iniciada>	
<T_0, A, 950>	
<T_0, B, 2050>	
<T_0 comprometida>	
	A = 950
	B = 2050
<T_1 iniciada>	
<T_1, C, 600>	
<T_1 comprometida>	
	C = 600

Figura 17.3 Estado del registro histórico y de la base de datos correspondiente a T_0 y T_1.

Las salidas reales que se producen en el sistema de bases de datos y en el registro histórico como consecuencia de la ejecución de T_0 y T_1 pueden seguir distintas ordenaciones. Una ordenación posible se presenta en la Figura 17.3. Obsérvese que el valor de A se cambia en la base de datos sólo después de que el registro $<T_0,\ A,\ 950>$ se haya introducido en el registro histórico.

Mediante la utilización del registro histórico, el sistema puede manejar cualquier fallo que conduzca a la pérdida de información en el almacenamiento volátil. El esquema de recuperación usa el siguiente procedimiento de recuperación:

- rehacer(T_i) define el valor de todos los elementos de datos actualizados por la transacción T_i como los valores nuevos.

El conjunto de elementos de datos actualizados por T_i y sus respectivos nuevos valores se encuentran en el registro histórico.

La operación rehacer debe ser **idempotente**, esto es, el resultado de ejecutarla varias veces debe ser equivalente al resultado de ejecutarla una sola vez. Esta característica es fundamental para garantizar un correcto comportamiento incluso si el fallo se produce durante el proceso de recuperación.

Después de ocurrir un fallo, el subsistema de recuperación consulta el registro histórico para determinar las transacciones que deben rehacerse. Una transacción T_i debe rehacerse si y sólo si el registro histórico contiene los registros $<T_i$ iniciada$>$ y $<T_i$ comprometida$>$. Así, si el sistema cae después de que la transacción complete su ejecución, la información en el registro histórico se utiliza para restituir el sistema a un estado consistente anterior.

Para ilustrar esto volvamos a considerar el ejemplo bancario con la ejecución ordenada de las transacciones T_0 y T_1, primero T_0 y después T_1. En la Figura 17.2 se muestra el registro histórico que resulta de la ejecución completa de T_0 y T_1. Supóngase que el sistema cae antes de completarse las transacciones para poder ver el modo en que la técnica de recuperación lleva a la base de datos a un estado consistente. Supóngase que la caída ocurre justo después de haber escrito en almacenamiento estable el registro del registro histórico para el paso

escribir(B)

de la transacción T_0. El contenido del registro histórico en el momento de la caída puede verse en la Figura 17.4a. Cuando el sistema vuelve a funcionar no es necesario llevar a cabo ninguna acción rehacer ya que no aparece ningún registro de compromiso en el registro histórico. Los saldos de las cuentas A y B siguen siendo de 1.000 y 2.000 € respectivamente. Pueden borrarse del registro histórico los registros de la transacción incompleta T_0.

Supóngase ahora que la caída sucede justo después de haber escrito en almacenamiento estable el registro del registro histórico para el paso

escribir(C)

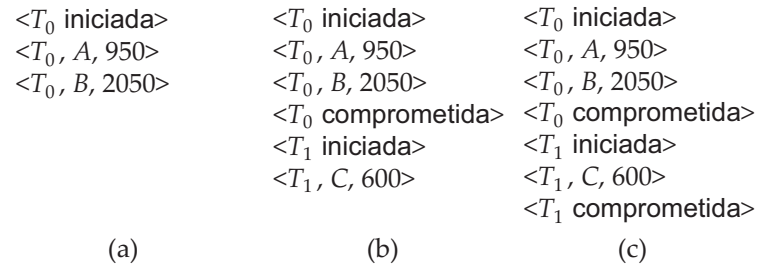


Figura 17.4 El mismo registro histórico que el de la Figura 17.3 en tres momentos distintos.

de la transacción T_1. En este caso, el contenido del registro histórico en el momento de la caída puede verse en la Figura 17.4b. Cuando el sistema vuelve a funcionar se realiza la operación rehacer(T_0) ya que el registro

$$<T_0 \text{ comprometida}>$$

aparece en el registro histórico en el disco. Después de la ejecución de esta operación, los saldos de las cuentas *A* y *B* son de 950 y 2.050 € respectivamente. El saldo de la cuenta *C* se mantiene en 700 €. Igual que antes, pueden borrarse del registro histórico los registros de la transacción incompleta T_1.

Por último, supóngase que la caída ocurre justo después de haber escrito en almacenamiento estable el registro del registro histórico

$$<T_1 \text{ comprometida}>$$

El contenido del registro histórico en el momento de la caída se muestra en la Figura 17.4c. Cuando el sistema vuelve a funcionar, hay dos registros comprometida en el registro: uno para T_0 y otro para T_1. Así pues, deben realizarse las operaciones rehacer(T_0) y rehacer(T_1). Después de la ejecución de estas operaciones, los saldos de las cuentas *A*, *B* y *C* son de 950, 2.050 y 600 €, respectivamente.

Considérese, finalmente, un caso en el que tiene lugar una segunda caída del sistema durante la recuperación de la primera. Deben hacerse algunos cambios en la base de datos como consecuencia de la ejecución de las operaciones rehacer, pero no se han realizado todos los cambios. Cuando el sistema vuelve a funcionar después de la segunda caída, la recuperación procede exactamente igual que en los ejemplos anteriores. Para cada registro

$$<T_i \text{ comprometida}>$$

que se encuentre en el registro histórico, se ejecuta la operación rehacer(T_i). En otras palabras, las acciones de recuperación se vuelven a reanudar desde el principio. Como rehacer escribe los valores en la base de datos independientemente de los datos que haya actualmente en la base de datos, el resultado de un segundo intento acabado con éxito en la ejecución de rehacer es el mismo que si rehacer hubiera acabado con éxito la primera vez.

17.4.2 Modificación inmediata de la base de datos

La **técnica de modificación inmediata** permite realizar la salida de las modificaciones de la base de datos a la propia base de datos mientras que la transacción está todavía en estado activo. Las modificaciones de datos escritas por transacciones activas se denominan **modificaciones no comprometidas**. En caso de una caída o de un fallo en la transacción, el sistema debe utilizar el campo para el valor anterior de los registros del registro histórico descritos en el Apartado 17.4 para restaurar los elementos de datos modificados a los valores que tuvieran antes de comenzar la transacción. Esta restauración se lleva a cabo mediante la operación deshacer descrita a continuación.

Antes de comenzar la ejecución de una transacción T_i, se escribe en el registro histórico el registro $<T_i$ iniciada$>$. Durante su ejecución, cualquier operación escribir(X) realizada por T_i, es *precedida* por la escritura en el registro histórico de un registro actualizado apropiado. Cuando T_i se compromete parcialmente se escribe en el registro histórico el registro $<T_i$ comprometida$>$.

<T_0 iniciada>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 comprometida>
<T_1 iniciada>
<T_1, C, 700, 600>
<T_1 comprometida>

Figura 17.5 Fragmento del registro histórico del sistema correspondiente a T_0 y a T_1.

Como la información del registro histórico se utiliza para reconstruir el estado de la base de datos, la actualización real de la base de datos no puede permitirse antes de que el registro del registro histórico correspondiente se haya escrito en almacenamiento estable. Por lo tanto, es necesario que antes de la ejecución de una salida(B), se escriban en almacenamiento estable los registros del registro histórico correspondientes a B. Esto volverá a tratarse en el Apartado 17.7.

Para ilustrarlo considérese de nuevo el sistema bancario simplificado con la ejecución ordenada de las transacciones T_0 y T_1, primero T_0 y después T_1. Las líneas del registro histórico que contienen la información relevante concerniente a estas dos transacciones se muestran en la Figura 17.5.

En la Figura 17.6 se describe una posible ordenación de las salidas reales que se producen en el sistema de bases de datos y en el registro histórico como consecuencia de la ejecución de T_0 y T_1. Obsérvese que esta ordenación no podría obtenerse con la técnica de modificación diferida que se vio en el Apartado 17.4.1.

Mediante la utilización del registro histórico, el sistema puede manejar cualquier fallo que no genere una pérdida de información en el almacenamiento no volátil. El esquema de recuperación usa dos procedimientos de recuperación:

- deshacer(T_i) restaura el valor de todos los elementos de datos actualizados por la transacción T_i a los valores anteriores.
- rehacer(T_i) cambia el valor de todos los elementos de datos actualizados por la transacción T_i a los nuevos valores.

El conjunto de elementos de datos actualizados por T_i y sus respectivos anteriores y nuevos valores se encuentran en el registro histórico.

Las operaciones deshacer y rehacer deben ser idempotentes para garantizar un comportamiento correcto incluso en el caso de que el fallo se produzca durante el proceso de recuperación.

Después de haberse producido un fallo, el esquema de recuperación consulta el registro histórico para determinar las transacciones que deben rehacerse y las que deben deshacerse.

Registro histórico	**Base de datos**
<T_0 iniciada>	
<T_0, A, 1000, 950>	
<T_0, B, 2000, 2050>	
<T_0 comprometida>	
	A = 950
	B = 2050
<T_1 iniciada>	
<T_1, C, 700, 600>	
	C = 600
<T_1 comprometida>	

Figura 17.6 Estado del registro histórico y de la base de datos correspondientes a T_0 y a T_1.

(a)	(b)	(c)
<T_0 iniciada>	<T_0 iniciada>	<T_0 iniciada>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 comprometida>	<T_0 comprometida>
	<T_1 iniciada>	<T_1 iniciada>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 comprometida>

Figura 17.7 El mismo registro histórico mostrado en tres momentos distintos.

- Una transacción T_i debe deshacerse si el registro histórico contiene el registro <T_i iniciada>, pero no contiene el registro <T_i comprometida>.
- Una transacción T_i debe rehacerse si el registro histórico contiene los registros <T_i iniciada> y <T_i comprometida>.

Para ilustrarlo, considérese de nuevo el sistema bancario con la ejecución ordenada de las transacciones T_0 y T_1, primero T_0 y después T_1. Supóngase que el sistema cae antes de completarse las transacciones. Se considerarán tres casos. El estado del registro histórico para cada uno de ellos se muestra en la Figura 17.7.

En primer lugar supóngase que la caída ocurre justo después de haber escrito en almacenamiento estable el registro del registro histórico para el paso

escribir(B)

de la transacción T_0 (Figura 17.7a). Cuando el sistema vuelve a funcionar encuentra en el registro histórico el registro <T_0 iniciada>, pero no su correspondiente <T_0 comprometida>. Por lo tanto, la transacción T_0 debe deshacerse y se ejecutaría deshacer(T_0). Como resultado de esta operación, los saldos de las cuentas A y B (en el disco) se restituirían a 1.000 y 2.000 € respectivamente.

Supóngase ahora que la caída sucede justo después de haber escrito en almacenamiento estable el registro del registro histórico para el paso

escribir(C)

de la transacción T_1 (Figura 17.7b). Cuando el sistema vuelve a funcionar, es necesario llevar a cabo dos acciones de recuperación. La operación deshacer(T_1) debe ejecutarse porque en el registro histórico aparece el registro <T_1 iniciada>, pero no aparece <T_1 comprometida>. La operación rehacer(T_0) debe ejecutarse porque el registro histórico contiene los registros <T_0 iniciada> y <T_0 comprometida>. Al final del procedimiento de recuperación, los saldos de las cuentas A, B y C son de 950, 2.050 y 700 € respectivamente. Obsérvese que la operación deshacer(T_1) se ejecuta antes que rehacer(T_0). En este ejemplo, el resultado sería el mismo si se cambiara el orden. Sin embargo, el hecho de realizar primero las operaciones deshacer y luego las operaciones rehacer es importante para el algoritmo de recuperación que se describe en el Apartado 17.5.

Por último, supóngase que la caída tiene lugar justo después de haber escrito en almacenamiento estable el registro del registro histórico

<T_1 comprometida>

(Figura 17.7c). Cuando el sistema vuelve a funcionar, deben rehacerse tanto T_0 como T_1 ya que se encuentran en el registro histórico los registros <T_0 iniciada> y <T_0 comprometida>, así como los registros <T_1 iniciada> y <T_1 comprometida>. Los saldos de las cuentas A, B y C después de la ejecución de los procedimientos de recuperación rehacer(T_0) y rehacer(T_1) se sitúan en 950, 2.050 y 600 € respectivamente.

17.4.3 Puntos de revisión

Cuando ocurre un fallo en el sistema se debe consultar el registro histórico para determinar las transacciones que deben rehacerse y las que deben deshacerse. En principio es necesario recorrer completamente el registro histórico para hallar esta información. En este enfoque hay dos inconvenientes principales:

1. El proceso de búsqueda consume tiempo.
2. La mayoría de las transacciones que deben rehacerse de acuerdo con el algoritmo ya tienen escritas sus actualizaciones en la base de datos. Aunque el hecho de volver a ejecutar estas transacciones no produzca resultados erróneos, sí repercutirá en un aumento del tiempo de ejecución del proceso de recuperación.

Para reducir este tipo de sobrecarga se introducen los puntos de revisión. Durante la ejecución, el sistema actualiza el registro histórico utilizando una de las dos técnicas que se describen en los Apartados 17.4.1 y 17.4.2. Además, el sistema realiza periódicamente **puntos de revisión**, en los cuales tiene lugar la siguiente secuencia de acciones:

1. Escritura en almacenamiento estable de todos los registros del registro histórico que residan en ese momento en memoria principal.
2. Escritura en disco de todos los bloques de memoria intermedia que se hayan modificado.
3. Escritura en almacenamiento estable de un registro del registro histórico <revisión>.

Mientras se lleva a cabo un punto de revisión no se permite que ninguna transacción realice acciones de actualización, tales como escribir en un bloque de memoria intermedia o escribir un registro del registro histórico.

La presencia de un registro <revisión> en el registro histórico permite que el sistema pueda hacer más eficiente su procedimiento de recuperación. Considérese una transacción T_i que se comprometió antes del punto de revisión. Para esa transacción el registro $<T_i$ comprometida> aparece en el registro histórico antes que el registro <revisión>. Todas las modificaciones sobre la base de datos hechas por T_i se deben haber escrito en la base de datos antes del punto de revisión o formando parte del propio punto de revisión. Así, en el momento de la recuperación, no es necesario ejecutar una operación rehacer sobre T_i.

Esta observación permite perfeccionar los esquemas anteriores de recuperación (sigue siendo válido el supuesto de que las transacciones se ejecutan secuencialmente). Cuando se produce un fallo, el esquema de recuperación examina el registro histórico para determinar la última transacción T_i que comenzó su ejecución antes de que tuviera lugar el último punto de revisión. Para encontrar una transacción de este tipo se recorre el registro histórico hacia atrás, esto es, se empieza a buscar por el final del registro histórico hasta que se encuentra el primer registro <revisión> (como se va recorriendo el registro histórico hacia atrás, el registro encontrado corresponde al último registro <revisión> del registro histórico); después se continúa la búsqueda hacia atrás hasta que se encuentra el siguiente registro $<T_i$ iniciada>. Este registro identifica a la transacción T_i.

Una vez que ha sido identificada la transacción T_i sólo es necesario aplicar las operaciones rehacer y deshacer a la transacción T_i y a las transacciones T_j que comenzaron su ejecución después que T_i. Sea T este conjunto de transacciones. Puede ignorarse el resto del registro histórico (la parte del principio) y puede borrarse cuando se desee. El conjunto exacto de operaciones de recuperación que han de llevarse a cabo depende de si se está usando la técnica de modificación inmediata o la de modificación diferida. Si se emplea la técnica de modificación inmediata, las operaciones de recuperación deben ser las siguientes:

- Ejecutar deshacer(T_k) para todas las transacciones T_k de T para las que no exista un registro $<T_k$ comprometida> en el registro histórico.
- Ejecutar execute rehacer(T_k) para todas las transacciones T_k de T para las que aparece un registro $<T_k$ comprometida> en el registro histórico.

Obviamente no es necesario aplicar la operación deshacer cuando se está utilizando la técnica de modificación diferida.

A modo de ejemplo, considérese el conjunto de transacciones $\{T_0, T_1, \ldots, T_{100}\}$ de modo que su ejecución se produce en el orden determinado por los subíndices. Supóngase que el último punto de revisión tiene lugar durante la ejecución de la transacción T_{67}. Así, durante el esquema de recuperación, sólo deben considerarse las transacciones T_{67}, $T_{68}, \ldots, T_{100}$. Será necesario rehacer cada una de ellas si éstas se comprometieron y será necesario deshacerlas en caso contrario.

En el Apartado 17.5.3 se estudia una extensión de la técnica de puntos de revisión para el procesamiento concurrente de transacciones.

17.5 Transacciones concurrentes y recuperación

Hasta ahora se ha tratado la recuperación en un entorno en el que se ejecutaba una sola transacción en cada instante. Ahora se verá cómo modificar y extender el esquema de recuperación basado en registro histórico para permitir la ejecución concurrente de múltiples transacciones. El sistema sigue teniendo una única memoria intermedia de disco y un único registro histórico independientemente del número de transacciones concurrentes. Todas las transacciones comparten los bloques de la memoria intermedia. Se permiten actualizaciones inmediatas y que un bloque de la memoria intermedia tenga elementos de datos que hayan sido modificados por una o más transacciones.

17.5.1 Interacción con el control de concurrencia

El esquema recuperación depende en gran medida del esquema de control de concurrencia que se use. Para hacer retroceder los efectos de una transacción fallida deben deshacerse las modificaciones realizadas por esa transacción. Supóngase que se debe retroceder una transacción T_0 y que un dato Q, que fue modificado por T_0, tiene que recuperar su antiguo valor. Si se está usando un esquema basado en registro histórico para la recuperación, es posible restablecer el valor de Q utilizando la información contenida en el registro histórico. Supóngase ahora que una segunda transacción T_1 realiza una nueva modificación sobre Q *antes* de retroceder T_0. En este caso, al retroceder T_0, se perdería la modificación realizada por T_1.

Es necesario, por tanto, que si una transacción T modifica el valor de un elemento de datos Q, ninguna otra transacción pueda modificar el mismo elemento de datos hasta que T se haya comprometido o haya sido retrocedida. Este requisito puede satisfacerse fácilmente utilizando bloqueo estricto de dos fases—esto es, bloqueo de dos fases que mantiene los bloqueos exclusivos hasta el final de la transacción.

17.5.2 Retroceso de transacciones

Se utiliza el registro histórico para retroceder una transacción T_i fallida. El registro histórico se explora hacia atrás; para cada registro del registro histórico de la forma $<T_i,\ X_j,\ V_1,\ V_2>$, se restablece el valor del elemento de datos X_j con su valor anterior: V_1. La exploración del registro histórico termina cuando se encuentra el registro $<T_i$, iniciada$>$.

Es importante el hecho de recorrer el registro histórico empezando por el final, ya que una transacción puede haber actualizado más de una vez el valor de un elemento de datos. Como ejemplo, considérese este par de registros:

$$<T_i,\ A,\ 10,\ 20>$$
$$<T_i,\ A,\ 20,\ 30>$$

Estos registros del registro histórico representan una modificación del elemento de datos A por parte de la transacción T_i, seguida de otra modificación de A hecha también por T_i. Al recorrer el registro histórico al revés se establece correctamente el valor de A como 10. Si el registro histórico se recorriera hacia delante, A tomaría como valor 20, lo cual es incorrecto.

Si para el control de concurrencia se utiliza el bloqueo estricto de dos fases, los bloqueos llevados a cabo por una transacción T sólo pueden ser desbloqueados después de que la transacción haya retrocedido según se acaba de describir. Una vez que T (que está retrocediendo) haya actualizado un elemento

de datos, ninguna otra transacción podría haber actualizado el mismo elemento de datos debido a los requisitos del control de concurrencia que se mencionaron en el Apartado 17.5.1. Así pues, la restitución del valor anterior de un elemento de datos no borrará los efectos de otra transacción.

17.5.3 Puntos de revisión

En el Apartado 17.4.3 se usaban los puntos de revisión para reducir el número de registros del registro histórico que debían ser examinados cuando el sistema se recuperaba de una caída. Como se asumía que no existía la concurrencia, durante la recuperación era necesario considerar solamente las siguientes transacciones:

- Las transacciones que comenzaron después del último punto de revisión.
- La única transacción, si la había, que estaba activa en el momento de grabarse el último punto de revisión.

Cuando las transacciones pueden ejecutarse concurrentemente, la situación se torna más complicada ya que varias transacciones pueden estar activas en el momento en que se produce el último punto de revisión.

En un sistema de procesamiento de transacciones concurrente es necesario que el registro del registro histórico correspondiente a un punto de revisión sea de la forma <revisión L>, donde L es una lista con las transacciones activas en el momento del punto de revisión. De nuevo se supone que, mientras que se realiza el punto de revisión, las transacciones no efectúan modificaciones ni sobre los bloques de la memoria intermedia ni sobre el registro histórico.

El requisito de que las transacciones no puedan realizar modificaciones sobre los bloques de la memoria intermedia ni sobre el registro histórico durante un punto de revisión puede resultar molesto ya que el procesamiento de transacciones tendrá que parar durante la ejecución de un punto de revisión. Un punto de revisión durante el cual se permite que las transacciones realicen modificaciones incluso mientras los bloques de memoria intermedia se están guardando en disco, se denomina **punto de revisión difuso**. En el Apartado 17.8.5 se describen esquemas de revisión difusa.

17.5.4 Recuperación al reiniciar

El sistema construye dos listas cuando se recupera de una caída: la lista-deshacer, que consta de las transacciones que han de deshacerse, y la lista-rehacer, que está formada por las transacciones que deben rehacerse.

Estas dos listas se construyen durante la recuperación de la siguiente manera. Al principio ambas están vacías. Luego se recorre el registro histórico hacia atrás examinando cada registro hasta que se encuentra el primer registro <revisión>:

- Para cada registro encontrado de la forma <T_i comprometida>, se añade T_i a la lista-rehacer.
- Para cada registro encontrado de la forma <T_i iniciada>, si T_i no está en la lista-rehacer, entonces se añade T_i a la lista-deshacer.

Una vez que se han examinado los registros apropiados del registro histórico, se atiende al contenido de la lista L en el registro punto de revisión. Para cada transacción T_i en L, si T_i no está en la lista-rehacer, entonces se añade T_i a la lista-deshacer.

Cuando se terminan la lista-rehacer y la lista-deshacer, el proceso de recuperación procede de la siguiente manera:

1. Se recorre de nuevo el registro histórico hacia atrás comenzando en el último registro y se realiza una operación deshacer por cada registro del registro histórico que pertenezca a una transacción T_i de la lista-deshacer. En esta fase se ignoran los registros del registro histórico concernientes a transacciones de la lista-rehacer. El recorrido del registro histórico termina cuando se encuentran registros <T_i iniciada> para cada transacción T_i de la lista-deshacer.

2. Se localiza el último registro <revisión L> del registro histórico. Obsérvese que este paso puede necesitar un recorrido del registro histórico hacia delante si el registro del punto de revisión quedara atrás en el paso 1.
3. Se recorre el registro histórico hacia delante desde el último registro <revisión L> y se realiza una operación rehacer por cada registro del registro histórico que pertenezca a una transacción T_i de la lista-rehacer. En esta fase se ignoran los registros del registro histórico concernientes a transacciones de la lista-deshacer.

Es importante procesar el registro histórico hacia atrás en el paso 1 para garantizar que el estado resultante de la base de datos sea correcto.

Después de haber deshecho todas las transacciones de la lista-deshacer se rehacen aquellas transacciones que pertenezcan a la lista-rehacer. En este caso es importante procesar el registro histórico hacia delante. Cuando se ha completado el proceso de recuperación, se continúa con el procesamiento normal de las transacciones.

Es importante el hecho de deshacer las transacciones de la lista-deshacer antes de rehacer las transacciones de la lista-rehacer al utilizar los pasos 1 a 3 del algoritmo anterior. El siguiente problema podría ocurrir de no hacerse así. Supóngase que el elemento de datos A vale inicialmente 10. Supóngase también que una transacción T_i modifica el valor de A situándolo en 20 y aborta a continuación; el retroceso de la transacción devolvería a A el valor 10. Supóngase que otra transacción T_j cambia entonces a 30 el valor de A, se compromete y, seguidamente, el sistema cae. El estado del registro histórico en el momento de la caída es:

$$<T_i,\ A,\ 10,\ 20>$$
$$<T_j,\ A,\ 10,\ 30>$$
$$<T_j \text{ comprometida}>$$

Si se rehace primero, A tomará el valor 30; y luego, al deshacer, A acabará valiendo 10, lo cual es incorrecto. El valor final de Q debe ser 30, lo que puede garantizarse si se deshace antes de rehacer.

17.6 Gestión de la memoria intermedia

En este apartado se consideran varios detalles sutiles que son esenciales para la implementación de un esquema de recuperación que garantice la consistencia de los datos y que lleve asociado una sobrecarga mínima respecto a la interacción con la base de datos.

17.6.1 Registro histórico con memoria intermedia

Hasta aquí se ha supuesto que cada registro del registro histórico en almacenamiento estable se escribe en el mismo momento de su creación. Esta suposición impone una sobrecarga muy alta en la ejecución del sistema por las siguientes razones. Habitualmente, la unidad de escritura en almacenamiento estable es el bloque. En la mayoría de los casos un registro del registro histórico es mucho más pequeño que un bloque. Así, la escritura de cada registro del registro histórico se traduce en una escritura mucho mayor en el nivel físico. Además de esto, como se vio en el Apartado 17.2.2, la escritura de un bloque en almacenamiento estable puede involucrar varias operaciones de escritura en el nivel físico.

El coste de realizar la escritura en almacenamiento estable de un bloque es suficientemente elevado para que sea deseable escribir de una sola vez varios registros del registro histórico. Para hacer esto se escriben los registros del registro histórico en una memoria intermedia almacenada en la memoria principal en la que permanecen durante un tiempo hasta que se guardan en almacenamiento estable. Se pueden acumular varios registros del registro histórico en la memoria intermedia del registro histórico y escribirse en almacenamiento estable con una sola operación. El orden de los registros del registro histórico en el almacenamiento estable debe ser exactamente el mismo orden en el que fueron escritos en la memoria intermedia del registro histórico.

Debido a la utilización de la memoria intermedia en el registro histórico, antes de ser escrito en almacenamiento estable, un registro del registro histórico puede permanecer únicamente en memoria principal (almacenamiento volátil) durante un espacio de tiempo considerable. Como esos registros se

perderían si el sistema cayese, es necesaria la imposición de nuevos requisitos sobre las técnicas de recuperación para garantizar la atomicidad de las transacciones:

- Después de que el registro $<T_i$ comprometida$>$ se haya escrito en almacenamiento estable, la transacción T_i pasa al estado comprometida.
- Antes de escribir en almacenamiento estable el registro $<T_i$ comprometida$>$, todos los registros del registro histórico pertenecientes a la transacción T_i se deben escribir en almacenamiento estable.
- Antes de que un bloque de datos en memoria principal se pueda escribir en la base de datos (en almacenamiento no volátil) todos los registros del registro histórico pertenecientes a los datos de ese bloque se deben escribir en almacenamiento estable.

  Este último requisito se denomina regla de **registro de escritura anticipada (REA)**. Estrictamente hablando, la regla REA sólo necesita que haya sido puesta en almacenamiento estable la información concerniente a la operación deshacer y permite que la información relativa a la operación rehacer pueda escribirse más tarde. La diferencia es relevante en aquellos sistemas en los que la información para rehacer y deshacer se guarda en registros del registro histórico independientes.

Las reglas anteriores representan situaciones en las que ciertos registros del registro histórico *deben* haber sido escritos en almacenamiento estable. No se produce ningún problema como resultado de la escritura de los registros del registro histórico *antes* de que sea necesaria. Así, cuando el sistema decide que es necesario escribir en almacenamiento estable un registro del registro histórico, puede escribir un bloque entero de ellos si hay suficientes registros en memoria principal como para llenar un bloque. Si no hay suficientes registros para llenar el bloque, se forma un bloque parcialmente lleno con todos los registros que hubiera en memoria principal y se ponen en almacenamiento estable.

La escritura en disco de la memoria intermedia del registro histórico se denomina a veces **forzar el registro histórico**.

17.6.2 Base de datos con memoria intermedia

En el Apartado 17.2 se describió el uso de una jerarquía de almacenamiento de dos niveles. La base de datos se almacena en almacenamiento no volátil (disco) y, cuando es necesario, se traen a memoria principal los bloques de datos que hagan falta. Como la memoria principal suele ser mucho más pequeña que la base de datos completa, puede ser necesaria la sobrescritura de un bloque B_1 en memoria principal cuando es necesario traer a memoria otro bloque B_2. Si B_1 ha sido modificado, B_1 se debe escribir antes de traer B_2. Como se estudió en el Apartado 11.5.1, en el Capítulo 11, esta jerarquía de almacenamiento se corresponde con el concepto usual de *memoria virtual*.

Las reglas que rigen la escritura de registros del registro histórico limitan la libertad del sistema para escribir bloques de datos. Si la lectura del bloque B_2 provoca que el bloque B_1 tenga que escribirse en almacenamiento estable, todos los registros del registro histórico pertenecientes a los datos de B_1 se deben escribir en almacenamiento estable antes de la escritura de B_1. Por tanto, la secuencia de acciones que ha de llevar a cabo el sistema sería ésta:

- Escritura en almacenamiento estable de los registros del registro histórico hasta que todos los registros pertenecientes al bloque B_1 se hayan escrito.
- Escritura en disco del bloque B_1.
- Lectura del bloque B_2 desde el disco a la memoria principal.

Es un hecho importante el que no se produzcan escrituras sobre el bloque B_1 mientras que se lleva a cabo la anterior secuencia de acciones. Esto puede garantizarse, como se explica a continuación, utilizando un medio de bloqueo especial. Antes de que una transacción realice una escritura sobre un elemento de datos debe adquirir un bloqueo en exclusiva sobre el bloque en el que reside el citado elemento de datos. Inmediatamente después de haber realizado la modificación, el bloqueo se puede liberar. Antes

de la escritura de un bloque, el sistema obtiene un bloqueo exclusivo sobre ese bloque para asegurarse de que ninguna transacción está modificándolo. Una vez que la escritura del bloque se ha completado, el bloqueo se libera. Los bloqueos de corta duración se denominan con frecuencia **pestillos**. Los pestillos y los bloqueos utilizados por el sistema de control de concurrencia se tratan de forma diferente. Como resultado, los pestillos pueden liberarse sin necesidad de cumplir ningún protocolo de bloqueo, como el bloqueo de dos fases requerido por el sistema de control de concurrencia.

Para ilustrar la necesidad del requisito de registro histórico con escritura anticipada considérese el ejemplo bancario con las transacciones T_0 y T_1. Supóngase que el estado del registro histórico es

$$<T_0 \text{ iniciada}>$$
$$<T_0,\ A,\ 1000,\ 950>$$

y que la transacción T_0 realiza una operación leer(B). Supóngase también que el bloque en el que se encuentra B no está en memoria principal y que esa memoria principal está llena. Supóngase por último que el bloque en el que reside A es el elegido para la sustitución y, por tanto, ha de escribirse en disco. Si el sistema escribe este bloque en disco y luego sucede una caída, los valores para las cuentas A, B y C en la base de datos son 950, 2.000 y 700 € respectivamente. Este estado de la base de datos es inconsistente. Sin embargo, según los requisitos de la regla REA, el registro del registro histórico

$$<T_0,\ A,\ 1000,\ 950>$$

debe escribirse en almacenamiento estable antes de producirse la escritura del bloque en el que se encuentra A. El sistema puede usar ese registro del registro histórico durante la recuperación para devolver a la base de datos a un estado consistente.

17.6.3 La función del sistema operativo en la gestión de la memoria intermedia

La memoria intermedia de la base de datos puede gestionarse usando uno de estos dos enfoques:

1. El sistema de base de datos reserva parte de la memoria principal para utilizarla como memoria intermedia y es él, en vez del sistema operativo, el que se encarga de gestionarlo. El sistema de base de datos gestiona la transferencia de los bloques de datos de acuerdo con los requisitos del Apartado 17.6.2.

   El inconveniente de este enfoque es que limita la flexibilidad en la utilización de la memoria principal. El tamaño de la memoria intermedia no debe ser muy grande para que otras aplicaciones tengan suficiente espacio disponible en la memoria principal para sus propias necesidades. Sin embargo, incluso cuando ninguna otra aplicación esté en ejecución, la base de datos no podrá hacer uso de toda la memoria disponible. Asimismo, aquellas aplicaciones que no tienen nada que ver con la base de datos no pueden usar la región de la memoria reservada para la memoria intermedia de la base de datos aunque no se estén utilizando algunas de las páginas almacenadas en la memoria intermedia.

2. El sistema de base de datos implementa su memoria intermedia dentro de la memoria virtual del sistema operativo. Como el sistema operativo conoce los requisitos de memoria de todos los procesos del sistema, es lógico que pueda decidir los bloques de la memoria intermedia que deben escribirse al disco y el momento en el que debe realizarse esta escritura. Pero para garantizar los requisitos del registro histórico de escritura anticipada del Apartado 17.6.1, el sistema operativo no debería realizar él mismo la escritura de las páginas de la base de datos de la memoria intermedia, sino que debería pedírselo al sistema de base de datos para que fuera éste el que forzara la escritura de los bloques de la memoria intermedia. El sistema de base de datos, después de escribir en almacenamiento estable los registros del registro histórico relevantes, forzaría la escritura en la base de datos de los bloques de la memoria intermedia. Desafortunadamente, casi todos los sistemas operativos de la generación actual ejercen un control completo sobre la memoria virtual. El sistema operativo reserva espacio en el disco para almacenar las páginas de memoria virtual que no se encuentran en ese momento en la memoria principal; este espacio se denomina **espacio de intercambio**. Si el sistema operativo decide escribir un bloque B_x, ese bloque se escribe

en el espacio de intercambio del disco por lo que el sistema de base de datos no tiene forma de controlar la escritura de los bloques de la memoria intermedia. Por consiguiente, si la memoria intermedia de la base de datos está en la memoria virtual, las transferencias entre los archivos de la base de datos y la memoria intermedia en memoria virtual deben estar gestionadas por el sistema de base de datos, hecho que subraya el cumplimiento de los requisitos del registro histórico de escritura anticipada que se vieron.

Este enfoque puede provocar una escritura adicional de datos en el disco. Si el sistema operativo realiza la escritura de un bloque B_x, éste no se escribe en la base de datos sino que se escribe en el espacio de intercambio que utiliza la memoria virtual del sistema operativo. Cuando la base de datos necesita escribir B_x, el sistema operativo puede necesitar primero leer B_x de su espacio de intercambio. Así, en lugar de realizar una sola escritura de B_x, son necesarias dos escrituras (una del sistema operativo y otra del sistema de base de datos) y una lectura adicional.

Aunque ambos enfoques tienen algunos inconvenientes, debe elegirse cualquiera de los dos excepto si el sistema operativo está diseñado para soportar los requisitos del registro histórico de base de datos. De los sistemas operativos actuales sólo unos pocos, como el sistema operativo Mach, soportan estos requisitos.

17.7 Fallo con pérdida de almacenamiento no volátil

Hasta ahora sólo se ha considerado el caso en el que un fallo conduce a la pérdida de información residente en almacenamiento volátil mientras que el contenido del almacenamiento no volátil permanecía intacto. A pesar de que es raro encontrarse con un fallo en el que se pierda información de almacenamiento no volátil, es necesario prepararse para afrontar este tipo de fallos. En este apartado se hablará sólo del almacenamiento en disco. La argumentación puede aplicarse también a otras clases de almacenamiento no volátil.

La idea básica es **volcar** periódicamente (digamos, una vez al día) el contenido entero de la base de datos en almacenamiento estable. Por ejemplo, puede volcarse la base de datos en una o más cintas magnéticas. Se utilizará el volcado más reciente para que la base de datos recupere un estado consistente cuando ocurra un fallo que conduzca a la pérdida de algunos bloques físicos de la base de datos. Una vez que se complete esta operación, el sistema utilizará el registro histórico para llevar al sistema de base de datos al último estado consistente en el que estuvo antes de producirse el fallo.

Más precisamente, ninguna transacción puede estar activa durante el procedimiento de volcado y tendrá lugar una secuencia de acciones similar a la utilizada en los puntos de revisión:

1. Escribir en almacenamiento estable todos los registros del registro histórico que residan en ese momento en memoria principal.
2. Escribir en disco todos los bloques de la memoria intermedia.
3. Copiar el contenido de la base de datos en almacenamiento estable.
4. Escribir el registro del registro histórico <volcar> en almacenamiento estable.

Los pasos 1, 2 y 4 se corresponden con los tres pasos utilizados para realizar un punto de revisión en el Apartado 17.4.3.

Para la recuperación por pérdida de almacenamiento no volátil se restituye la base de datos en el disco utilizando el último volcado realizado. Entonces se consulta el registro histórico y se rehacen todas las transacciones que se hubieran comprometido desde que se efectuó ese último volcado. Obsérvese que no es necesario ejecutar ninguna operación deshacer.

Un volcado del contenido de una base de datos se denomina también **volcado de archivo** ya que pueden archivarse los volcados y utilizarlos más tarde para examinar estados anteriores de la base de datos. Los volcados de una base de datos y la realización de puntos de revisión de las memorias intermedias son dos procesos análogos.

El procedimiento de volcado simple que se ha descrito anteriormente es costoso debido a las dos razones siguientes. En primer lugar, debe copiarse en almacenamiento estable la base de datos entera, lo

que conlleva una considerable transferencia de datos. En segundo lugar, se pierden ciclos de CPU porque se detiene el procesamiento de transacciones durante el procedimiento de volcado. Se han desarrollado esquemas de **volcado difuso** que permiten que las transacciones sigan activas mientras se realiza el volcado. Son esquemas similares a los de los puntos de revisión difusos. Para obtener más detalles véanse las notas bibliográficas.

17.8 Técnicas avanzadas de recuperación**

Las técnicas de recuperación descritas en el Apartado 17.5 requieren que, una vez que una transacción modifica un elemento de datos, ninguna otra pueda modificar el mismo elemento de datos hasta que la primera se comprometa o retroceda. Utilizando el bloqueo estricto de dos fases se garantiza esa condición. Aunque el bloqueo estricto de dos fases es aceptable para los registros en las relaciones, como se vio en el Apartado 16.9, provoca un decremento significativo en la concurrencia cuando se aplica sobre determinadas estructuras, como las páginas indexadas con árboles B^+.

Para incrementar la concurrencia puede usarse el algoritmo de control de concurrencia en árboles B^+ descrito en el Apartado 16.9 y así permitir que los bloqueos se liberen rápidamente, no con el procedimiento de dos fases. Como consecuencia, sin embargo, no serán aplicables las técnicas de recuperación del Apartado 17.5. Se han propuesto varias técnicas de recuperación alternativas que pueden aplicarse incluso con bloqueos de liberación rápida. Estos esquemas se pueden usar en varias aplicaciones, no sólo para recuperar árboles B^+. En primer lugar se describe un esquema de recuperación avanzado que soporta los bloqueos de liberación rápida. A continuación se describe el esquema de recuperación ARIES, que se usa ampliamente en la industria. ARIES es más complejo que el esquema de recuperación avanzado descrito anteriormente, pero incorpora varias optimizaciones para minimizar el tiempo de recuperación, y proporciona varias características útiles.

17.8.1 Registro de deshacer lógico

En las operaciones donde los bloqueos son de liberación rápida no pueden realizarse las operaciones deshacer escribiendo simplemente el valor anterior de los elementos de datos. Considérese una transacción T que inserta una entrada en un árbol B^+ y, siguiendo el protocolo de control de concurrencia con árboles B^+, libera algunos bloqueos después de completarse la operación inserción, pero antes de que la transacción se comprometa. Después de liberar los bloqueos, otras transacciones pueden realizar inserciones o borrados posteriores cambiando de este modo los nodos del árbol B^+.

Incluso aunque la operación libere rápidamente algunos bloqueos debe conservar suficientes bloqueos como para garantizar que ninguna otra transacción tenga permiso para ejecutar cualquier operación conflictiva (como leer o borrar el valor insertado). Por este motivo, el protocolo de control de concurrencia con árboles B^+ que se estudió anteriormente mantiene ciertos bloqueos en las hojas del árbol B^+ hasta el final de la transacción.

Considérese ahora cómo realizar retrocesos de transacciones. Si se usa una operación **deshacer física**, es decir, si durante el retroceso se escriben los valores anteriores de los nodos internos del árbol B^+ (antes de ejecutar la operación inserción), podrían perderse algunas de las modificaciones realizadas por inserciones o borrados ejecutados posteriormente por otras transacciones. La operación inserción no debe deshacerse así, sino con una operación **deshacer lógica**, esto es, mediante la ejecución de una operación de borrado en este caso.

Así, cuando finaliza la acción de inserción, antes de que libere ningún bloqueo, escribe en el registro histórico un registro $<T_i, O_j,$ fin-operación, $U>$, donde D denota la información para deshacer y O_j es un identificador único de la operación. Por ejemplo, si la operación insertó una entrada en el árbol B^+, la información para deshacer D indicaría lo que habría que borrar del árbol B^+. Este registro histórico de información acerca de las operaciones se denomina **registro histórico lógico**. En cambio, el registro histórico de la información sobre el valor anterior y el nuevo valor se denomina **registro histórico físico**, y los correspondientes registros del registro histórico se llaman **registros del registro histórico físico**.

Las operaciones de inserción y borrado son ejemplos de un tipo de operaciones que requiere operaciones deshacer lógicas ya que liberan rápidamente los bloqueos. Estas operaciones se denominan **operaciones lógicas**. Antes de que una operación lógica dé comienzo, escribe en el registro histórico un registro $<T_i, O_j,$ inicio-operación$>$, donde O_j es el identificador único de la operación. Durante la

ejecución de la operación se registran todas las modificaciones realizadas por la operación de forma normal. De esta manera se escribe para cada modificación la información habitual sobre el valor anterior y el nuevo valor. Al finalizar la operación se escribe en el registro histórico un registro de fin-operación como se describió anteriormente.

17.8.2 Retroceso de transacciones

Considérese primero el retroceso de transacciones durante el modo de operación normal (esto es, no durante la fase de recuperación). Se recorre el registro histórico hacia atrás y se usan los registros del registro histórico pertenecientes a la transacción para devolver a los elementos de datos sus valores anteriores. A diferencia del retroceso durante una operación normal, se escriben registros especiales sólo para la operación rehacer de la forma $<T_i,\ X_j,\ V>$ que contienen el valor V con el que se ha restaurado el elemento de datos X_j durante el retroceso. Estos registros del registro histórico se denominan a veces **registros de compensación del registro histórico**. Estos registros no necesitan información para deshacer puesto nunca es necesario realizar esa operación.

Se toman acciones especiales cuando se encuentra un registro del registro histórico de la forma $<T_i, O_j,$ fin-operación, $U>$:

1. Se hace retroceder la operación mediante la información para deshacer D que se encuentra en el registro del registro histórico. Las modificaciones realizadas durante el retroceso de la operación se registran de la misma manera que las modificaciones realizadas cuando la operación se ejecutó por primera vez. En otras palabras, el sistema registra información de deshacer física para las actualizaciones realizadas durante el retroceso, en lugar de usar registros de compensación del registro histórico. Esto es debido a que puede ocurrir una caída mientras la operación deshacer lógica se encuentre en curso, y el sistema debe completar durante la recuperación la operación deshacer lógica, usando la información deshacer física, y después realizar de nueva la operación deshacer lógica, como se verá en el Apartado 17.8.4.

   Al final de la operación de retroceso, en lugar de generar un registro del registro histórico $<T_i, O_j,$ fin-operación, $U>$, el sistema de bases de datos genera $<T_i, O_j,$ abortar-operación$>$.

2. Cuando continúa el recorrido hacia atrás del registro histórico, el sistema omite todos los registros del registro histórico de la transacción hasta que encuentra el registro $<T_i, O_j,$ inicio-operación$>$. Una vez que encuentra el registro inicio-operación en el registro histórico, el resto de registros referentes a la transacción se procesa de nuevo normalmente.

Obsérvese que, durante el retroceso, la omisión de los registros del registro histórico físico cuando se encuentra el registro fin-operación asegura que los valores anteriores del registro no se usen para el retroceso una vez que termine la operación.

Si el sistema encuentra un registro $<T_i, O_j,$ abortar-operación$>$, omite todos los registros precedentes hasta que se encuentra el registro $<T_i, O_j,$ inicio-operación$>$. Estos registros precedentes se deben omitir para evitar varios retrocesos de la misma operación en el caso de que haya una caída durante el retroceso anterior, y de que la transacción haya retrocedido parcialmente. Cuando la transacción T_i haya retrocedido, el sistema añadirá al registro $<T_i$ abortada$>$.

Si sucede un fallo mientras se está ejecutando una operación lógica, el registro fin-operación de la operación no se encontrará cuando la transacción retroceda. Sin embargo, para cada actualización realizada por la operación hay disponible en el registro histórico información para deshacer (como valor anterior en los registros del registro histórico físico). Los registros del registro histórico físico se usarán para hacer retroceder la operación incompleta.

17.8.3 Puntos de revisión

Los puntos de revisión se llevan a cabo como se describió en el Apartado 17.5. Se suspenden temporalmente las modificaciones sobre la base de datos y se llevan a cabo las siguientes acciones:

1. Se escriben en almacenamiento estable todos los registros del registro histórico que se encuentren en ese momento en la memoria principal.

2. Se escriben en disco todos los bloques de la memoria intermedia que se hayan modificado.
3. Se escribe en almacenamiento estable el registro <revisión L>, donde L es una lista de todas las transacciones activas.

17.8.4 Recuperación al reiniciar

Las acciones de recuperación se realizan en dos fases cuando se vuelve a iniciar el sistema de base de datos después de un fallo:

1. En la **fase rehacer** se vuelven a realizar modificaciones de *todas* las transacciones mediante la exploración hacia delante del registro histórico a partir del último punto de revisión. Los registros del registro histórico que se vuelven a ejecutar incluyen los registros de transacciones que retrocedieron antes de la caída del sistema y de las que no se habían comprometido cuando ocurrió la caída. Los registros del registro histórico son los registros habituales de la forma $<T_i,\ X_j,\ V_1,\ V_2>$, así como los registros especiales del registro histórico de la forma $<T_i,\ X_j,\ V_2>$; el elemento de datos X_j adquiere el valor V_2 en cualquier caso. Esta fase también determina todas las transacciones que o bien se encuentran en la lista de transacciones del registro del punto de revisión, o bien comenzaron más tarde, pero no tienen ni el registro $<T_i$ abortada$>$ ni el registro $<T_i$ comprometida$>$ en el registro histórico. Todas estas transacciones deben retrocederse y sus identificadores de transacción se ponen en una lista-deshacer.
2. En la **fase deshacer** retroceden todas las transacciones de la lista-deshacer. El retroceso se realiza recorriendo el registro histórico hacia atrás empezando por el final. Cuando se encuentra un registro del registro histórico perteneciente a una transacción de la lista-deshacer se realizan las operaciones para deshacer de la misma manera que si el registro se hubiera encontrado durante el retroceso de una transacción fallida. Así pues, se ignoran los registros del registro histórico de una transacción que preceden a un registro fin-operación, pero que se encuentran detrás del correspondiente registro inicio-operación.

   Cuando se encuentra en el registro histórico un registro $<T_i$ iniciada$>$ para una transacción T_i de la lista-deshacer, se escribe en el registro histórico un registro $<T_i$ abortada$>$. El recorrido hacia atrás del registro histórico finaliza cuando se encuentran los registros $<T_i$ iniciada$>$ para todas las transacciones de la lista-deshacer.

La fase rehacer de la recuperación al reiniciar reproduce cada registro físico del registro histórico desde que tuvo lugar el último punto de revisión. En otras palabras, esta fase de la recuperación al reiniciar repite todas las acciones de modificación que fueron ejecutadas después del punto de revisión y cuyos registros alcanzaron un registro histórico estable. Se incluyen aquí las acciones de transacciones incompletas y las acciones llevadas a cabo para retroceder transacciones fallidas. Las acciones se repiten en el mismo orden en el que se llevaron a cabo; de aquí que este proceso se denomine **repetición de la historia**. Al repetir la historia se simplifican bastante los esquemas de recuperación.

Obsérvese que si una operación deshacer estaba en curso cuando ocurrió la caída del sistema, se encontrarían los registros del registro histórico físico escritos durante la operación deshacer, y la operación deshacer parcial se desharía según estos registros del registro histórico físico. Después de ello, el registro fin-operación de la operación original se encontraría durante la recuperación, y la operación deshacer se ejecutaría de nuevo.

17.8.5 Revisión difusa

La revisión difusa descrita en el Apartado 17.5.3 requiere que, mientras se efectúa el punto de revisión, se suspendan temporalmente todas las modificaciones de la base de datos. Si el número de páginas de la memoria intermedia es grande, un punto de revisión puede llevar mucho tiempo, lo que puede provocar una interrupción inaceptable en el procesamiento de transacciones.

Para evitar estas interrupciones es posible modificar la técnica para permitir modificaciones después de haber escrito en el registro histórico el registro revisión, pero antes de escribir en disco los bloques

de la memoria intermedia que han sufrido modificaciones. El punto de revisión así generado recibe el nombre de **punto de revisión difuso**.

Dado que las páginas se escriben en disco sólo después de que se haya escrito el registro revisión, es posible que el sistema caiga antes de que todas las páginas se hayan escrito. Por tanto, los puntos de revisión en disco pueden estar incompletos. Una forma de manejar los puntos de revisión incompletas es la siguiente. La ubicación del registro de revisión en el registro histórico del último punto de revisión completado se almacena en una posición fijada, última-revisión, en disco. El sistema no actualiza esta información cuando escribe el registro revisión. En su lugar, antes de escribirlo, crea una lista de todos los bloques de memoria intermedia modificados. La información de última-revisión se actualiza sólo cuando todos los bloques de memoria intermedia de la lista de bloques modificados se hayan escrito en disco.

Incluso con la revisión difusa, un bloque de memoria intermedia no se debe actualizar mientras se esté escribiendo en disco, aunque otros bloques sí se pueden actualizar concurrentemente. Una vez que todos los bloques han sido escritos en disco debe seguirse el protocolo de registro histórico de escritura anticipada.

Obsérvese que en este esquema sólo se utiliza el proceso de registro histórico lógico para deshacer mientras que el proceso de registro histórico físico se usa tanto para deshacer como para rehacer. Existen otros esquemas de recuperación que utilizan el proceso de registro histórico lógico para rehacer. Para deshacer lógicamente, el estado de la base de datos en disco debe ser **consistente en cuanto a operaciones**, es decir, no debería tener efectos parciales de ninguna operación. Es difícil garantizar esta consistencia de la base de datos en disco si una operación puede afectar a más de una página, puesto que no es posible escribir más de una página de forma atómica. Por tanto, el registro histórico deshacer lógico se restringe usualmente sólo a operaciones que afectan a una única página. Se verá la forma de tratar estas operaciones rehacer lógicas en el Apartado 17.8.6. En cambio, las operaciones deshacer lógicas se realizan sobre un estado consistente en cuanto a operaciones de la base de datos repitiendo la historia y después realizando la operación deshacer física de las operaciones completadas parcialmente.

17.8.6 ARIES

El método de recuperación ARIES es un representante de los métodos actuales de recuperación. La técnica de recuperación avanzada que se ha descrito se ha modelado después de ARIES, pero se ha simplificado significativamente para ilustrar los conceptos clave y hacerlo más fácil de comprender. En cambio, ARIES utiliza varias técnicas para reducir el tiempo de recuperación y para reducir la sobrecarga de los puntos de revisión. En particular, ARIES es capaz de evitar rehacer muchas operaciones registradas que ya se han realizado y de reducir la cantidad de información registrada. El precio pagado supone una mayor complejidad, pero los beneficios merecen la pena.

Las diferencias principales entre ARIES y el algoritmo de recuperación avanzada expuesto son que ARIES:

1. Usa un **número de secuencia del registro histórico (NSR)** para identificar a los registros del registro histórico, y emplea esos números en las páginas de la base de datos para identificar las operaciones que se han realizado sobre una página de la base de datos.

2. Soporta operaciones **rehacer fisiológicas**, que son físicas en el sentido en que la página afectada está físicamente identificada, pero que pueden ser lógicas en la página.

   Por ejemplo, el borrado de un registro de una página puede provocar que muchos otros registros de la página se desplacen si se usa una estructura de páginas con ranuras. Con el registro histórico rehacer físico, hay que registrar todos los bytes de la página afectada por el desplazamiento de los registros. Con el registro histórico fisiológico, la operación borrado se puede registrar, lo que da lugar a un registro mucho más pequeño. Al rehacer la operación borrado se borraría el registro y se desplazarían los registros que hiciera falta.

3. Emplea una **tabla de páginas desfasadas** para minimizar las operaciones rehacer innecesarias durante la recuperación. Las páginas desfasadas son las que se han actualizado en memoria pero no en su versión en disco.

4. Utiliza un esquema de revisión difusa que sólo registra información sobre las páginas desfasadas e información asociada, y no requiere siquiera la escritura de las páginas desfasadas a disco. Procesa las páginas desfasadas en segundo plano continuamente, en lugar de escribirlas durante los puntos de revisión.

En el resto de este apartado se proporciona una visión general de ARIES. Las notas bibliográficas incluyen referencias que proporcionan una descripción completa de ARIES.

17.8.6.1 Estructuras de datos

Cada registro del registro histórico de ARIES tiene un **número de secuencia del registro histórico (NSR)** que lo identifica unívocamente. El número es conceptualmente tan sólo un identificador lógico cuyo valor es mayor para los registros que aparecen después en el registro histórico. En la práctica, el NSR se genera de forma que también se puede usar para localizar el registro del registro histórico en disco. Normalmente, ARIES divide el registro histórico en varios archivos de registro histórico, cada uno con un número de archivo. Cuando un archivo crece hasta un determinado límite, ARIES añade los nuevos registros del registro histórico en un nuevo archivo; el nuevo archivo de registro histórico tiene un número de archivo que es 1 mayor que el anterior archivo. El NSR consiste en un número de archivo y un desplazamiento dentro del archivo.

Cada página también mantiene un identificador denominado **NSRPágina**. Cada vez que se aplica una operación (física o lógica) en la página, la operación almacena el NSR de su registro en el campo NSRPágina de la página. Durante la fase rehacer de la recuperación cualquier registro con un NSR menor o igual que el NSRPágina de la página no se debería ejecutar, ya que sus acciones ya están reflejadas en la página. En combinación con un esquema para el registro de los NSRPágina como parte de los puntos de revisión, que se presenta más adelante, ARIES puede evitar incluso leer muchas páginas cuyas operaciones registradas ya se han reflejado en el disco. Por tanto, el tiempo de recuperación se reduce significativamente.

El NSRPágina es esencial para asegurar la idempotencia en presencia de operaciones rehacer fisiológicas, ya que volver a aplicar una operación rehacer fisiológica que ya se haya aplicado a una página podría causar cambios incorrectos en una página.

Las páginas no se deberían enviar a disco mientras se esté realizando una actualización, dado que las operaciones fisiológicas no se pueden rehacer sobre el estado parcialmente actualizado de la página en disco. Por tanto, ARIES usa pestillos sobre las páginas de la memoria intermedia para evitar que se escriban en disco mientras se actualicen. Los pestillos de las páginas de la memoria intermedia sólo se liberan cuando se completan las actualizaciones, y el registro del registro histórico para la actualización se haya escrito en el registro histórico.

Cada registro del registro histórico también contiene el NSR del registro anterior de la misma transacción. Este valor, almacenado en el campo NSRAnterior, permite que se encuentren los registros del registro histórico anteriores sin necesidad de leer el registro histórico completo. En ARIES hay registros especiales sólo-rehacer generados durante el retroceso de transacciones, denominados **registros de compensación del registro histórico** (RCR). Los RCR tienen un campo extra denominado DeshacerSiguienteNSR, que registra el NSR del registro que hay que deshacer a continuación cuando la transacción retrocede. Este campo sirve para el mismo propósito que el identificador de operaciones en el registro abortar-operación del esquema anterior, que ayuda a omitir los registros que ya hayan retrocedido. La **TablaPáginasDesfasadas** contiene una lista de páginas que se han actualizado en la memoria intermedia de la base de datos. Para cada página se almacena el NSRPágina y un campo denominado RegNSR que ayuda a identificar los registros que ya se han aplicado a la versión en disco de la página. Cuando se inserta una página en la TablaPáginasDesfasadas (cuando se modifica por primera vez en el grupo de memorias intermedias) el valor de RegNSR se establece en el fin actual del registro histórico. Cada vez que se envía una página a disco, la página se elimina de la TablaPáginasDesfasadas.

El **registro punto de revisión del registro histórico** contiene la TablaPáginasDesfasadas y una lista de transacciones activas. Para cada transacción, el registro punto de revisión del registro histórico también anota ÚltimoNSR, el NSR del último registro escrito por la transacción. Una posición fijada en disco también anota el NSR del último registro punto de revisión del registro histórico (completado).

17.8.6.2 Algoritmo de recuperación

ARIES recupera de una caída del sistema en tres fases:

- **Paso de análisis**. Este paso determina las transacciones que hay que deshacer, las páginas que están desfasadas en el momento de la caída y el NSR en el que debería comenzar el paso rehacer.
- **Paso rehacer**. Este paso comienza en una posición determinada durante el análisis y realiza una operación rehacer, repitiendo la historia, para llevar a la base de datos al estado anterior a la caída.
- **Paso deshacer**. Este paso hace retroceder todas las transacciones incompletas en el momento de la caída.

Paso de análisis. El paso de análisis busca el último registro punto de revisión completado del registro histórico y lee la TablaPáginasDesfasadas en este registro. A continuación establece RehacerNSR al mínimo RegistroNSR de las páginas de TablaPáginasDesfasadas. Si no hay páginas desfasadas, establece RehacerNSR al NSR del registro punto de revisión del registro histórico. El paso rehacer comienza explorando el registro histórico desde RehacerNSR. Todos los registros anteriores a este punto ya se han aplicado a las páginas de la base de datos en el disco. El paso de análisis establece inicialmente la lista de transacciones que se deben deshacer, lista-deshacer, a la lista de transacciones en el registro punto de revisión del registro histórico. El paso de análisis también lee del registro punto de revisión del registro histórico los NSR del último registro del registro histórico de cada transacción de las lista-deshacer.

El paso de análisis continúa examinando hacia delante desde el punto de revisión. Cada vez que encuentra un registro de una transacción que no esté en la lista-deshacer, añade la transacción a la lista-deshacer. Cada vez que encuentra un registro de fin de transacción, borra la transacción de la lista-deshacer. Todas las transacciones que queden en la lista-deshacer al final del análisis se deben hacer retroceder más tarde en el paso deshacer. El paso de análisis también almacena el último registro de cada transacción en la lista-deshacer, que se usa en el paso deshacer.

El paso de análisis también actualiza TablaPáginasDesfasadas cada vez que encuentra un registro del registro histórico de la actualización de una página. Si la página no está en la TablaPáginasDesfasadas, el paso de análisis la añade a ella y establece el RegistroNSR de la página al NSR del registro.

Paso rehacer. El paso rehacer repite la historia volviendo a ejecutar cada acción sobre una página que no se haya reflejado en disco. El paso rehacer examina el registro histórico hacia delante a partir de RehacerNSR. Cada vez que encuentra un registro actualizar realiza:

1. Si la página no está en la TablaPáginasDesfasadas o el NSR del registro actualizar es menor que el RegistroNSR de la página de TablaPáginasDesfasadas, entonces el paso rehacer omite el registro.
2. En caso contrario, el paso rehacer extrae la página de disco y, si NSRPágina es menor que el NSR del registro, se rehace el registro.

Obsérvese que si cualquiera de las comprobaciones son negativas, entonces los efectos del registro del registro histórico ya han aparecido en la página. Si la primera comprobación es negativa, ni siquiera es necesario extraer la página de disco.

Paso deshacer y retroceso de transacciones. El paso deshacer es relativamente simple. Realiza una exploración hacia atrás del registro histórico, deshaciendo todas las transacciones de la lista-deshacer. Si se encuentra un RCR, usa el campo DeshacerSiguienteNSR para omitir los registros que ya se hayan retrocedido. En caso contrario, usa el campo NSRAnterior del registro para encontrar el siguiente a deshacer.

Cada vez que se usa un registro del registro histórico para realizar una operación deshacer (para el retroceso de transacciones durante el procesamiento normal del retroceso o durante el reinicio del paso deshacer) el paso deshacer genera un RCR que contiene la acción deshacer realizada (que debe ser fisiológica). Establece DeshacerSiguienteNSR del RCR al valor NSRAnterior del registro actualizar del registro histórico.

17.8.6.3 Otras características

Algunas de las características que proporciona ARIES son:

- **Independencia de recuperación**. Algunas páginas se pueden recuperar independientemente de otras, de forma que se pueden usar incluso cuando se estén recuperando otras. Si fallan algunas páginas del disco se pueden recuperara sin parar el procesamiento de transacciones en otras páginas.
- **Puntos de almacenamiento**. Las transacciones pueden registrar puntos de almacenamiento y pueden retroceder parcialmente hasta un punto de almacenamiento. Esto puede ser muy útil en el manejo de interbloqueos, dado que las transacciones pueden retroceder hasta un punto que permita la liberación de los bloqueos requeridos y luego reiniciarse desde ese punto.
- **Bloqueo de grano fino**. El algoritmo de recuperación ARIES se puede usar con algoritmos de control de concurrencia de índices que permiten el bloqueo en el nivel de tuplas de los índices, en lugar del bloqueo en el nivel de las páginas, lo que aumenta significativamente la concurrencia.
- **Optimizaciones de la recuperación**. La TablaPáginasDesfasadas se puede usar para preextraer páginas durante la operación rehacer, en lugar de extraer una página sólo cuando el sistema encuentra un registro del registro histórico a aplicar a la página. La operación rehacer-no-válidos también es posible. Esta operación se puede posponer sobre una página que se vaya a extraer del disco y realizarse cuando se extraiga. Mientras tanto se pueden seguir procesando otros registros.

En resumen, el algoritmo ARIES es un algoritmo de recuperación actual que incorpora varias optimizaciones diseñadas para mejorar la concurrencia, reducir la sobrecarga por el registro histórico y reducir el tiempo de recuperación.

17.9 Sistemas remotos de copias de seguridad

Los sistemas tradicionales de procesamiento de transacciones son sistemas centralizados o sistemas cliente-servidor. Esos sistemas son vulnerables frente a desastres ambientales como el fuego, las inundaciones o los terremotos. Hay una necesidad creciente de sistemas de procesamiento de transacciones que ofrezcan una disponibilidad elevada y que puedan funcionar pese a los desastres ambientales. Estos sistemas deben proporcionar una **disponibilidad elevada**, es decir, el tiempo en que el sistema no es utilizable debe ser extremadamente pequeño.

Se puede obtener una disponibilidad elevada realizando el procesamiento de transacciones en un solo sitio, denominado **sitio principal**, pero tener un **sitio remoto copia de seguridad**, en el que se repliquen todos los datos del sitio principal. El sitio remoto copia de seguridad se denomina también a veces **sitio secundario**. El sitio remoto debe mantenerse sincronizado con el sitio principal, ya que las actualizaciones se realizan en el sitio principal. La sincronización se obtiene enviando todos los registros del registro histórico desde el sitio principal al sitio remoto de copia de seguridad. El sitio remoto copia de seguridad debe hallarse físicamente separado del principal—por ejemplo, se puede ubicar en otra provincia—para que una catástrofe en el sitio principal no afecte al sitio remoto copia de seguridad. En la Figura 17.8 se muestra la arquitectura de los sistemas remotos para copias de seguridad.

Cuando falla el sitio principal, el sitio remoto de copia de seguridad asume el procesamiento. En primer lugar, sin embargo, lleva a cabo la recuperación utilizando su copia (tal vez anticuada) de los datos del sitio principal y de los registros del registro histórico recibidos del mismo. En realidad, el sitio remoto copia de seguridad lleva a cabo acciones de recuperación que se hubieran llevado a cabo en el sitio principal cuando éste se hubiera recuperado. Se pueden utilizar los algoritmos estándar de recuperación para la recuperación en el sitio remoto copia de seguridad con pocas modificaciones. Una vez realizada la recuperación, el sitio remoto copia de seguridad comienza a procesar transacciones.

La disponibilidad aumenta mucho en comparación con los sistemas con un solo sitio, dado que el sistema puede recuperarse aunque se pierdan todos los datos del sitio principal. El rendimiento de los sistemas remotos para copias de seguridad es mejor que el de los sistemas distribuidos con compromiso de dos fases.

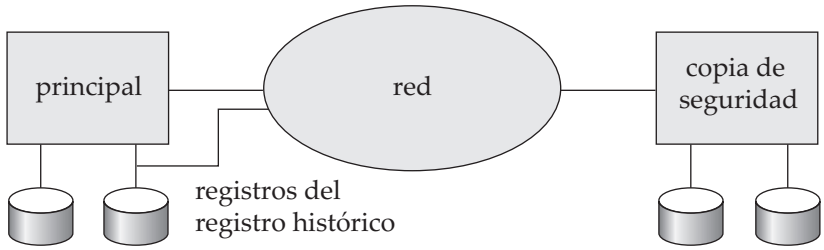


Figura 17.8 Arquitectura de los sistemas remotos de copias de seguridad.

Se deben abordar varios aspectos al diseñar sistemas remotos para copias de seguridad, tales como:

- **Detección de fallos**. Al igual que en los protocolos para el manejo de fallos en sistemas distribuidos, es importante que el sistema remoto de copia de seguridad detecte que el sitio principal ha fallado. El fallo de las líneas de comunicación puede hacer creer al sitio remoto de copia de seguridad que el sitio principal ha fallado. Para evitar este problema hay que mantener varios enlaces de comunicaciones con modos de fallo independientes entre el sitio principal y el sitio remoto copia de seguridad. Por ejemplo, además de la conexión de red puede haber otra conexión mediante módem por línea telefónica con servicio suministrado por diferentes compañías de telecomunicaciones. Estas conexiones pueden complementarse con la intervención manual de operadores, que se pueden comunicar por vía telefónica.

- **Transferencia del control**. Cuando el sitio principal falla, el sitio de copia de seguridad asume el procesamiento y se transforma en el nuevo sitio principal. Cuando el sitio principal original se recupera puede desempeñar el papel de sitio remoto de copia de seguridad o volver a asumir el papel de sitio principal. En cualquiera de los casos, el sitio principal antiguo debe recibir un registro histórico de actualizaciones realizado por el sitio de copia de seguridad mientras el sitio principal antiguo estaba fuera de servicio.

  La manera más sencilla de transferir el control es que el sitio principal antiguo reciba el registro histórico de operaciones rehacer del sitio de copia de seguridad antiguo y se ponga al día con las actualizaciones aplicándolas de manera local. El sitio principal antiguo puede entonces actuar como sitio remoto de copia de seguridad. Si hay que devolver el control, el sitio remoto copia de seguridad antiguo puede simular que ha fallado, lo que da lugar a que el sitio principal antiguo asuma el control.

- **Tiempo de recuperación**. Si el registro histórico del sitio remoto de copia de seguridad se hace grande, la recuperación puede tardar mucho. El sitio remoto copia de seguridad puede procesar de manera periódica los registros rehacer del registro histórico que haya recibido y realizar un punto de revisión, de manera que se puedan borrar las partes más antiguas del registro histórico. Como consecuencia se puede reducir el retraso antes de que el sitio remoto de copia de seguridad asuma el control.

  Una configuración de **relevo en caliente** puede hacer la toma del control por el sitio de copia de seguridad casi instantáneo. En esta configuración el sitio remoto copia de seguridad procesa los registros rehacer del registro histórico según llegan, y aplica las actualizaciones de manera local. Tan pronto como se detecta el fallo del sitio principal, el sitio de copia de seguridad completa la recuperación haciendo retroceder las transacciones incompletas y queda preparado para procesar las nuevas.

- **Tiempo de compromiso**. Para asegurar que las actualizaciones de una transacción comprometida sean duraderas no se debe declarar comprometida una transacción hasta que sus registros del registro histórico hayan alcanzado el sitio de copia de seguridad. Este retraso puede dar lugar a una espera más prolongada para comprometer la transacción y, en consecuencia, algunos sistemas permiten grados inferiores de durabilidad. Los grados de durabilidad pueden clasificarse de la manera siguiente.

- **Uno seguro**. Las transacciones se comprometen tan pronto como sus registros de compromiso del registro histórico se escriben en un almacenamiento estable en el sitio principal.

  El problema de este esquema es que puede que las actualizaciones de una transacción comprometida no hayan alcanzado el sitio de copia de seguridad cuando éste asuma el control del procesamiento. Por tanto, puede parecer que las actualizaciones se han perdido. Cuando se recupera el sitio principal, las actualizaciones perdidas no se pueden mezclar directamente, dado que pueden entrar en conflicto con actualizaciones posteriores llevadas a cabo en el sitio de copia de seguridad. Por tanto, puede que se necesite la intervención humana para devolver a la base de datos a un estado consistente.
- **Dos muy seguro**. Las transacciones se comprometen tan pronto como sus registros de compromiso del registro histórico se escriben en un almacenamiento estable en el sitio principal y en el sitio de copia de seguridad.

  El problema de este esquema es que el procesamiento de transacciones no puede continuar si alguno de los puntos no está operativo. Por tanto, la disponibilidad es realmente menor que en el caso de un solo sitio, aunque la posibilidad de la pérdida de datos es mucho más reducida.
- **Dos seguro**. Este esquema es idéntico al esquema dos muy seguro si tanto el sitio principal como el sitio de copia de seguridad están activos. Si sólo está activo el sitio principal se permite que la transacción se comprometa tan pronto como su registro de compromiso del registro histórico se escriba en un almacenamiento estable en el sitio principal.

  Este esquema proporciona mejor disponibilidad que el esquema dos muy seguro, al tiempo que evita el problema de las transacciones perdidas afrontado por el esquema uno seguro. Da lugar a un compromiso más lento que el esquema uno seguro pero las ventajas generalmente superan los inconvenientes.

Varios sistemas comerciales de disco compartido proporcionan un nivel de tolerancia de fallos intermedio entre el de los sistemas centralizados y el de los sistemas remotos para copias de seguridad. En estos sistemas el fallo de una CPU no da lugar al fallo del sistema. En lugar de ello, otra CPU asume el control y lleva a cabo la recuperación. Las acciones de recuperación incluyen el retroceso de las transacciones que se ejecutaban en la CPU que falló y la recuperación de los bloqueos mantenidos por dichas transacciones. Dado que los datos se hallan en un disco compartido, no hace falta transferir registros del registro histórico. Sin embargo, se deberían proteger los datos contra el fallo del disco utilizando, por ejemplo, una organización de discos RAID.

Una forma alternativa de conseguir alta disponibilidad es usar una base de datos distribuida con los datos replicados en más de un sitio. Son necesarias transacciones para actualizar todas las réplicas de cualquier elemento de datos que actualicen. Las bases de datos distribuidas, incluyendo la réplica, se estudian en el Capítulo 22.

17.10 Resumen

- Los sistemas informáticos, al igual que cualquier otro dispositivo eléctrico o mecánico, están sujetos a fallos. Estos fallos se producen por diferentes motivos incluyendo fallos de disco, cortes de corriente o fallos en el software. En cada uno de estos casos puede perderse información concerniente a la base de datos.
- Las transacciones pueden fallar, además de por un fallo del sistema, por otras razones como una violación de las restricciones de integridad o interbloqueos.
- El esquema de recuperación es una parte integrante del sistema de base de datos el cual es responsable de la detección de fallos y del restablecimiento de un estado de la base datos anterior al momento de producirse el fallo.
- Los diferentes tipos de almacenamiento en una computadora son el volátil, el no volátil y el almacenamiento estable. Los datos del almacenamiento volátil, como ocurre con los guardados en la memoria RAM, se pierden cuando la computadora cae. Los datos del almacenamiento no volátil, como los guardados en un disco, no se pierden cuando la computadora cae, pero pueden

perderse ocasionalmente debido a fallos de disco. Los datos del almacenamiento estable nunca se pierden.

- El almacenamiento estable de acceso en tiempo real puede aproximarse con discos con imagen u otras formas de RAID que proporcionan almacenamiento redundante de datos. El almacenamiento estable, sin conexión o de archivo, puede consistir en una serie de copias en cinta de los datos guardadas en un lugar físicamente seguro.
- El estado del sistema de base de datos puede no volver a ser consistente en caso de ocurrir un fallo; esto es, puede no reflejar un estado del mundo potencialmente alcanzable por la base de datos. Para preservar la consistencia es necesario que cada transacción sea atómica. Garantizar la propiedad de atomicidad es responsabilidad del esquema de recuperación.
- En los esquemas basados en registro histórico todas las modificaciones se escriben en el registro histórico, el cual debe estar guardado en almacenamiento estable.
  - En el esquema de modificación diferida, durante la ejecución de una transacción, se difieren todas las operaciones escribir hasta que la transacción se compromete parcialmente, momento en el que se utiliza la información del registro histórico asociada con la transacción para ejecutar las escrituras diferidas.
  - Con la técnica de modificación inmediata, todas las modificaciones se aplican directamente sobre la base de datos. Si ocurre una caída se utiliza la información del registro histórico para conducir a la base de datos a un estado consistente previo.

  Puede usarse la técnica de los puntos de revisión para reducir la sobrecarga que conlleva la búsqueda en el registro histórico y rehacer las transacciones.
- El procesamiento de transacciones se basa en un modelo de almacenamiento en el que la memoria principal contiene una memoria intermedia para el registro histórico, una memoria intermedia para la base de datos y una memoria intermedia para el sistema. La memoria intermedia del sistema alberga páginas de código objeto del sistema y áreas de trabajo local de las transacciones.
- Una implementación eficiente de un esquema de recuperación de datos requiere que el número de escrituras en la base de datos y en almacenamiento estable sea mínimo. Los registros del registro histórico pueden guardarse inicialmente en la memoria intermedia del registro histórico en almacenamiento volátil, pero se deben copiar en almacenamiento estable cuando se da una de estas dos condiciones:
  - Deben escribirse en almacenamiento estable todos los registros del registro histórico pertenecientes a la transacción T_i antes de que el registro $<T_i$ comprometida$>$ se pueda escribir en almacenamiento estable.
  - Deben escribirse en almacenamiento estable todos los registros del registro histórico pertenecientes a los datos de un bloque antes de que ese bloque de datos se escriba desde la memoria principal a la base de datos (en almacenamiento no volátil).
- Para recuperarse de los fallos que resultan en la pérdida de almacenamiento no volátil debe realizarse un volcado periódicamente (por ejemplo, una vez al día) del contenido entero de la base de datos en almacenamiento estable. Se usará el último volcado para devolver a la base de datos a un estado consistente previo cuando ocurra un fallo que conduzca a la pérdida de algún bloque físico de la base de datos. Una vez realizada esta operación se utilizará el registro histórico para llevar a la base de datos al estado consistente más reciente.
- Se han desarrollado técnicas avanzadas de recuperación para soportar técnicas de bloqueo de alta concurrencia, como las utilizadas para el control de concurrencia con árboles B^+. Estas técnicas se basan en el registro deshacer lógico y siguen el principio de repetir la historia. En la recuperación de un fallo del sistema se realiza una fase rehacer utilizando el registro histórico, seguida de una fase deshacer sobre el registro histórico para retroceder las transacciones incompletas.
- El esquema de recuperación ARIES es un esquema actual que soporta varias características para proporcionar mayor concurrencia, reducir la sobrecarga del registro histórico y permitir opera-

ciones deshacer lógicas. El esquema procesa páginas continuamente y no necesita procesar todas las páginas en el momento de un punto de revisión. Usa números de secuencia del registro histórico (NSR) para implementar varias optimizaciones que reducen el tiempo de recuperación.

- Los sistemas remotos de copia de seguridad proporcionan un alto nivel de disponibilidad, permitiendo que continúe el procesamiento de transacciones incluso si se destruye el sitio primario por fuego, inundación o terremoto.

Términos de repaso

- Esquema de recuperación.
- Clasificación de los fallos:
  - Fallo de transacción.
  - Error lógico.
  - Error del sistema.
  - Caída del sistema.
  - Fallo de transferencia de datos.
- Supuesto de fallo-parada.
- Fallo de disco.
- Tipos de almacenamiento:
  - Volátil
  - No volátil
  - Estable
- Bloques:
  - Físicos.
  - De memoria intermedia.
- Memoria intermedia de disco.
- Escritura forzada.
- Recuperación basada en el registro histórico.
- Registro histórico.
- Registros del registro histórico.
- Registro actualizar del registro histórico.
- Modificación diferida.
- Idempotente.
- Modificación inmediata.
- Modificaciones no comprometidas.
- Puntos de revisión.
- Recogida de basura.
- Recuperación con transacciones concurrentes.
  - Retroceso de transacciones.
  - Puntos de revisión difusos.
  - Recuperación al reiniciar.
- Gestión de la memoria intermedia.
- Registro histórico con memoria intermedia.
- Registro de escritura anticipada (REA).
- Forzar el registro histórico.
- Memoria intermedia de la base de datos.
- Pestillos.
- Sistema operativo y gestión de la memoria intermedia.
- Pérdida del almacenamiento no volátil.
- Volcado de archivo.
- Volcado difuso.
- Técnica de recuperación avanzada.
  - Operación deshacer física.
  - Operación deshacer lógica.
  - Registro histórico físico.
  - Registro histórico lógico.
  - Operaciones lógicas.
  - Retroceso de transacciones.
  - Puntos de revisión.
  - Recuperación al reiniciar.
  - Fase rehacer.
  - Fase deshacer.
- Repetición de la historia.
- Puntos de revisión difusos.
- ARIES
  - Número de secuencia del registro histórico (NSR).
  - NSRPágina.
  - Operación rehacer fisiológica.
  - Registros de compensación del registro histórico (RCR).
  - TablaPáginasDesfasadas.
  - Registro punto de revisión.
- Alta disponibilidad.
- Sistemas remotos de copia de seguridad.
  - Sitio principal.
  - Sitio remoto de copia de seguridad.
  - Sitio secundario.
- Detección de fallos.
- Transferencia del control.
- Tiempo de recuperación.
- Configuración de relevo en caliente.
- Tiempo de compromiso.

□ Uno seguro.
□ Dos muy seguro.
□ Dos seguro.

Ejercicios prácticos

17.1 Compárense, en términos de facilidad de implementación y de sobrecarga, las versiones de modificación inmediata y modificación diferida de las técnicas de recuperación basadas en registro histórico.

17.2 Cuando el sistema se recupera después de una caída (véase el Apartado 17.5.4) construye una lista-deshacer y una lista-rehacer. Explíquese por qué deben procesarse en orden inverso los registros del registro histórico de las transacciones que se encuentran en la lista-deshacer, mientras que los registros del registro histórico correspondientes a las transacciones de la lista-rehacer se procesan hacia delante.

17.3 Explíquense las razones por las que la recuperación en transacciones interactivas es más difícil de tratar que la recuperación en transacciones por lotes. ¿Existe alguna forma sencilla de tratar esta dificultad? *Sugerencia*: considérese una transacción de un cajero automático por la que se retira dinero.

17.4 A veces hay que deshacer una transacción después de que se haya comprometido porque se ejecutó erróneamente, debido por ejemplo a la introducción incorrecta de datos de un cajero.

a. Dese un ejemplo para demostrar que el uso de un mecanismo normal para deshacer esta transacción podría conducir a un estado inconsistente.

b. Una forma de manejar esta situación es llevar la base de datos a un estado anterior al compromiso de la transacción errónea (denominado recuperación *a un instante*). En este esquema se deshacen los efectos de las transacciones comprometidas después.

Sugiérase una modificación del mecanismo de recuperación avanzada para implementar la recuperación a un instante.

c. Las transacciones correctas se pueden volver a ejecutar lógicamente, pero no se pueden reejecutar usando sus registros del registro histórico. ¿Por qué?

17.5 En los lenguajes de programación persistentes no se realiza explícitamente el registro histórico de las modificaciones. Descríbase cómo pueden usarse las protecciones de acceso a las páginas que proporcionan los sistemas operativos modernos para crear imágenes anteriores y posteriores de las páginas que son modificadas. *Sugerencia*: véase el Ejercicio práctico 16.9.

17.6 ARIES da por supuesto que hay espacio en cada página para un NSR. Al manejar objetos grandes que abarcan varias páginas, tales como archivos del sistema operativo, un objeto puede usar una página completa, sin dejar espacio para el NSR. Sugiérase una técnica para manejar esta situación; esta técnica debe soportar operaciones rehacer físicas pero no es necesario que soporte operaciones rehacer fisiológicas.

Ejercicios

17.7 Explíquese la diferencia en cuanto al coste E/S entre los tres tipos de almacenamiento—volátil, no volátil y estable.

17.8 El almacenamiento estable no se puede implementar.

a. Explíquese el motivo.

b. Explíquese cómo tratan este problema los sistemas de las bases de datos.

17.9 Supóngase que un sistema utiliza modificación inmediata. Demuéstrese con un ejemplo cómo podría darse un estado inconsistente en la base de datos si no se escriben en almacenamiento estable los registros del registro histórico de una transacción antes de que el dato actualizado por la transacción se escriba a disco.

17.10 Explíquese el propósito del mecanismo de los puntos de revisión. ¿Con qué frecuencia deberían realizarse los puntos de revisión? Explíquese cómo afecta la frecuencia de los puntos de revisión:

- Al rendimiento del sistema cuando no ocurre ningún fallo.
- Al tiempo que se tarda para recuperarse de una caída del sistema.
- Al tiempo que se tarda para recuperarse de una caída del disco.

17.11 Explíquese cómo el gestor de la memoria intermedia puede conducir a la base de datos a un estado inconsistente si algunos registros del registro histórico pertenecientes a un bloque no se escriben en almacenamiento estable antes de escribir en el disco el citado bloque.

17.12 Explíquese las ventajas del registro histórico lógico. Proporciónense ejemplos de una situación en la que sea preferible el registro histórico lógico frente al registro histórico físico y de lo contrario.

17.13 Explíquese la diferencia entre una caída del sistema y un "desastre".

17.14 Para cada uno de los siguientes requisitos identifíquese la mejor opción del grado de durabilidad en un sistema remoto de copia de seguridad.

a. Pérdida de datos que se debe evitar pero se puede tolerar alguna pérdida de disponibilidad.
b. El compromiso de transacciones se debe realizar rápidamente, incluso perdiendo algunas transacciones comprometidas en caso de desastre.
c. Se requiere un alto grado de disponibilidad y durabilidad, pero es aceptable un mayor tiempo de ejecución para el protocolo de compromiso de transacciones.

17.15 Las aplicaciones del sistema de bases de datos de Oracle deshacen los registros del registro histórico para proporcionar una vista instantánea de la base de datos a las transacciones de sólo lectura. La vista instantánea refleja las actualizaciones de todas las transacciones comprometidas cuando comenzó la transacción de sólo lectura; las actualizaciones del resto de transacciones no son visibles para la transacción de sólo lectura.

Descríbase un esquema para el gestor de memoria intermedia en el que se proporcione una vista instantánea de las páginas de la memoria intermedia. Inclúyanse los detalles de la forma de usar el registro histórico para generar la vista instantánea, asumiendo que se use el algoritmo avanzado de recuperación. Asúmase por simplicidad que tanto la operación lógica como deshacerla afectan sólo a una página.

Notas bibliográficas

El libro de Gray y Reuter [1993] constituye una excelente fuente de información sobre la recuperación incluyendo interesantes implementaciones y detalles históricos. El libro de Bernstein et al. [1987] es una fuente de información sobre el control de concurrencia y recuperación. Davies [1973] y Bjork [1973] son dos de los primeros documentos que presentan trabajos teóricos en el campo de la recuperación. Otro trabajo pionero en este campo es el de Chandy et al. [1975], que describe modelos analíticos para el retroceso y las estrategias de recuperación en los sistemas de bases de datos.

En Gray et al. [1981] se presenta una visión de conjunto del esquema de recuperación de System R. En Gray [1978], Lindsay et al. [1980] y Verhofstad [1978] pueden encontrarse guías de aprendizaje y visiones de conjunto sobre varias técnicas de recuperación para sistemas de bases de datos. Los conceptos de punto de revisión difusa y volcado difuso se describen en Lindsay et al. [1980]. Haerder y Reuter [1983] ofrece una amplia presentación de los principios de la recuperación.

La situación actual de los métodos de recuperación se ilustra mejor con el método de recuperación ARIES, descrito en Mohan et al. [1992] y en Mohan [1990b]. ARIES y sus variantes se usan en varios productos de bases de datos, incluyendo DB2 de IBM y SQL Server de Microsoft. La recuperación en Oracle se describe en Lahiri et al. [2001a].

Mohan y Levine [1992] y Mohan [1993] proporcionan técnicas de recuperación especializadas para estructuras con índices; Mohan y Narang [1994] describe técnicas de recuperación para arquitecturas cliente-servidor, mientras que Mohan y Narang [1991] y Mohan y Narang [1992] describen técnicas de recuperación para arquitecturas de bases de datos paralelas.

King et al. [1991] y Polyzois y Garcia-Molina [1994] consideran las copias de seguridad remotas para recuperación de desastres (pérdida completa de un componente del sistema informático a causa de, por ejemplo, un incendio, una inundación o un terremoto).

En el Capítulo 25 se encuentran referencias sobre las transacciones de larga duración y los aspectos de recuperación relacionados.

PARTE 6

Minería de datos y recuperación de información

Las consultas de la base de datos se diseñan normalmente para extraer información específica, como el saldo de una cuenta o la suma de los saldos de la cuentas de un cliente. Sin embargo, las consultas diseñadas para ayudar a formular una estrategia corporativa requieren a menudo la agregación a una escala mucho más grande, e incluyen análisis estadísticos que no se expresan fácilmente con las características de SQL vistas anteriormente. Estas consultas deben tener acceso a menudo a datos de diversas fuentes.

Un almacén de los datos es un repositorio de datos recopilados de múltiples fuentes y almacenados bajo un esquema unificado de la base de datos. Los datos almacenados en almacén se analizan mediante agregaciones complejas y análisis estadísticos. Además, las técnicas de descubrimiento del conocimiento se pueden utilizar para intentar descubrir reglas y patrones de datos. Por ejemplo, un minorista puede descubrir que ciertos productos tienden a comprarse juntos, y puede utilizar esa información para desarrollar estrategias de marketing. Este proceso del descubrimiento del conocimiento se llama *minería de datos*. El Capítulo 18 trata estos aspectos.

En nuestras estudios hasta el momento nos hemos centrado en datos relativamente simples y bien estructurados. Sin embargo, hay una cantidad enorme de datos textuales no estructurados en Internet, intranets en organizaciones y en las computadoras de usuarios individuales. Los usuarios desean encontrar la información relevante de esta gran fuente de información textual fundamentalmente, usando mecanismos simples de consulta como consultas basadas en palabras clave. El campo de la recuperación de información se ocupa de consultar estos datos no estructurados, y presta atención particular a la clasificación de los resultados de la consulta. Aunque este campo de investigación recorre varias décadas, ha experimentado un gran crecimiento con el desarrollo de World Wide Web. El Capítulo 19 proporciona una introducción al campo de la recuperación de información.

CAPÍTULO 18

Análisis y minería de datos

Las empresas han comenzado a aprovechar los cada vez más numerosos datos en línea para tomar mejores decisiones sobre sus actividades, como los artículos que deben tener en inventario y el modo de dirigirse mejor a los clientes para aumentar las ventas. Muchas de las consultas, sin embargo, son bastante complejas y algunos tipos de información no pueden extraerse ni siquiera empleando SQL.

Se encuentran disponibles varias técnicas y herramientas de ayuda a la toma de decisiones. Existen herramientas para el análisis de datos que permiten a los analistas ver los datos de diferentes formas. Otras herramientas de análisis realizan un cálculo previo de resúmenes de cantidades muy grandes de datos con objeto de dar respuestas rápidas a las consultas. Las normas SQL:1999 y SQL:2003 contienen ahora constructores adicionales para soportar el análisis de datos. Otro enfoque para obtener información de los datos es utilizar la *minería de datos*, que pretende detectar varios tipos de estructuras en grandes volúmenes de datos. La minería de datos complementa varios tipos de técnicas estadísticas con objetivos parecidos.

Este capítulo trata sobre la ayuda a la toma de decisiones, incluidos el procesamiento analítico en línea, los almacenes de datos y la minería de datos.

18.1 Sistemas de ayuda a la toma de decisiones

Las aplicaciones de bases de datos pueden clasificarse grosso modo en sistemas de procesamiento de transacciones y de ayuda a la toma de decisiones. Los sistemas de procesamiento de transacciones son sistemas que registran información sobre las transacciones, como ventas de productos o matrículas e información de titulaciones para las universidades. Los sistemas de procesamiento de transacciones se utilizan mucho hoy en día y las empresas han acumulado una enorme cantidad de información generada por ellos. Los sistemas de ayuda a la toma de decisiones facilitan a los gestores la decisión de los productos que se deben almacenar en una tienda, los productos que es necesario fabricar o las personas que se deberían admitir en una universidad.

Por ejemplo, las bases de datos de las empresas suelen contener enormes cantidades de información sobre los clientes y las transacciones. La cantidad de información necesaria puede llegar a varios centenares de gigabytes o, incluso, a los terabytes para las cadenas de grandes almacenes. La información de las transacciones de un gran almacén puede incluir el nombre o identificador (como puede ser el número de la tarjeta de crédito) del cliente, los artículos adquiridos, el precio pagado y las fechas en que se realizaron las compras. La información sobre los artículos adquiridos puede incluir el nombre del artículo, el fabricante, el número del modelo, el color y la talla. La información sobre los clientes puede incluir su historial de crédito, sus ingresos anuales, su domicilio, su edad e, incluso, su nivel académico.

Estas bases de datos de gran tamaño pueden resultar minas de información para adoptar decisiones empresariales, como los artículos que debe haber en inventario y los descuentos que hay que ofrecer. Por ejemplo, puede que una cadena de grandes almacenes note un aumento súbito de las compras de

camisas de franela en la Sierra de Guadarrama, darse cuenta de que hay una tendencia y comenzar a almacenar un mayor número de esas camisas en las tiendas de esa zona. O puede que una empresa automovilística descubra, al consultar su base de datos, que la mayor parte de los coches deportivos de pequeño tamaño los compran mujeres jóvenes cuyos ingresos anuales superan los 50.000 €. Puede que la empresa dirija su publicidad para que atraiga más mujeres de esas características a que compren coches deportivos de pequeño tamaño y evite desperdiciar dinero intentando atraer a otras categorías de consumidores para que compren esos coches. En ambos casos la empresa ha identificado pautas de comportamiento de los consumidores y las ha utilizado para adoptar decisiones empresariales.

El almacenamiento y recuperación de los datos para la ayuda a la toma de decisiones plantea varios problemas:

- Aunque muchas consultas para ayuda a la toma de decisiones pueden escribirse en SQL, otras no pueden expresarse en SQL o no pueden hacerlo con facilidad. En consecuencia, se han propuesto varias extensiones de SQL para facilitar el análisis de los datos. El área de *procesamiento analítico en línea (Online Analytical Processing*, OLAP) trata de las herramientas y de las técnicas para el análisis de datos que pueden dar respuestas casi instantáneas a las consultas de datos resumidos, aun cuando la base de datos sea extremadamente grande. En el Apartado 18.2 se estudian las extensiones de SQL para el análisis de datos y las técnicas para el procesamiento analítico en línea.
- Los lenguajes de consultas de bases de datos no son eficaces para el **análisis estadístico** detallado de datos. Existen varios paquetes, como SAS y S++, que ayudan en el análisis estadístico. A estos paquetes se les han añadido interfaces con las bases de datos para permitir que se almacenen en ellas grandes volúmenes de datos y se recuperen de manera eficiente para su análisis. El campo del análisis estadístico es una gran disciplina por sí misma, véanse las referencias en las notas bibliográficas para obtener más información.
- Las grandes empresas tienen varios orígenes de datos que necesitan utilizar para adoptar decisiones empresariales. Los orígenes pueden almacenar los datos según diferentes esquemas. Por motivos de rendimiento (así como por motivos de control de la organización) los orígenes de datos no suelen permitir que otras partes de la empresa recuperen datos a petición.

  Para ejecutar de manera eficiente las consultas sobre datos tan diferentes las empresas han creado *almacenes de datos*. Los almacenes de datos reúnen los datos de varios orígenes bajo un esquema unificado en un solo sitio. Por tanto, ofrecen al usuario una sola interfaz uniforme para los datos. Los problemas de la creación y mantenimiento de los almacenes de datos se estudian en el Apartado 18.3.
- Las técnicas de descubrimiento de conocimiento intentan determinar de manera automática reglas estadísticas y patrones a partir de los datos. El campo de la *minería de datos* combina las técnicas de descubrimiento de conocimiento creadas por los investigadores en inteligencia artificial y los analistas estadísticos con las técnicas de implementación eficiente que permiten utilizarlas en bases de datos extremadamente grandes. El Apartado 18.4 estudia la minería de datos.

El área de **ayuda a la toma de decisiones** puede considerarse que abarca todas las áreas anteriores, aunque algunas personas utilizan el término en un sentido más restrictivo que excluye el análisis estadístico y la minería de datos.

18.2 Análisis de datos y OLAP

Aunque es mejor dejar el análisis estadístico complejo a los paquetes estadísticos las bases de datos deben soportar las formas sencillas, utilizadas frecuentemente, de análisis estadístico. Dado que los datos almacenados en las bases de datos suelen ser de gran volumen, hay que resumirlos de algún modo si hay que obtener información que puedan utilizar los usuarios.

Las herramientas OLAP soportan el análisis interactivo de la información de resumen. Se han desarrollado varias extensiones de SQL para soportar las herramientas OLAP. Hay muchas tareas utilizadas con frecuencia que no pueden realizarse empleando las facilidades básicas de agregación y agrupamiento

talla: **all**

nombre_artículo	*color*			
	oscuro	pastel	blanco	Total
falda	8	35	10	53
vestido	20	10	5	35
camisa	14	7	28	49
pantalón	20	2	5	27
Total	62	54	48	164

Figura 18.1 Tabulación cruzada de *ventas* con *nombre_artículo* y *color*.

de SQL. Entre los ejemplos se hallan la búsqueda de percentiles, las distribuciones acumulativas o los agregados sobre ventanas deslizantes de datos ordenados secuencialmente. Recientemente se han propuesto varias extensiones de SQL para soportar estas tareas y se han implementado en productos como Oracle y DB2 de IBM.

18.2.1 Procesamiento analítico en línea

El análisis estadístico suele necesitar el agrupamiento de varios atributos. Considérese una aplicación en que una tienda desea averiguar las prendas que son más populares. Supóngase que las prendas están caracterizadas por su nombre de artículo, su color y su talla y que se tiene la relación *ventas* con el esquema *ventas* (*nombre_artículo, color, talla, número*). Supóngase que *nombre_artículo* puede adoptar los valores (falda, vestido, camisa, pantalón), *color* puede adoptar los valores (oscuro, pastel, blanco) y *talla* puede adoptar los valores (pequeña, mediana, grande).

Dada una relación utilizada para el análisis de datos se pueden identificar algunos de sus atributos como **atributos de medida**, ya que miden algún valor y pueden agregarse. Por ejemplo, el atributo *número* de la relación *ventas* es un atributo de medida, ya que mide la cantidad de unidades vendidas. Algunos de los demás atributos (o todos ellos) de la relación se identifican como **atributos de dimensión**, ya que definen las dimensiones en las que se ven los atributos de medida y los resúmenes de los atributos de medida. En la relación *ventas*, *nombre_artículo, color* y *talla* son atributos de dimensión. (Una versión más realista de la relación *ventas* tendría más dimensiones, como tiempo o lugar de venta, y más medidas como el valor monetario de la venta).

Los datos que pueden modelarse como atributos de dimensión y como atributos de medida se denominan **datos multidimensionales**.

Para analizar los datos multidimensionales puede que el administrador desee ver los datos dispuestos como se encuentran en la tabla de la Figura 18.1. La tabla muestra las cifras totales de diferentes combinaciones de *nombre_artículo* y *color*. El valor de *talla* se especifica como **todas**, lo que indica que los valores mostrados son un resumen para todos los valores de *talla*.

La tabla de la Figura 18.1 es un ejemplo de **tabulación cruzada**, también denominada **tabla dinámica**. En general, las tabulaciones cruzadas son tablas en las que los valores de uno de los atributos (por ejemplo, A) forman las cabeceras de las filas, los valores del otro atributo (por ejemplo, B) forman las cabeceras de las columnas y los valores de cada celda se obtienen como sigue: cada celda puede identificarse como (a_i, b_j), donde a_i es un valor de A, y b_j es un valor de B. Si hay como máximo una tupla con cualquier valor de (a_i, b_j), el valor de la celda se obtiene de esa única tupla (si es que hay alguna); por ejemplo, puede ser el valor de uno o varios atributos de la tupla. Si puede haber varias tuplas con el valor (a_i, b_j), el valor de la celda debe obtenerse por agregación de las tuplas con ese valor. En este ejemplo la agregación utilizada es la suma de los valores del atributo *número* para todos los valores de *talla*, como se indica por *talla:* **all** en la tabla cruzada de la Figura 18.1. En este ejemplo la tabulación cruzada también tiene una columna y una fila adicionales que guardan los totales de las celdas de cada fila o columna. La mayor parte de las tabulaciones cruzadas tienen esas filas y columnas de resumen.

Las tabulaciones cruzadas son diferentes de las tablas relacionales que suelen guardarse en las bases de datos, ya que el número de columnas de la tabulación cruzada depende de los datos. Una modificación en los valores de los datos puede dar lugar a que se añadan más columnas, lo que no resulta

nombre_artículo	*color*	*talla*	*número*
falda	oscuro	**all**	8
falda	pastel	**all**	35
falda	blanco	**all**	10
falda	**all**	**all**	53
vestido	oscuro	**all**	20
vestido	pastel	**all**	10
vestido	blanco	**all**	5
vestido	**all**	**all**	35
camisa	oscuro	**all**	14
camisa	pastel	**all**	7
camisa	blanco	**all**	28
camisa	**all**	**all**	49
pantalón	oscuro	**all**	20
pantalón	pastel	**all**	2
pantalón	blanco	**all**	5
pantalón	**all**	**all**	27
all	oscuro	**all**	62
all	pastel	**all**	54
all	blanco	**all**	48
all	**all**	**all**	164

Figura 18.2 Representación relacional de los datos de la Figura 18.1.

deseable para el almacenamiento de los datos. No obstante, la vista de tabulación cruzada es deseable para mostrársela a los usuarios. La representación de las tabulaciones cruzadas sin valores resumen en un formulario relacional con un número fijo de columnas es directa. La tabulación cruzada con columnas o filas resumen puede representarse introduciendo el valor especial **todos** para representar los subtotales, como en la Figura 18.2. La norma SQL:1999 utiliza realmente el valor **null** (nulo) en lugar de **all**; pero, para evitar confusiones con los valores nulos habituales, en el libro se seguirá utilizando **all**.

Considérense las tuplas (falda, **all**, **all**, 53) y (vestido, **all**, **all**, 35). Se han obtenido estas tuplas eliminando las tuplas individuales con diferentes valores de *color* y *talla*, y sustituyendo el valor de *número* por un agregado—es decir, una suma. El valor **all** puede considerarse representante del conjunto de los valores de un atributo. Las tuplas con el valor **all** para las dimensiones *color* y *talla* pueden obtenerse mediante una agregación de la relación *ventas* con una agrupación en la columna *nombre_artículo*. De manera parecida, se puede utilizar una agrupación en *color* y *talla* para conseguir las tuplas con el valor **all** para *nombre_artículo*, y se puede utilizar una agrupación sin atributo alguno (que en SQL puede omitirse simplemente) para obtener la tupla con el valor **all** para *nombre_artículo*, *color* y *talla*.

La generalización de las tabulaciones cruzadas, que son bidimensionales, a n dimensiones pueden visualizarse como cubos n-dimensionales, denominados **cubos de datos**. La Figura 18.3 muestra un cubo de datos para la relación *ventas*. El cubo de datos tiene tres dimensiones, a saber, *nombre_artículo*, *color* y *talla*, y el atributo de medida es *número*. Cada celda se identifica por los valores de estas tres dimensiones. Cada celda del cubo de datos contiene un valor, igual que en la tabulación cruzada. En la Figura 18.3, el valor contenido en la celda se muestra en una de las caras de la celda; las otras caras de la celda se muestran en blanco si son visibles. Todas las celdas contienen valores, aunque no sean visibles.

El valor de una dimensión puede ser **all**, en cuyo caso la celda contiene un resumen de todos los valores de esa dimensión, como en el caso de las tabulaciones cruzadas. El número de maneras diferentes en que las tuplas pueden agruparse para su agregación puede ser grande. De hecho, para una tabla con n dimensiones, se puede realizar la agregación con la agrupación de cada uno de los 2^n subconjuntos de las n dimensiones[1].

El sistema de procesamiento analítico en línea o sistema OLAP es un sistema interactivo que permite a los analistas ver diferentes resúmenes de los datos multidimensionales. Las palabras *en línea* indican que

1. La agrupación sobre el conjunto de las n dimensiones sólo resulta útil si la tabla puede tener duplicados.

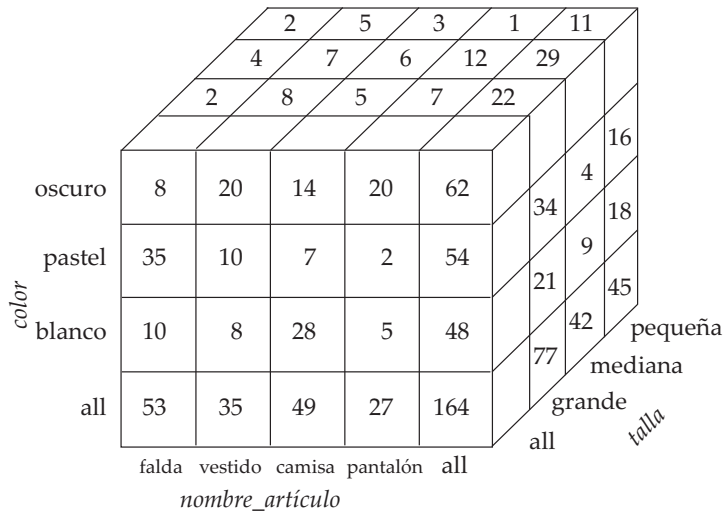


Figura 18.3 Cubo de datos tridimensional.

los analistas deben poder solicitar nuevos resúmenes y obtener respuestas en línea, en pocos segundos, y no deberían verse obligados a esperar mucho tiempo para ver el resultado de las consultas.

Con los sistemas OLAP los analistas de datos pueden ver diferentes tabulaciones cruzadas para los mismos datos seleccionando de manera interactiva los atributos de las tabulaciones cruzadas. Cada tabulación cruzada es una vista bidimensional del cubo de datos multidimensional. Por ejemplo, el analista puede seleccionar una tabulación cruzada para *nombre_artículo* y *talla* o una tabulación cruzada para *color* y *talla*. La operación de modificación de las dimensiones utilizadas en las tabulaciones cruzadas se denomina **pivotaje**.

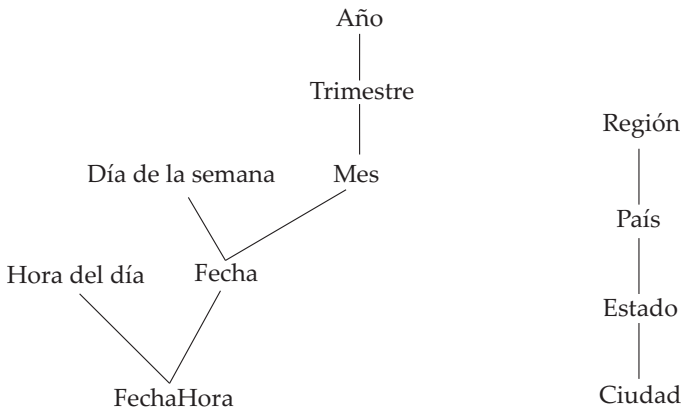
Los sistemas OLAP también ofrecen otras funcionalidades. Por ejemplo, puede que un analista desee ver una tabulación cruzada de *nombre_artículo* y *color* para un valor fijo de *talla*, por ejemplo, grande, en lugar de la suma para todas las tallas. Esta operación se conoce como **corte**, ya que puede considerarse como la vista de una rebanada del cubo de datos. A veces la operación se denomina **corte de cubos**, especialmente cuando se fijan los valores de varias dimensiones.

Cuando se utilizan tabulaciones cruzadas para ver cubos multidimensionales los valores de los atributos de dimensión que no forman parte de la tabulación cruzada se muestran por encima de ella. El valor de estos atributos puede ser **all**, como puede verse en la Figura 18.1, lo que indica que los datos de la tabulación cruzada son un resumen de todos los valores del atributo. El corte y la creación de datos consisten simplemente en la selección de valores concretos de estos atributos, que se muestran luego por encima de la tabulación cruzada.

Los sistemas OLAP permiten a los usuarios examinar los datos con cualquier nivel de granularidad que se desee. La operación de pasar de datos con una granularidad más fina a una granularidad más gruesa (mediante la agregación) se denomina **abstracción**. En este ejemplo, a partir del cubo de datos para la tabla *ventas*, se obtiene la tabulación cruzada de ejemplo abstrayendo el atributo *talla*. La operación inversa—la de pasar de datos con una granularidad más gruesa a una más fina—se denomina **concreción**. Claramente, los datos con granularidad más fina no pueden generarse a partir datos con una granularidad más gruesa; deben generarse a partir de los datos originales o de datos resumidos de granularidad aún más fina.

Puede que un analista desee examinar una dimensión con niveles diferentes de detalle. Por ejemplo, los atributos de tipo **FechaHora** contienen una fecha y una hora del día. El empleo de horas con precisión de segundos (o menos) puede que no sea significativo: los analistas que estén interesados en la hora aproximada del día puede que sólo miren el valor horario. Los analistas que estén interesados en las ventas de cada día de la semana puede que apliquen la fecha al día de la semana y sólo se fijen en eso. Puede que otro analista esté interesado en agregados mensuales, trimestrales o de años enteros.

Los diferentes niveles de detalle de los atributos pueden organizarse en una **jerarquía**. La Figura 18.4a muestra una jerarquía para el atributo **FechaHora**. La Figura 18.4b, que puede ser otro ejem-



(a) Jerarquía de tiempo

(b) Jerarquía de ubicación

Figura 18.4 Jerarquías de las dimensiones.

plo, muestra una jerarquía para la ubicación, con la ciudad en la parte inferior de la jerarquía, el estado por encima, el país en el nivel siguiente y la región en el nivel superior. En el ejemplo anterior la ropa puede agruparse por categorías (por ejemplo, ropa de hombre o de mujer); la *categoría* estaría por encima de *nombre-artículo* en la jerarquía de la ropa. En el nivel de los valores reales las faldas y los vestidos caerían dentro de la categoría de ropa de mujer y los pantalones y las camisas en la de ropa de hombre.

Puede que un analista esté interesado en consultar las ventas de ropa divididas entre ropa de hombre y ropa de mujer y que no esté interesado en sus valores individuales. Tras ver los agregados en el nivel de ropa de hombre y ropa de mujer puede que el analista *concrete la jerarquía* para ver los valores individuales. Un analista que examine el nivel detallado puede *abstraer la jerarquía* y examinar agregados de niveles más gruesos. Ambos niveles pueden mostrarse en la misma tabulación cruzada, como en la Figura 18.5.

18.2.2 Implementación de OLAP

Los primeros sistemas de OLAP utilizaban arrays de memoria multidimensionales para almacenar los cubos de datos y se denominaban sistemas **OLAP multidimensionales (Multidimensional OLAP, MOLAP)**. Posteriormente, los servicios OLAP se integraron en los sistemas relacionales y los datos se almacenaron en las bases de datos relacionales. Estos sistemas se denominan sistemas **OLAP relacionales (Relational OLAP, ROLAP)**. Los sistemas híbridos, que almacenan algunos resúmenes en la memoria y los datos básicos y otros resúmenes en bases de datos relacionales, se denominan sistemas **OLAP híbridos (Hybrid OLAP, HOLAP)**.

talla: **all**

categoría	*nombre_artículo*	*color*			
		oscuro	pastel	blanco	total
ropa de mujer	falda	8	8	10	53
	vestido	20	20	5	35
	subtotal	28	28	15	88
ropa de hombre	pantalón	14	14	28	49
	camisa	20	20	5	27
	subtotal	34	34	33	76
total		62	62	48	164

Figura 18.5 Tabulación cruzada de *ventas* con la jerarquía para *nombre_artículo*.

Muchos sistemas OLAP se implementan como sistemas cliente–servidor. El servidor contiene la base de datos relacional y los cubos de datos MOLAP. Los sistemas clientes obtienen vistas de los datos comunicándose con el servidor.

Una manera ingenua de calcular todo el cubo de datos (todas las agrupaciones) de una relación es utilizar cualquier algoritmo estándar para calcular las operaciones de agregación, agrupación a agrupación. El algoritmo ingenuo necesita un gran número de exploraciones de la relación. Una optimización sencilla consiste en calcular el agregado para, por ejemplo, (*nombre_artículo*, *color*) a partir del agregado (*nombre_artículo*, *color*, *talla*), en lugar de hacerlo a partir de la relación original.

Para las funciones de agregación estándar de SQL se pueden calcular agregados con agrupaciones sobre un conjunto de atributos A a partir de un agregado con agrupación sobre un conjunto de atributos B si $A \subseteq B$; se puede hacer como ejercicio (véase el Ejercicio 18.1), pero hay que tener en cuenta que al calcular **avg** también es necesario el valor **count** (para algunas funciones de agregación no estándar como la mediana, los agregados no pueden calcularse de la manera indicada; la optimización aquí descrita no es aplicable a estas funciones de agregación *no-descomponibles*). La cantidad de datos que se lee disminuye de manera significativa al calcular los agregados a partir de otros agregados, en lugar de hacerlo a partir de la relación original. Se pueden conseguir otras mejoras; por ejemplo, se pueden calcular varias agrupaciones con una sola lectura de los datos. Véanse las notas bibliográficas para hallar referencias a los algoritmos para el cálculo eficiente de cubos de datos.

Las primeras implementaciones OLAP calculaban previamente los cubos de datos completos, es decir, las agrupaciones por todos los subconjuntos de los atributos de dimensión, y los almacenaban. El cálculo previo permite que las consultas OLAP se respondan en pocos segundos, incluso para conjuntos de datos que pueden contener millones de tuplas que suponen gigabytes de datos. No obstante, hay 2^n agrupaciones con n atributos de dimensión; las jerarquías de los atributos aumentan más aún el número. En consecuencia, todo el cubo de datos suele ser mayor que la relación original que lo generó y, en muchos casos, no resulta posible almacenarlo entero.

En lugar de calcular previamente todas las agrupaciones posibles y almacenarlas, resulta razonable calcular previamente algunas de las agrupaciones y almacenarlas y calcular el resto según se soliciten. En lugar de calcular las consultas a partir de la relación original, lo que puede tardar mucho tiempo, se pueden calcular a partir de otras consultas calculadas previamente. Por ejemplo, supóngase que una consulta necesita resúmenes según (*nombre_artículo*, *color*), que no se ha calculado con anterioridad. El resultado de la consulta puede calcularse a partir de resúmenes según (*nombre_artículo*, *color*, *talla*), si ya se ha calculado. Véanse las notas bibliográficas para hallar referencias al modo de seleccionar para su cálculo previo un buen conjunto de agrupaciones, dados los límites de almacenamiento disponible para los resultados calculados previamente.

Los datos de los cubos no se pueden generar mediante una sola consulta SQL utilizando las estructuras básicas **group by**, ya que los agregados se calculan para varias agrupaciones diferentes de los atributos de dimensión. El Apartado 18.2.3 estudia las extensiones de SQL para el soporte de la funcionalidad OLAP.

18.2.3 Agregación extendida

La funcionalidad de agregación de SQL-92 está limitada, por lo que diferentes sistemas de bases de datos han implementado varias extensiones. No obstante, la norma SQL:1999 define un amplio conjunto de funciones de agregación, que se describen en este apartado y en los dos siguientes. Las bases de datos de Oracle y de DB2 de IBM soportan la mayor parte de estas características y, sin duda, otras bases de datos soportarán estas características en un futuro próximo.

Las nuevas funciones de agregación para un solo atributo son la desviación estándar y la varianza (**stddev** y **variance**). La desviación estándar es la raíz cuadrada de la varianza[2]. Algunos sistemas de bases de datos soportan otras funciones de agregación como la mediana y la moda. Algunos sistemas de bases de datos incluso permiten que los usuarios añadan nuevas funciones de agregación.

2. La norma SQL:1999 soporta en realidad dos tipos de varianza, denominadas *varianza de la población* y *varianza de la muestra* y, por tanto, dos tipos de desviación estándar. La definición de los dos tipos difiere ligeramente; los detalles se pueden consultar en cualquier libro de texto de estadística.

SQL:1999 también soporta una nueva clase de **funciones de agregación binarias**, que pueden calcular resultados estadísticos para parejas de atributos; entre ellas están las correlaciones, las covarianzas y las curvas de regresión, que dan una línea que aproxima la relación entre los valores de la pareja de atributos. Las definiciones de estas funciones pueden hallarse en cualquier libro de texto estándar, como los que se citan en las notas bibliográficas.

SQL:1999 también soporta generalizaciones de la estructura **group by**, mediante las estructuras **cube** y **rollup**. Un uso representativo de la estructura **cube** es el siguiente:

select *nombre_artículo, color, talla*, **sum**(*número*)
from *ventas*
group by cube(*nombre_artículo, color, talla*)

Esta consulta calcula la unión de ocho agrupaciones diferentes de la relación *ventas*:

{ (*nombre_artículo, color, talla*), (*nombre_artículo, color*), (*nombre_artículo, talla*),
(*color, talla*), (*nombre_artículo*), (*color*), (*talla*), () }

donde () denota una lista **group by** vacía.

Para cada agrupación el resultado contiene el valor nulo para los atributos no presentes en la agrupación. Por ejemplo, la tabla de la Figura 18.2, sustituyendo **all** por *null*, la puede calcular la consulta

select *nombre_artículo, color*, **sum**(*número*)
from *ventas*
group by cube(*nombre_artículo, color*)

Una estructura **rollup** representativa es:

select *nombre_artículo, color, talla*, **sum**(*número*)
from *ventas*
group by rollup(*nombre_artículo, color, talla*)

En este caso sólo se han generado cuatro agrupaciones:

{ (*nombre_artículo, color, talla*), (*nombre_artículo, color*), (*nombre_artículo*), () }

La instrucción rollup puede utilizarse para generar agregados en varios niveles de una jerarquía para una columna. Por ejemplo, supóngase que se tiene la tabla *categoríaartículo(nombre_artículo, categoría)* que da la categoría de cada artículo. La consulta

select *categoría, nombre_artículo*, **sum**(*número*)
from *ventas*, *categoríaartículo*
where *ventas.nombre_artículo = categoríaartículo.nombre_artículo*
group by rollup(*categoría, nombre_artículo*)

da un resumen jerárquico según *nombre_artículo* y según *categoría*.

Se pueden utilizar varios **rollup** y varios **cube** en una sola cláusula **group by**. Por ejemplo, la consulta siguiente:

select *nombre_artículo, color, talla*, **sum**(*número*)
from *ventas*
group by rollup(*nombre_artículo*), **rollup**(*color, talla*)

genera las agrupaciones

{ (*nombre_artículo, color, talla*), (*nombre_artículo, color*), (*nombre_artículo*),
(*color, talla*), (*color*), () }

Para comprender el motivo hay que tener en cuenta que **rollup**(*nombre_artículo*) genera dos agrupaciones, {(*nombre_artículo*), ()}, y que **rollup**(*color, talla*) genera tres agrupaciones, {(*color, talla*), (*color*), () }. El producto cartesiano de los dos da como resultado las seis agrupaciones mostradas.

Como ya se ha mencionado en el Apartado 18.2.1, SQL:1999 utiliza el valor **null** para indicar el sentido habitual de nulo así como **all**. Este uso dual de **null** puede generar ambigüedad si los atributos utilizados en una cláusula rollup o en una cláusula cube contienen valores nulos. Se puede aplicar la función **grouping** a un atributo; devuelve 1 si el valor es un valor nulo que represente a **all**, y devuelve 0 en los demás casos. Considérese la consulta siguiente:

```
select nombre_artículo, color, talla, sum(número),
    grouping(nombre_artículo) as indicador_nombre_artículo,
    grouping(color) as indicador_color,
    grouping(talla) as indicador_talla
from ventas
group by cube(nombre_artículo, color, talla)
```

El resultado es el mismo que en la versión de la consulta sin **grouping**, pero con tres columnas adicionales denominadas *indicador_nombre_artículo, indicador_color* e *indicador_talla*. En cada tupla el valor de los campos indicador es 1 si el campo correspondiente es un valor nulo que representa a **all**.

En lugar de utilizar etiquetas para indicar los valores nulos que representan a **all**, se pueden sustituir los valores nulos por un valor a la elección del usuario:

```
decode(grouping(nombre_artículo), 1, 'todos', nombre_artículo)
```

Esta expresión devuelve el valor "todos" si el valor de *nombre_artículo* es un valor nulo que se corresponda con **all**, y devuelve el valor real de *nombre_artículo* en caso contrario. Esta expresión puede utilizarse en lugar de *nombre_artículo* en la cláusula select para obtener "todos" en el resultado de la consulta, en lugar de valores nulos que representen a **all**.

Ni la cláusula **rollup** ni la cláusula **cube** ofrecen un control completo de las agrupaciones que se generan. Por ejemplo, no se pueden utilizar para especificar que sólo se desean las agrupaciones {(*color, talla*), (*talla, nombre_artículo*)}. Estas agrupaciones restringidas pueden generarse utilizando la estructura **grouping** en la cláusula **having**; los detalles se dejan como ejercicio para el lector.

18.2.4 Clasificación

Hallar la posición de un valor en un conjunto más grande es una operación frecuente. Por ejemplo, puede que se desee asignar una clasificación a los estudiantes de acuerdo con sus notas totales, con el puesto 1 para el estudiante con las notas más altas, el puesto 2 para el estudiante con las segundas mejores notas, etc. Aunque estas consultas pueden expresarse en SQL-92, resultan difíciles de expresar e ineficientes a la hora de la evaluación. Los programadores suelen recurrir a escribir en parte la consulta en SQL y en parte en un lenguaje de programación. Un tipo de consulta relacionado es la búsqueda del percentil que le corresponde a un valor de un (multi)conjunto, por ejemplo, el tercio inferior, el tercio central o el tercio superior. Aquí se estudiará el soporte de SQL:1999 para estos tipos de consultas.

La clasificación se realiza conjuntamente con una especificación **order by**. Supóngase que se tiene una relación *notas_estudiante(id_estudiante, notas)* que almacena las notas obtenidas por cada estudiante. La consulta siguiente da la clasificación de cada estudiante.

```
select id_estudiante, rank() over (order by (notas) desc) as clasificación_e
from notas_estudiante
```

Obsérvese que el orden de las tuplas en el resultado no se ha definido, por lo que puede que no estén ordenadas según su clasificación. Se necesita una cláusula **order by** adicional para dejarlas ordenadas, como puede verse a continuación.

```
select id_estudiante, rank () over (order by (notas) desc) as clasificación_e
from notas_estudiante order by clasificación_e
```

Un aspecto básico de las clasificaciones es el modo de tratar el caso de que haya varias tuplas que sean iguales en el atributo o atributos de ordenación. En el ejemplo anterior esto significa decidir lo que se hace si hay dos estudiantes con la misma nota. La función **rank** devuelve la misma clasificación a todas las tuplas que sean iguales en los atributos **order by**. Por ejemplo, si la nota más elevada la comparten dos estudiantes, los dos obtendrían el puesto 1. El puesto siguiente sería el 3, no el 2, por lo que si tres estudiantes consiguieran la siguiente nota más alta, todos ellos obtendrían el puesto 3, y los siguientes estudiantes obtendrían el puesto 5, etc. También hay una función **dense_rank** que no crea saltos en la ordenación. En el ejemplo anterior las tuplas con el segundo valor más alto obtienen el puesto 2 y las tuplas con el tercer valor más elevado obtienen el puesto 3, etc. Se puede llevar a cabo la clasificación dentro de particiones de los datos. Por ejemplo, supóngase que se tiene una relación adicional *estudiante_sección(id_estudiante, sección)* que almacena para cada estudiante la sección en la que estudia. La consulta siguiente da la clasificación de los estudiantes dentro de cada sección.

```
select id_estudiante, sección,
    rank () over (partition by sección order by notas desc) as clasificación_sec
from notas_estudiante, sección_estudiante
where notas_estudiante.id_estudiante = sección_estudiante.id_estudiante
order by sección, clasificación_sec
```

La cláusula **order by** externa ordena las tuplas del resultado por secciones y, dentro de cada sección, por su clasificación.

Se pueden utilizar varias expresiones **rank** dentro de cada sentencia **select**; por tanto, se puede obtener la clasificación general y la clasificación dentro de cada sección empleando dos expresiones **rank** en la misma cláusula **select**. Una pregunta interesante es lo que ocurre cuando se produce una clasificación (posiblemente con partición) junto con una cláusula **group by**. En ese caso, la cláusula **group by** se aplica en primer lugar y la partición y la clasificación se realizan sobre los resultados de la cláusula **group by**. Así, los valores agregados pueden utilizarse para la clasificación. Por ejemplo, supóngase que se tienen las notas de cada estudiante en varias asignaturas. Para clasificar a los estudiantes por la suma de sus notas en varias asignaturas se puede utilizar una cláusula **group by** para calcular las notas agregadas de cada estudiante y luego clasificar a los estudiantes por la suma agregada. Los detalles se le dejan al lector como ejercicio.

Las funciones de clasificación pueden utilizarse para hallar las n primeras tuplas incrustando una consulta de clasificación en una consulta de un nivel exterior; los detalles se dejan para un ejercicio. Téngase en cuenta que las n últimas tuplas son simplemente las mismas que las n primeras con un orden inverso. Varios sistemas de bases de datos ofrecen extensiones no estándar de SQL para especificar directamente que sólo se necesitan los n primeros resultados; esas extensiones no necesitan la función de clasificación y simplifican el trabajo del optimizador, pero (actualmente) no son tan generales, ya que no soportan las particiones.

SQL:1999 también especifica otras funciones que pueden utilizarse en lugar de **rank**. Por ejemplo, **percent_rank** de una tupla da la clasificación de la tupla en forma de fracción. Si hay n tuplas en la partición[3] y la clasificación de la tupla es r, su clasificación percentual se define como $(r-1)/(n-1)$ (y como nula si sólo hay una tupla en la partición). La función **cume_dist**, abreviatura de distribución acumulativa (cumulative distribution), para una tupla se define como p/n, donde p es el número de tuplas de la partición con valores de ordenación que preceden o son iguales al valor de ordenación de la tupla, y n es el número de tuplas de la partición. La función **row_number** ordena las filas y da a cada una un número único correspondiente a su posición en el orden; filas diferentes con el mismo valor de ordenación reciben números de fila diferentes, de manera no determinista.

Finalmente, para una constante dada n, la función de clasificación **ntile**(n) toma las tuplas de cada partición en el orden especificado y las divide en n cajones con igual número de tuplas[4]. Para cada tupla, **ntile**(n) da el número del cajón en el que se halla, con los números de los cajones comenzando por 1. Esta

3. Todo el conjunto se trata como una sola partición si no se utiliza ninguna partición explícita.
4. Si el número total de tuplas de una partición no es divisible por n, el número de tuplas de cada cajón puede variar como mucho en 1. Las tuplas con el mismo valor del atributo de ordenación pueden asignarse cajones diferentes, de manera no determinista, para hacer igual el número de tuplas de cada cajón.

función resulta especialmente útil para la creación de histogramas basados en percentiles. Por ejemplo, se pueden ordenar los empleados por su salario y utilizar **ntile**(3) para hallar el rango (tercio inferior, tercio central o tercio superior) en el que se halla cada empleado, y para calcular el salario total ganado por los empleados de cada rango:

```
select tercil, sum(sueldo)
from (
    select sueldo, ntile(3) over (order by (sueldo)) as tercil
    from empleado) as s
group by tercil
```

La presencia de valores nulos puede complicar la definición de la clasificación, dado que no está claro si deben colocarse antes en el orden. SQL:1999 permite que el usuario especifique dónde deben aparecer mediante **nulls first** o **nulls last**, por ejemplo:

```
select id_estudiante, rank () over (order by notas desc nulls last) as clasificación_e
from notas_estudiante
```

18.2.5 Ventanas

Un ejemplo de consulta *ventana* es una consulta que, dados los valores de ventas para cada fecha, calcula para cada fecha el promedio de ventas de ese día, del día anterior y del día siguiente; esas consultas de media móvil se utilizan para suavizar las variaciones aleatorias. Otro ejemplo de consulta ventana es la consulta que calcula el saldo acumulado de una cuenta, dada una relación que especifique las imposiciones y las retiradas de fondos de la cuenta. Esas consultas resultan difíciles o imposibles de expresar (depende de la consulta en concreto) en SQL básico.

SQL:1999 ofrece una característica de ventanas para soportar esas consultas. A diferencia de **group by**, la misma tupla puede estar en varias ventanas. Supóngase que se tiene la relación *transacción* (*número_cuenta*, *fecha_hora*, *valor*), donde *valor* es positivo para las imposiciones de fondos y negativo para las retiradas. Se da por supuesto que hay como máximo una transacción por cada valor *fecha_hora*.

Considérese la consulta

```
select número_cuenta, fecha_hora,
    sum(valor) over
        (partition by número_cuenta
         order by fecha_hora
         rows unbounded preceding)
    as saldo
from transacción
order by número_cuenta, fecha_hora
```

La consulta da los saldos acumulados de cada cuenta justo antes de cada transacción en esa cuenta; el saldo acumulado de una cuenta es la suma de valores de todas las transacciones anteriores de la cuenta.

La cláusula **partition by** separa las tuplas por número de cuenta, de modo que para cada fila sólo se consideren las tuplas de su partición. Se crea una ventana para cada tupla; las palabras clave **rows unbounded preceding** especifican que la ventana de cada tupla consiste en todas las tuplas de la partición que la preceden en el orden especificado (en este caso, orden creciente de *fecha_hora*). La función de agregación **sum**(*valor*) se aplica a todas las tuplas de la ventana. Obsérvese que la consulta no utiliza ninguna cláusula **group by**, ya que hay una tupla de resultado por cada tupla de la relación *transacción*.

Aunque la consulta puede escribirse sin las estructuras ampliadas, sería bastante difícil de formular. Obsérvese también que se pueden superponer diferentes ventanas, es decir, una tupla puede estar presente en más de una ventana.

Se pueden especificar otros tipos de ventanas. Por ejemplo, para obtener una ventana que contenga las 10 filas anteriores a cada fila, se puede especificar **rows** 10 **preceding**. Para obtener una ventana que contenga la fila actual, la anterior y la siguiente se puede utilizar **between rows** 1 **preceding and** 1 **following**. Para obtener las filas siguientes y la fila actual se puede decir **between rows unbounded**

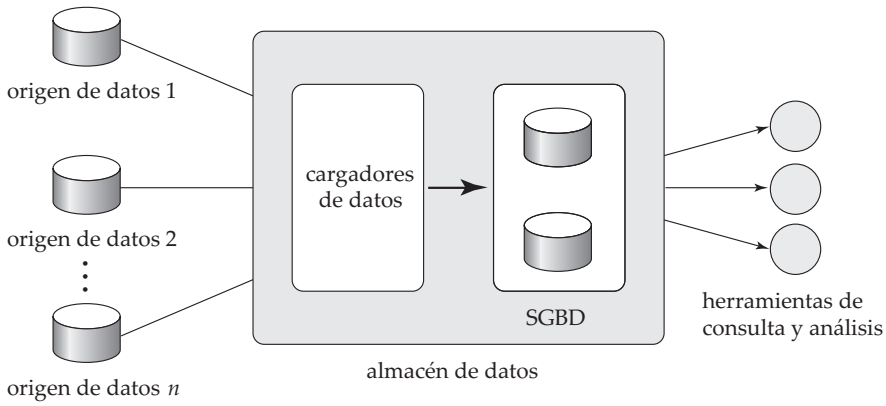


Figura 18.6 Arquitectura de los almacenes de datos.

preceding and current. Obsérvese que si la ordenación se realiza sobre un atributo que no sea clave el resultado no es determinista, ya que el orden de las tuplas no está definido completamente.

Se pueden especificar ventanas mediante rangos de valores, en lugar de hacerlo mediante número de filas. Por ejemplo, supóngase que el valor de ordenación de una tupla es v; entonces, **range between** 10 **preceding and current row** devolverá tuplas cuyo valor de ordenación se halle entre $v - 10$ y v (ambos valores inclusive). Al tratar con fechas se puede utilizar **range interval** 10 **day preceding** para obtener una ventana que contenga tuplas con los 10 días anteriores, pero sin incluir la fecha de la tupla.

Evidentemente, la funcionalidad de ventanas de SQL:1999 es mucho más rica y puede utilizarse para escribir consultas bastante complejas con poco esfuerzo.

18.3 Almacenes de datos

Las grandes empresas tienen presencia en muchos lugares, cada uno de los cuales puede generar un gran volumen de datos. Por ejemplo, las cadenas de tiendas minoristas poseen centenares o millares de tiendas, mientras que las compañías de seguros pueden tener datos de millares de oficinas locales. Además, las organizaciones grandes tienen una estructura compleja de organización interna y, por tanto, puede que los diferentes datos se hallen en ubicaciones, sistemas operativos o bajo esquemas diferentes. Por ejemplo, puede que los datos de los problemas de fabricación y los datos sobre las quejas de los clientes estén almacenados en diferentes sistemas de bases de datos. Los encargados de adoptar las decisiones empresariales necesitan tener acceso a la información de todos esos orígenes. La formulación de consultas a cada uno de los orígenes es a la vez engorrosa e ineficiente. Además, puede que los orígenes de datos sólo almacenen los datos actuales, mientras que es posible que los encargados de adoptar las decisiones empresariales necesiten tener acceso también a datos anteriores, por ejemplo, información sobre la manera en que se han modificado las pautas de compra el año pasado puede resultar de gran importancia. Los almacenes de datos proporcionan una solución a estos problemas.

Los **almacenes de datos** (data warehouses) son depósitos (o archivos) de información reunida de varios orígenes, almacenada bajo un esquema unificado en un solo sitio. Una vez reunida, los datos se almacenan mucho tiempo, lo que permite el acceso a datos históricos. Así, los almacenes de datos proporcionan a los usuarios una sola interfaz consolidada con los datos, por lo que las consultas de ayuda a la toma de decisiones resultan más fáciles de escribir. Además, al tener acceso a la información para la ayuda de la toma de decisiones desde un almacén de datos, el encargado de adoptar las decisiones se asegura de que los sistemas de procesamiento en línea de las transacciones no se vean afectados por la carga de trabajo de la ayuda de la toma de decisiones.

18.3.1 Componentes de los almacenes de datos

La Figura 18.6 muestra la arquitectura de un almacén de datos típico e ilustra la recogida de los datos, su almacenamiento y el soporte de las consultas y del análisis de datos. Entre los problemas que hay que resolver al crear un almacén de datos están los siguientes:

- **Momento y modo de la recogida de datos.** En una **arquitectura dirigida por los orígenes** para la recogida de los datos, los orígenes de los datos transmiten la información nueva, bien, de manera continua (a medida que se produce el procesamiento de las transacciones) o de manera periódica (de noche, por ejemplo). En una **arquitectura dirigida por el destino**, el almacén de datos envía de manera periódica solicitudes de datos nuevos a los orígenes de datos.

  A menos que las actualizaciones de los orígenes de datos se repliquen en el almacén de datos mediante un compromiso de dos fases, el almacén de datos nunca estará actualizado respecto a los orígenes de datos. El compromiso de dos fases suele resultar demasiado costoso para ser una opción aceptable, por lo que los almacenes de datos suelen tener datos ligeramente desactualizados. Eso, no obstante, no suele suponer un problema para los sistemas de ayuda a la toma de decisiones.

- **Selección del esquema.** Es probable que los orígenes de datos que se han creado de manera independiente tengan esquemas diferentes. De hecho, puede que utilicen diferentes modelos de datos. Parte de la labor de los almacenes de datos es llevar a cabo la integración de los esquemas y convertir los datos al esquema integrado antes de almacenarlos. En consecuencia, los datos almacenados en el almacén de datos no son una mera copia de los datos de los orígenes de datos. Por el contrario, se pueden considerar como una vista materializada de los datos de los orígenes de datos.

- **Transformación y limpieza de los datos.** La labor de corregir y realizar un procesamiento previo de los datos se denomina **limpieza de los datos**. Los orígenes de datos suelen entregar datos con numerosas inconsistencias de carácter menor, que pueden corregirse. Por ejemplo, los nombres suelen estar mal escritos y pueden que las direcciones tengan mal escritos los nombres de la calle, del distrito o de la ciudad, o puede que los códigos postales se hayan introducido de manera incorrecta. Esto puede corregirse en un grado razonable consultando una base de datos de los nombres de las calles y de los códigos postales de cada ciudad. El encaje aproximado de los datos requeridos para esta tarea se conoce como **búsqueda difusa**.

  Las listas de direcciones recogidas de varios orígenes pueden tener duplicados que haya que eliminar en una **operación de mezcla–purga** (esta operación también se conoce como **desduplicación**). Los registros de varias personas de una misma casa pueden agruparse para que sólo se realice a cada casa un envío de correo; esta operación se denomina **domiciliación**.

  Los datos se pueden **transformar** de otras formas además de la limpieza, como cambiar las unidades de medida o convertir los datos a un esquema diferente reuniendo datos de relaciones de varios orígenes. Los almacenes de los datos tienen normalmente herramientas gráficas para dar soporte a la transformación de datos. Estas herramientas permiten que la transformación se especifique con cuadros, y se puede crear arcos para indicar el flujo de datos. Los cuadros condicionales pueden encaminar datos al siguiente paso apropiado en la transformación. Véase en la Figura 29.7 un ejemplo de una transformación especificada usando la herramienta gráfica proporcionada por SQL Server de Microsoft.

- **Propagación de las actualizaciones**. Las actualizaciones de las relaciones en los orígenes de datos deben propagarse a los almacenes de datos. Si las relaciones en los almacenes de datos son exactamente las mismas que en los orígenes de datos, la propagación es directa. En caso contrario, el problema de la propagación de las actualizaciones es básicamente el problema del *mantenimiento de las vistas* que se estudió en el Apartado 14.5.

- **Resúmenes de los datos.** Los datos brutos generados por un sistema de procesamiento de transacciones pueden ser demasiado grandes para almacenarlos en línea. No obstante, se pueden responder muchas consultas manteniendo únicamente datos resumen obtenidos por agregación de las relaciones, en lugar de mantener las relaciones enteras. Por ejemplo, en lugar de almacenar los datos de cada venta de ropa, se pueden almacenar las ventas totales de ropa por nombre de artículo y por categoría.

  Supóngase que la relación r ha sido sustituida por la relación resumen s. Todavía se puede permitir a los usuarios que planteen consultas como si la relación r estuviera disponible en lí-

nea. Si la consulta sólo necesita datos resumidos, puede que sea posible transformarla en una equivalente utilizando s en lugar de r; véase el Apartado 14.5.

Los distintos pasos implicados en obtener datos a partir de un almacén de los datos se denominan **extracción**, **transformación** y **carga** o tareas **ETC**; la extracción se refiere a conseguir datos de los orígenes, mientras que la carga refiere a cargar los datos en el almacén de datos.

18.3.2 Esquemas de los almacenes de datos

Los almacenes de datos suelen tener esquemas diseñados para el análisis de los datos y emplean herramientas como las herramientas OLAP. Por tanto, los datos suelen ser datos multidimensionales, con atributos de dimensión y atributos de medida. Las tablas que contienen datos multidimensionales se denominan **tablas de hechos** y suelen ser de gran tamaño. Las tablas que registran información de ventas de una tienda minorista, con una tupla para cada artículo a la venta, son un ejemplo típico de tablas de hechos. Las dimensiones de la tabla *ventas* incluyen lo que es el artículo (generalmente un identificador del artículo como el utilizado en los códigos de barras), la fecha en que se ha vendido, la ubicación (tienda) en que se vendió, el cliente que lo ha comprado, etc. Entre los atributos de medida pueden estar el número de artículos vendidos y el precio de cada artículo.

Para minimizar los requisitos de almacenamiento los atributos de dimensiones suelen ser identificadores breves que actúan de claves externas en otras tablas denominadas **tablas de dimensiones**. Por ejemplo, la tabla de hechos *ventas* tiene los atributos *id_artículo*, *id_tienda*, *id_cliente* y *fecha* y los atributos de medida *número* y *precio*. El atributo *id_tienda* es una clave externa en la tabla de dimensiones *tienda*, que tiene otros atributos como la ubicación de la tienda (ciudad, estado, país). El atributo *id_artículo* de la tabla *ventas* es una clave externa de la tabla de dimensiones *info_artículo*, que contiene información como el nombre del artículo, la categoría a la que pertenece el artículo y otros detalles del artículo como el color y la talla. El atributo *id_cliente* es una clave externa de la tabla *cliente*, que contiene atributos como el nombre y la dirección de los clientes. También se puede ver el atributo *fecha* como clave externa de la tabla *info_fecha*, que da el mes, el trimestre y el año de cada fecha.

El esquema resultante aparece en la Figura 18.7. Un esquema así, con una tabla de hechos, varias tablas de dimensiones y claves externas procedentes de la tabla de hechos en las tablas de dimensiones, se denomina **esquema en estrella**. Los diseños complejos de almacenes de datos pueden tener varios niveles de tablas de dimensiones; por ejemplo, la tabla *info_artículo* puede tener un atributo *id_fabricante* que es clave externa en otra tabla que da detalles del fabricante. Estos esquemas se denominan **esquemas en copo de nieve**. Los diseños complejos de almacenes de datos pueden tener también más de una tabla de hechos.

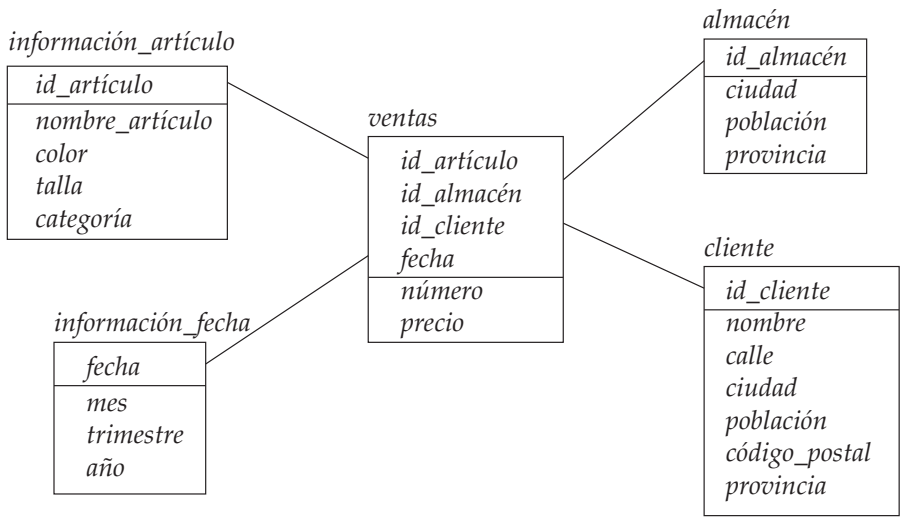


Figura 18.7 Esquema en estrella de un almacén de datos.

18.4 Minería de datos

El término **minería de datos** (data mining) hace referencia vagamente al proceso de análisis semiautomático de bases de datos de gran tamaño para hallar estructuras útiles. Al igual que la búsqueda de conocimiento en la inteligencia artificial (también denominada aprendizaje de la máquina), o el análisis estadístico, la minería de datos intenta descubrir reglas y estructuras a partir de los datos. No obstante, la minería de datos se diferencia del aprendizaje de la máquina y de la estadística en que trata con grandes volúmenes de datos, almacenados sobre todo en disco. Es decir, la minería de datos trata de la "búsqueda de conocimiento en las bases de datos".

Algunos tipos de conocimiento descubiertos a partir de una base de datos pueden representarse por un conjunto de **reglas**. A continuación se ofrece un ejemplo de regla, formulada de manera informal: "Las mujeres jóvenes con ingresos anuales superiores a 50.000 € son las personas que con mayor probabilidad compran coches deportivos de pequeño tamaño". Por supuesto, estas reglas no son verdaderas de modo universal, y tienen grados de "soporte" y de "confianza" como se estudiará más adelante. Otros tipos de conocimiento se representan por ecuaciones que relacionan entre sí diferentes variables, o mediante otros mecanismos de predicción de resultados cuando se conocen los valores de algunas variables.

Hay gran variedad de tipos posibles de estructuras que pueden resultar útiles, y se emplean diferentes técnicas para hallar tipos diferentes de estructuras. Se estudiarán unos cuantos ejemplos de estructuras y se verá el modo en que pueden obtenerse de manera automática de las bases de datos.

Suele haber una parte manual en la minería de datos, que consiste en el preprocesamiento de los datos hasta una forma aceptable para los algoritmos, y en el posprocesamiento de las estructuras descubiertas para hallar otras nuevas que puedan resultar útiles. También puede haber más de un tipo de estructura que se pueda descubrir a partir de una base de datos dada, y puede que se necesite la interacción manual para escoger los tipos de estructuras útiles. Por este motivo, la minería de datos es realmente un proceso semiautomático en la vida real. No obstante, la descripción que sigue se centrará en el aspecto automático de la minería.

18.4.1 Aplicaciones de la minería de datos

La información hallada tiene numerosas aplicaciones. Las aplicaciones más utilizadas son las que necesitan algún tipo de **predicción**. Por ejemplo, cuando una persona solicita una tarjeta de crédito, la compañía emisora quiere predecir si la persona constituye un buen riesgo de crédito. La predicción tiene que basarse en los atributos conocidos de la persona, como la edad, sus ingresos, sus deudas y su historial de pago de deudas. Las reglas para realizar la predicción se deducen de los mismos atributos de titulares de tarjetas de crédito pasados y actuales, junto con su conducta observada, como puede ser si han dejado de pagar los cargos de su tarjeta de crédito. Entre otros tipos de predicción está la de los clientes que puedan elegir a un competidor (puede que se ofrezca a esos clientes descuentos especiales para intentar que no se cambien), la predicción de la gente que pueda responder a correo publicitario ("correo basura") o la predicción de los usos de tarjetas telefónicas que puedan resultar fraudulentos.

Otra clase de aplicaciones busca **asociaciones**, por ejemplo, los libros que se suelen comprar a la vez. Si un cliente compra un libro, puede que la librería en línea le sugiera otros libros asociados. Si una persona compra una cámara, puede que el sistema sugiera accesorios que suelen comprarse junto a las cámaras. Un buen vendedor está atento a esas tendencias y las aprovecha para realizar más ventas. El desafío consiste en automatizar el proceso. Puede que otros tipos de asociación lleven al descubrimiento de relaciones causa—efecto. Por ejemplo, el descubrimiento de asociaciones inesperadas entre un medicamento recién introducido y los problemas cardiacos llevó al hallazgo de que el medicamento puede causar problemas cardiacos en algunas personas. El medicamento se retiró del mercado.

Las asociaciones son un ejemplo de **patrones descriptivos**. Las **agrupaciones** son otro ejemplo de este tipo de patrones. Por ejemplo, hace más de un siglo se descubrió una agrupación de casos de fiebre tifoidea alrededor de un pozo, lo que llevó al descubrimiento de que el agua del pozo estaba contaminada y estaba difundiendo la fiebre tifoidea. La detección de agrupaciones de enfermedades sigue siendo importante hoy en día.

18.4.2 Clasificación

Como ya se mencionó en el Apartado 18.4.1, la predicción es uno de los tipos más importantes de minería de datos. Se describirá lo que es la clasificación, se estudiarán técnicas para la creación de un tipo de clasificadores, denominados clasificadores de árboles de decisión, y se estudiarán otras técnicas de predicción.

De manera abstracta, el problema de la **clasificación** es el siguiente: dado que los elementos pertenecen a una de las clases, y dados los casos pasados (denominados **ejemplos de formación**) de los elementos junto con las clases a las que pertenecen, el problema es predecir la clase a la que pertenece un elemento nuevo. La clase del caso nuevo no se conoce, por lo que hay que utilizar los demás atributos del caso para predecir la clase.

La clasificación se puede llevar a cabo hallando reglas que dividan los datos dados en grupos disjuntos. Por ejemplo, supóngase que una compañía de tarjetas de crédito quiera decidir si debe conceder una tarjeta a un solicitante. La compañía tiene amplia información sobre esa persona, como puede ser su edad, su nivel educativo, sus ingresos anuales y sus deudas actuales, la cual puede utilizar para adoptar una decisión.

Parte de esta información puede ser importante para el valor de crédito del solicitante, mientras que puede que otra parte no lo sea. Para adoptar la decisión la compañía asigna un nivel de valor de crédito de excelente, bueno, mediano o malo a cada integrante de un conjunto de muestra de clientes *actuales* según su historial de pagos. Luego, la compañía intenta hallar reglas que clasifiquen a sus clientes actuales como excelentes, buenos, medianos o malos con base en la información sobre esas personas diferente de su historial de pagos actual (que no está disponible para los clientes nuevos). Considérense sólo dos atributos: el nivel educativo (la titulación más alta conseguida) y los ingresos. Las reglas pueden ser de la forma siguiente:

$$\forall persona P, P.titulación = máster \textbf{ and } P.ingresos > 75.000 \Rightarrow P.crédito = excelente$$

$$\forall persona P, P.titulación = bachiller \textbf{ or } (P.ingresos \geq 25.000 \textbf{ and } P.ingresos \leq 75.000) \Rightarrow P.crédito = bueno$$

También aparecen reglas parecidas para los demás niveles de valor de crédito (mediano y malo).

El proceso de creación de clasificadores comienza con una muestra de los datos, denominada **conjunto de formación**. Para cada tupla del conjunto de formación ya se conoce la clase a la que pertenece. Por ejemplo, el conjunto de formación de las solicitudes de tarjetas de crédito pueden ser los clientes ya existentes, con su riesgo crediticio determinado a partir de su historial de pagos. Los datos actuales, o población, puede consistir en toda la gente, incluida la que no es todavía cliente. Hay varias maneras de crear clasificadores, como se verá.

18.4.2.1 Clasificadores de árboles de decisión

Los clasificadores de árboles de decisión son una técnica muy utilizada para la clasificación. Como sugiere el nombre, los **clasificadores de árboles de decisión** utilizan un árbol; cada nodo hoja tiene una clase asociada, y cada nodo interno tiene un predicado (o, de manera más general, una función) asociado. La Figura 18.8 muestra un ejemplo de árbol de decisión.

Para clasificar un nuevo caso se empieza por la raíz y se recorre el árbol hasta alcanzar una hoja; en los nodos internos se evalúa el predicado (o la función) para el ejemplo de datos, para hallar a qué nodo hijo hay que ir. El proceso continúa hasta que se llega a un nodo hoja. Por ejemplo, si el nivel académico de la persona es de máster y sus ingresos son de 40K, partiendo de la raíz se sigue el arco etiquetado "máster", y desde allí el arco etiquetado "25K a 75K", hasta alcanzar una hoja. La clase de la hoja es "bueno", por lo que se puede predecir que la solvencia de esa persona es buena.

Creación de clasificadores de árboles de decisión

La pregunta que se plantea es el modo de crear un clasificador de árboles de decisión, dado un conjunto de casos de formación. La manera más frecuente de hacerlo es utilizar un algoritmo **impaciente**, que trabaja de manera recursiva, comenzando por la raíz y construyendo el árbol hacia abajo. Inicialmente sólo hay un nodo, la raíz, y todos los casos de formación están asociados con ese nodo.

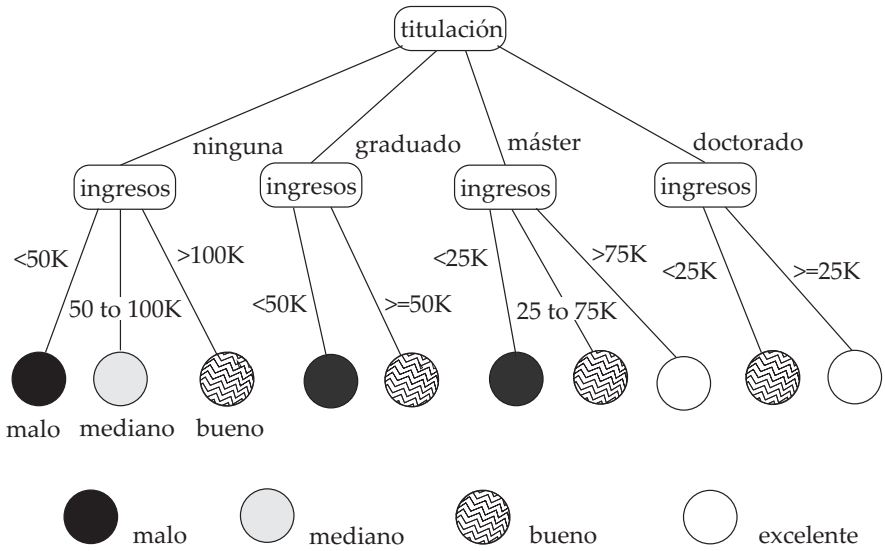


Figura 18.8 Árbol de clasificación.

En cada nodo, si todos o "casi todos" los ejemplos de formación asociados con el nodo pertenecen a la misma clase, el nodo se convierte en un nodo hoja asociado con esa clase. En caso contrario, hay que seleccionar un **atributo de partición** y **condiciones de partición** para crear nodos hijo. Los datos asociados con cada nodo hijo son el conjunto de ejemplos de formación que satisfacen la condición de partición de ese nodo hijo. En el ejemplo elegido, se escoge el atributo *titulación* y se crean cuatro hijos, uno por cada valor de la titulación. Las condiciones para los cuatro nodos hijo son *titulación* = ninguna, *titulación* = bachiller, *titulación* = máster y *titulación* = doctorado, respectivamente. Los datos asociados con cada hijo son los ejemplos de formación asociados con ese hijo. En el nodo correspondiente a máster se escoge el atributo *ingresos* con el rango de valores dividido en los intervalos 0 a 25K, 25K a 50K, 50K a 75K y más de 75K. Los datos asociados con cada nodo son los ejemplos de formación con atributo *titulación* igual a máster y el atributo *ingresos* en cada uno de los rangos, respectivamente. Como optimización, ya que la clase para el rango de 25K a 50K y el rango de 50K a 75K es el mismo bajo el nodo *titulación* = máster, se han unido los dos rangos en uno solo que va de 25K a 75K.

Las mejores particiones

De manera intuitiva, al escoger una secuencia de atributos de partición, se comienza con el conjunto de todos los ejemplos de formación, que es "impuro" en el sentido de que contiene ejemplos de muchas clases, y se acaba con las hojas, que son "puras" en el sentido de que en cada hoja todos los ejemplos de formación pertenecen a una única clase. Se verá brevemente el modo de medir cuantitativamente la pureza. Para evaluar la ventaja de escoger un atributo concreto y la condición para la partición de los datos en un nodo se mide la pureza de los datos en los hijos resultantes de la partición según ese atributo. Se escogen el atributo y la condición que producen la pureza máxima.

La pureza de un conjunto S de ejemplos de formación puede medirse cuantitativamente de varias maneras. Supóngase que hay k clases y que de los ejemplos en S la fracción de ejemplos de la clase i es p_i. Una medida de pureza, la **medida de Gini**, se define como

$$\text{Gini}(S) = 1 - \sum_{i-1}^{k} p_i^2$$

Cuando todos los ejemplos están en una sola clase, el valor de Gini es 0, mientras que alcanza su máximo (de $1 - 1/k$) si cada clase tiene el mismo número de ejemplos. Otra medida de la pureza es la **medida de la entropía**, que se define como

$$\text{Entropía}(S) = -\sum_{i-1}^{k} p_i \log_2 p_i$$

la entropía es 0 si todos los ejemplos están en una sola clase y alcanza su máximo cuando cada clase tiene el mismo número de ejemplos. La medida de la entropía proviene de la teoría de la información.

Cuando un conjunto S se divide en varios conjuntos $S_i, i = 1, 2, \ldots, r$, se puede medir la pureza del conjunto de conjuntos resultante como:

$$\text{Pureza}(S_1, S_2, \ldots, S_r) = \sum_{i=1}^{r} \frac{|S_i|}{|S|} \text{pureza}(S_i)$$

Es decir, la pureza es la media ponderada de la pureza de los conjuntos S_i. La fórmula anterior puede utilizarse tanto con la medida de la pureza de Gini como con la medida de la pureza de la entropía.

La **ganancia de información** debida a una partición concreta de S en $S_i, i = 1, 2, \ldots, r$ es, entonces,

$$\text{Ganancia_información}(S, \{S_1, S_2, \ldots, S_r\}) = \text{pureza}(S) - \text{pureza}(S_1, S_2, \ldots, S_r)$$

Las particiones en menor número de conjuntos son preferibles a las particiones en muchos conjuntos, ya que llevan a árboles de decisión más sencillos y significativos. El número de elementos en cada uno de los conjuntos S_i también puede tenerse en cuenta; en caso contrario, que un conjunto S_i tenga uno o ningún elemento supondría una gran diferencia en el número de conjuntos, aunque la partición fuera la misma para casi todos los elementos. El **contenido de información** de una partición concreta puede expresarse en términos de entropía como

$$\text{Contenido_información}(S, \{S_1, S_2, \ldots, S_r\}) = -\sum_{i-1}^{r} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

Todo esto lleva a una definición: la **mejor partición** para un atributo es la que da el máximo **índice de ganancia de información**, definido como

$$\frac{\text{Ganancia_información}(S, \{S_1, S_2, \ldots, S_r\})}{\text{Contenido_información}(S, \{S_1, S_2, \ldots, S_r\})}$$

Búsqueda de las mejores particiones

Hay que averiguar el modo de hallar la mejor partición para un atributo. El modo de dividir un atributo depende del tipo de atributo. Los atributos pueden tener **valores continuos**, es decir, los valores se pueden ordenar de manera significativa para la clasificación, como la edad o los ingresos, o pueden ser **determinantes**; es decir, no tener ningún orden significativo, como los nombres de los departamentos o los de los países. No se espera que el orden de los nombres de los departamentos o el de los países tenga ningún significado para la clasificación.

Generalmente, los atributos que son números (enteros o reales) se tratan como valores continuos y los atributos de cadenas de caracteres se tratan como categóricos, pero esto puede controlarlo el usuario del sistema. En el ejemplo escogido se ha tratado el atributo *titulación* como categórico y el atributo *ingresos* como valor continuo.

En primer lugar se considera el modo de hallar las mejores particiones para los atributos con valores continuos. Por simplificar sólo se considerarán **particiones binarias** de los atributos con valores continuos, es decir, particiones que den lugar a dos hijos. El caso de las **particiones múltiples** es más complicado; véanse las notas bibliográficas para hallar referencias sobre este asunto.

Para hallar la mejor partición binaria de un atributo con valores continuos, en primer lugar, se ordenan los valores del atributo en los ejemplos de formación. Luego se calcula la ganancia de información obtenida por la división en cada valor. Por ejemplo, si los ejemplos de formación tienen los valores $1, 10, 15$ y 25 para un atributo, los puntos de partición considerados son 1, 10 y 15; en cada caso, los valores menores o iguales que el punto de partición forman una partición y el resto de los valores forman la otra. La mejor partición binaria para el atributo es la partición que da la ganancia de información máxima.

Para los atributos categóricos se pueden tener particiones múltiples, con un hijo para cada valor del atributo. Esto funciona muy bien para los atributos categóricos con pocos valores diferentes, como la titulación o el sexo. No obstante, si el atributo tiene muchos valores diferentes, como los nombres de los departamentos en compañías grandes, la creación de un hijo para cada valor no es una buena idea. En

```
procedure CultivarÁrbol(S)
    Partición(S);

procedure Partición (S)
    if (pureza(S) > δ_p or |S| < δ_s ) then
        return;
    for each atributo A
        evaluar las particiones según el atributo A;
    Usar la mejor partición hallada (para todos los atributos) para dividir
            S into S_1, S_2, ..., S_r;
    for i = 1, 2, ..., r
        Partición(S_i);
```

Figura 18.9 Construcción recursiva de un árbol de decisión.

esos casos se procura combinar varios valores en cada hijo para crear un número menor de hijos. Véanse las notas bibliográficas para hallar referencias al modo de hacerlo.

Algoritmo de construcción del árbol de decisión

La idea principal de la construcción de árboles de decisión es la evaluación de los diferentes atributos y de las distintas condiciones de partición y la selección del atributo y de la condición de partición que generen el índice máximo de ganancia de información. El mismo procedimiento funciona de manera recursiva en cada uno de los conjuntos resultantes de la partición, lo que hace que se construya de manera recursiva el árbol de decisión. Si los datos pueden clasificarse de manera perfecta, la recursión se detiene cuando la pureza de un conjunto sea 0. No obstante, los datos suelen tener ruido, o puede que un conjunto sea tan pequeño que no se justifique estadísticamente su partición. En ese caso, la partición se detiene cuando la pureza del conjunto es "bastante alta" y la clase de la hoja resultante se define como la clase de la mayoría de los elementos del conjunto. En general, las diferentes ramas del árbol pueden crecer hasta niveles diferentes.

La Figura 18.9 muestra pseudocódigo para un procedimiento recursivo de construcción de un árbol, que toma al conjunto de ejemplos de formación S como parámetro. La recursión se detiene cuando el conjunto es lo bastante puro o el conjunto S es demasiado pequeño para que más particiones resulten estadísticamente significativas. Los parámetros δ_p y δ_s definen los valores de corte para la pureza y el tamaño; puede que el sistema les dé valores predeterminados, que los usuarios pueden cancelar.

Hay gran variedad de algoritmos de construcción de árboles de decisión y se esbozarán las características distintivas de unos cuantos. Véanse las notas bibliográficas para hallar más detalles. Con conjuntos de datos de tamaño muy grande la realización de particiones puede resultar muy costosa, ya que implica la realización repetida de copias. Por tanto, se han desarrollado varios algoritmos para minimizar el coste de E/S y el coste de computación cuando los datos de formación son mayores que la memoria disponible.

Varios de los algoritmos también podan los subárboles del árbol de decisión generado para reducir el **exceso de ajuste**: un subárbol tiene exceso de ajuste si se ha ajustado tanto a los detalles de los datos de formación que comete muchos errores de clasificación con otros datos. Se poda un subárbol sustituyéndolo por un nodo hoja. Hay varias heurísticas de poda; una utiliza parte de los datos de formación para construir el árbol y otra parte para comprobarlo. La heurística poda el subárbol si descubre que los errores de clasificación de los casos de prueba se reducirían si se sustituyera por un nodo hoja.

Se pueden generar reglas de clasificación a partir de los árboles de decisión, si se desea. Para cada hoja se genera una regla de la manera siguiente: la parte izquierda es la conjunción de todas las condiciones de partición del camino hasta la hoja y la clase es la clase de la mayoría de los ejemplos de formación de la hoja. Un ejemplo de estas reglas de clasificación es

$$titulación = máster \textbf{ and } ingresos > 75.000 \Rightarrow excelente$$

18.4.2.2 Otros tipos de clasificadores

Hay varios tipos de clasificadores aparte de los clasificadores de árbol. Dos tipos que han resultado bastante útiles son los *clasificadores de redes neuronales* y los *clasificadores bayesianos*. Los clasificadores de redes neuronales utilizan los datos de formación para adiestrar redes neuronales artificiales. Hay gran cantidad de literatura sobre las redes neuronales y aquí no se hablará más de ellas.

Los **clasificadores bayesianos** hallan la distribución de los valores de los atributos para cada clase de los datos de formación; cuando se da un nuevo caso, d, utilizan la información de la distribución para estimar, para cada clase c_j, la probabilidad de que el caso d pertenezca a la clase c_j, denotada por $p(c_j|d)$, de la manera que aquí se describe. La clase con la probabilidad máxima se transforma en la clase predicha para el caso d.

Para hallar la probabilidad $p(c_j|d)$ de que el caso d esté en la clase c_j los clasificadores bayesianos utilizan el **teorema de Bayes**, que establece

$$p(c_j|d) = \frac{p(d|c_j)p(c_j)}{p(d)}$$

donde $p(d|c_j)$ es la probabilidad de que se genere el caso d dada la clase c_j, $p(c_j)$ es la probabilidad de ocurrencia de la clase c_j, y $p(d)$ es la probabilidad de que ocurra el caso d. De éstas, p (d) puede ignorarse, ya que es igual para todas las clases. $p(d)$ no es más que la fracción de los casos de formación que pertenecen a la clase c_j.

Hallar exactamente $p(d|c_j)$ resulta difícil, ya que exige una distribución completa de los casos de c_j. Para simplificar la tarea los **clasificadores bayesianos ingenuos** dan por hecho que los atributos tienen distribuciones independientes y, por tanto, estiman

$$p(d|c_j) = p(d_1|c_j) * p(d_2|c_j) * \ldots * p(d_n|c_j)$$

Es decir, la probabilidad de que ocurra el caso d es el producto de la probabilidad de ocurrencia de cada uno de los valores d_i del atributo d, dado que la clase es c_j.

Las probabilidadess $p(d_i|c_j)$ proceden de la distribución de los valores de cada atributo i, para cada clase c_j. Esta distribución se calcula a partir de los ejemplos de formación que pertenecen a cada clase c_j; la distribución suele aproximarse mediante un histograma. Por ejemplo, se puede dividir el rango de valores del atributo i en intervalos iguales y almacenar la fracción de casos de la clase c_j que caen en cada intervalo. Dado un valor d_i para el atributo i, el valor de $p(d_i|c_j)$ es simplemente la fracción de casos que pertenecen a la clase c_j que caen en el intervalo al que pertenece d_i.

Una ventaja significativa de los clasificadores bayesianos es que pueden clasificar los casos con valores de los atributos desconocidos y nulos—los atributos desconocidos o nulos simplemente se omiten del cálculo de probabilidades. Por el contrario, los clasificadores de árboles de decisión no pueden tratar de manera significativa las situaciones en que el caso que hay que clasificar tiene un valor nulo para el atributo de partición utilizado para avanzar por el árbol de decisión.

18.4.2.3 Regresión

La **regresión** trata de la predicción de valores, no de clases. Dados los valores de un conjunto de variables, $X_1, X_2, \ldots, X_n$, se desea predecir el valor de una variable Y. Por ejemplo, se puede tratar el nivel educativo como un número y los ingresos como otro número y , con base en estas dos variables, querer predecir la posibilidad de impago, que podría ser un porcentaje de probabilidad de impago o el importe impagado.

Una manera de inferir los coeficientes $a_0, a_1, a_2, \ldots, a_n$ tales que

$$Y = a_0 + a_1 * X_1 + a_2 * X_2 + \cdots + a_n * X_n$$

La búsqueda de ese polinomio lineal se denomina **regresión lineal**. En general, se quiere hallar una curva (definida por un polinomio o por otra fórmula) que se ajuste a los datos; el proceso también se denomina **ajuste de la curva**.

El ajuste sólo puede ser aproximado, debido al ruido de los datos o a que la relación no sea exactamente un polinomio, por lo que la regresión pretende hallar coeficientes que den el mejor ajuste posible.

Hay técnicas estándar en estadística para hallar los coeficientes de regresión. Aquí no se estudiarán esas técnicas, pero las notas bibliográficas ofrecen referencias.

18.4.3 Reglas de asociación

Los comercios minoristas suelen estar interesados en las **asociaciones** entre los diferentes artículos que compra la gente. Ejemplos de esas asociaciones son:

- Alguien que compra pan es bastante probable que compre también leche.
- Una persona que compró el libro *Fundamentos de bases de datos* es bastante probable que también compre el libro *Fundamentos de sistemas operativos*.

La información de asociación puede utilizarse de varias maneras. Cuando un cliente compra un libro determinado puede que la librería en línea le sugiera los libros asociados. Puede que la tienda de alimentación decida colocar el pan cerca de la leche, ya que suelen comprarse juntos, para ayudar a los clientes a hacer la compra más rápidamente. O puede que la tienda los coloque en extremos opuestos del mostrador y coloque otros artículos asociados entre medias para inducir a la gente a comprar también esos artículos, mientras los clientes van de un extremo a otro del mostrador. Puede que una tienda que ofrece descuento en un artículo asociado no lo ofrezca en el otro, ya que, de todos modos, el cliente comprará el segundo artículo.
Un ejemplo de regla de asociación es

$$pan \Rightarrow leche$$

En el contexto de las compras de alimentación, la regla dice que los clientes que compran pan también tienden a comprar leche con una probabilidad elevada. Una regla de asociación debe tener una **población** asociada: la población consiste en un conjunto de **casos**. En el ejemplo de la tienda de alimentación, la población puede consistir en todas las compras en la tienda de alimentación; cada compra es un caso. En el caso de una librería, la población puede consistir en toda la gente que realiza compras, independientemente del momento en que las hayan realizado. Cada consumidor es un caso. Aquí, el analista ha decidido que el momento de realización de la compra no es significativo, mientras que, para el ejemplo de la tienda de alimentación, puede que el analista haya decidido concentrarse en cada compra, ignorando las diferentes visitas de un mismo cliente.

Las reglas tienen un *soporte*, así como una *confianza* asociados. Los dos se definen en el contexto de la población:

- El **soporte** es una medida de la fracción de la población que satisface tanto el antecedente como el consecuente de la regla.

  Por ejemplo, supóngase que sólo el 0.001 por ciento de todas las compras incluyen leche y destornilladores. El soporte de la regla

  $$leche \Rightarrow destornilladores$$

  es bajo. Puede que la regla ni siquiera sea estadísticamente significativa—quizás solo hubiera una única compra que incluyera leche y destornilladores. Las empresas no suelen estar interesadas en las reglas que tienen un soporte bajo, ya que afectan a pocos clientes y no merece la pena prestarles atención.

  Por otro lado, si el 50 por ciento de las compras implica leche y pan, el soporte de las reglas que afecten al pan y a la leche (y a ningún otro artículo) es relativamente elevado, y puede que merezca la pena prestarles atención. El grado mínimo de soporte que se considera deseable exactamente depende de la aplicación.

- La **confianza** es una medida de la frecuencia con que el consecuente es cierto cuando lo es el antecedente. Por ejemplo, la regla

  $$pan \Rightarrow leche$$

  tiene una confianza del 80 por ciento si el 80 por ciento de las compras que incluyen pan incluyen también leche. Las reglas con una confianza baja no son significativas. En las aplicaciones comer-

ciales las reglas suelen tener confianzas significativamente menores del 100 por ciento, mientras que en otros campos, como la física, las reglas pueden tener confianzas elevadas.

Hay que tener en cuenta que la confianza de $pan \Rightarrow leche$ puede ser muy diferente de la confianza de $leche \Rightarrow pan$, aunque las dos tienen el mismo soporte.

Para descubrir reglas de asociación de la forma

$$i_1, i_2, \ldots, i_n \Rightarrow i_0$$

en primer lugar hay que determinar los conjuntos de elementos con soporte suficiente, denominados **conjuntos grandes de elementos**. En el ejemplo que se trata se hallan conjuntos de elementos que están incluidos en un número de casos lo bastante grande. En breve se verá el modo de calcular conjuntos grandes de elementos.

Para cada conjunto grande de elementos se obtienen todas las reglas con confianza suficiente que afecten a todos los elementos del conjunto y sólo a ellos. Para cada conjunto grande de elementos S se obtiene una regla $S - s \Rightarrow s$ para cada subconjunto $s \subset S$, siempre que $S - s \Rightarrow s$ tenga confianza suficiente; la confianza de la regla la da el soporte de s dividido por el soporte de S.

Ahora se considerará el modo de generar todos los conjuntos grandes de elementos. Si el número de conjuntos de elementos posibles es pequeño, basta con un solo paso por los datos para detectar el nivel de soporte de todos los conjuntos. Se lleva una cuenta, con valor inicial 0, para cada conjunto de elementos. Cuando se captura el registro de una compra, la cuenta se incrementa para cada conjunto de elementos tal que todos los elementos del conjunto estén contenidos en la compra. Por ejemplo, si una compra incluye los elementos a, b y c, se incrementará el contador para $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b\}$, $\{b, c\}$, $\{a, c\}$ y $\{a, b, c\}$. Los conjuntos con un contador lo bastante elevado al final del pase se corresponden con los elementos que tienen un grado de asociación elevado.

El número de conjuntos crece de manera exponencial, lo que hace inviable el proceso que se acaba de describir si el número de elementos es elevado. Afortunadamente, casi todos los conjuntos tienen normalmente un soporte muy bajo; se han desarrollado optimizaciones para no considerar la mayor parte de esos conjuntos. Estas técnicas utilizan varios pases por la base de datos y sólo consideran algunos conjuntos en cada pase.

En la técnica **a priori** para la generación de conjuntos de artículos grandes sólo se consideran en el primer pase los conjuntos con un solo elemento. En el segundo pase se consideran los conjuntos con dos artículos, etc.

Al final de cada pase todos los conjuntos con soporte suficiente se consideran conjuntos grandes de elementos. Los conjuntos que se ha hallado que tienen demasiado poco soporte al final de cada pase se eliminan. Una vez eliminado un conjunto no hace falta considerar ninguno de sus superconjuntos. En otros términos, en el pase i sólo hay que contar el soporte de los conjuntos de tamaño i tales que se haya hallado que todos sus subconjuntos tienen un soporte lo bastante elevado; basta con probar todos los subconjuntos de tamaño $i - 1$ para asegurase de que se cumple esta propiedad. Al final del pase i se halla que ningún conjunto de tamaño i tiene el soporte suficiente, por lo que no hace falta considerar ningún conjunto de tamaño $i + 1$. Entonces, el cálculo se termina.

18.4.4 Otros tipos de asociación

El uso de meras reglas de asociación tiene varios inconvenientes. Uno de los principales es que muchas asociaciones no son muy interesantes, ya que pueden predecirse. Por ejemplo, si mucha gente compra cereales y mucha gente compra pan, se puede predecir que un número bastante grande de personas comprará las dos cosas, aunque no haya ninguna relación entre las dos compras. De hecho, incluso si la compra de cereal tiene una influencia negativa suave en la compra de pan (es decir, los clientes que compran cereal tienden a comprar pan menos a menudo que el cliente medio), la asociación entre el cereal y el pan puede tener un soporte alto.

Lo que resultaría interesante es una **desviación** de la ocurrencia conjunta de las dos compras. En términos estadísticos, se buscan **correlaciones** entre los artículos; las correlaciones pueden ser positivas, en las que la ocurrencia conjunta es superior a lo esperado, o negativa, en la que los elementos ocurren conjuntamente menos frecuentemente de lo predicho. Así, si la compra de pan no se correlaciona con el cereal, no se informa, incluso si hubiese una asociación fuerte entre los dos. Hay medidas estándares de

correlación usadas ampliamente en el área de la estadística. Se puede consultar cualquier libro de texto estándar de estadística para hallar más información sobre las correlaciones.

Otra clase importante de aplicaciones de minería de datos son las asociaciones de secuencias (o correlaciones de secuencias). Las series de datos temporales, como las cotizaciones bursátiles en una serie de días, constituyen un ejemplo de datos de secuencias. Los analistas bursátiles desean hallar asociaciones entre las secuencias de cotizaciones. Un ejemplo de asociación de este tipo es la regla siguiente: "Siempre que las tasas de interés de los bonos suben, las cotizaciones bursátiles bajan en un plazo de dos días". El descubrimiento de esta asociación entre secuencias puede ayudar a adoptar decisiones de inversión inteligentes. Véanse las notas bibliográficas para hallar referencias a la investigación en este campo.

Las desviaciones de las estructuras temporales suelen resultar interesantes. Por ejemplo, si una empresa ha estado creciendo a una tasa constante cada año, una desviación de la tasa de crecimiento habitual resulta sorprendente. Si las ventas de ropa de invierno bajan en verano, ya que puede predecirse con base en los años anteriores, una desviación que no se pudiera predecir a partir de la experiencia pasada se consideraría interesante. Las técnicas de minería pueden hallar desviaciones de lo esperado con base en las estructuras temporales o secuenciales pasadas. Véanse las notas bibliográficas para hallar referencias a la investigación en este campo.

18.4.5 Agrupamiento

De manera intuitiva, el agrupamiento hace referencia al problema de hallar agrupaciones de puntos en los datos dados. El problema del **agrupamiento** puede formalizarse de varias maneras a partir de las métricas de distancias. Una manera es formularlo como el problema de agrupar los puntos en k conjuntos (para un k dado) de modo que la distancia media de los puntos al *centroide* de su agrupación asignada sea mínima[5]. Otra manera es agrupar los puntos de modo que la distancia media entre cada par de puntos de cada agrupación sea mínima. Hay otras definiciones; véanse las notas bibliográficas para hallar más detalles. Pero la intuición subyacente a todas estas definiciones es agrupar los puntos parecidos en un único conjunto.

Otro tipo de agrupamiento aparece en los sistemas de clasificaciones de la biología (estos sistemas de clasificación no intentan *predecir* las clases, sino agrupar los elementos relacionados). Por ejemplo, los leopardos y los seres humanos se agrupan bajo la clase mamíferos, mientras que los cocodrilos y las serpientes se agrupan bajo los reptiles. Tanto los mamíferos como los reptiles están bajo la clase común de los cordados. La agrupación de los mamíferos tiene subagrupaciones, como los carnívoros y los primates. Por tanto, se tiene un **agrupamiento jerárquico**. Dadas las características de las diferentes especies, los biólogos han creado un esquema complejo de agrupamiento jerárquico que agrupa las especies relacionadas en diferentes niveles de la jerarquía.

El agrupamiento jerárquico también resulta útil en otros dominios—para agrupar documentos, por ejemplo. Los sistemas de directorio de Internet (como el de Yahoo) agrupan los documentos relacionados de manera jerárquica (véase el Apartado 19.9). Los algoritmos de agrupamiento jerárquico pueden clasificarse como algoritmos de **agrupamiento aglomerativo**, que comienzan creando agrupaciones pequeñas y luego crean los niveles superiores, o como algoritmos de **agrupamiento divisivo**, que primero crean los niveles superiores del agrupamiento jerárquico y luego refinan cada agrupación resultante en agrupaciones de niveles inferiores.

La comunidad estadística ha estudiado extensamente los agrupamientos. La investigación en bases de datos ha proporcionado algoritmos escalables de agrupamiento que pueden agrupar conjuntos de datos de tamaño muy grande (que puede que no quepan en la memoria). El algoritmo de agrupamiento Birch es un algoritmo de este tipo. De manera intuitiva, los puntos de datos se insertan en una estructura arbórea multidimensional (basada en los árboles R descritos en el Apartado 24.3.5.3) y son llevados a los nodos hoja correspondientes de acuerdo con su cercanía a los puntos representativos de los nodos internos del árbol. Los puntos próximos, por tanto, se agrupan en los nodos hoja, y se resumen si hay más puntos de los que caben en la memoria. El resultado de esta primera fase de agrupamiento es crear un conjunto de datos agrupado parcialmente que quepa en memoria. Las técnicas estándares de

5. El centroide de un conjunto de puntos se define como un punto cuyas coordenadas en cada dimensión son el promedio de las coordenadas de todos los puntos de ese conjunto en esa dimensión. Por ejemplo, en dos dimensiones, el centroide de un conjunto de puntos $\{ (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \}$ viene dado por $(\frac{\sum_{i=1}^{n} x_i}{n}, \frac{\sum_{i=1}^{n} y_i}{n})$.

agrupamiento se pueden ejecutar con datos en memoria para conseguir el agrupamiento final. Véanse las notas bibliográficas para hallar las referencias al algoritmo Birch y a otras técnicas de agrupamiento, incluidos los algoritmos para el agrupamiento jerárquico.

Una aplicación interesante del agrupamiento es la predicción de las películas nuevas (o de los libros nuevos, o de la música nueva) que es probable que interesen a una persona, con base en:

1. Las preferencias cinematográficas pasadas de esa persona.
2. Otras personas con preferencias pasadas parecidas.
3. Las preferencias de esa gente entre las películas nuevas.

Un enfoque de este problema es el siguiente. Para hallar gente con preferencias anteriores parecidas se crean agrupaciones de personas de acuerdo con sus preferencias cinematográficas. La exactitud del agrupamiento puede mejorarse agrupando previamente las películas por su parecido, de modo que, aunque la gente no haya visto las mismas películas, se agruparán si han visto películas parecidas. Se puede repetir el agrupamiento, agrupando alternativamente gente y películas hasta que se alcance un equilibrio. Dado un nuevo usuario, se halla una agrupación de usuarios lo más parecidos posibles a él, con base en las preferencias del usuario por las películas que ya ha visto. Luego se predice que las películas de las agrupaciones de películas que son populares en la agrupación de ese usuario es probable que resulten interesantes para el nuevo usuario. De hecho, este problema es un caso de *filtrado colaborativo*, en el que los usuarios colaboran en la tarea de filtrado de la información para hallar información de interés.

18.4.6 Otros tipos de minería

La **minería de texto** aplica las técnicas de minería de datos en documentos de texto. Por ejemplo, hay herramientas que forman agrupaciones de las páginas que ha visitado un usuario; esto ayuda a los usuarios cuando examinan su historial de exploración para hallar las páginas que han visitado anteriormente. La distancia entre las páginas puede basarse, por ejemplo, en las palabras frecuentes en esas páginas (véase el Apartado 19.2.2). Otra aplicación es la clasificación automática de las páginas en directorios Web, de acuerdo con su parecido con otras páginas (véase el Apartado 19.9).

Los sistemas de **visualización de datos** ayudan a los usuarios a examinar grandes volúmenes de datos y a detectar visualmente las estructuras. Las visualizaciones de datos—como los mapas, los gráficos y otras representaciones gráficas—permiten que los datos se presenten a los usuarios de manera compacta. Una sola pantalla gráfica puede codificar tanta información como un número mucho mayor de pantallas de texto. Por ejemplo, si el usuario desea averiguar si los problemas de producción en las factorías están correlacionadas con su ubicación se pueden codificar las ubicaciones problemáticas en un color especial—por ejemplo, rojo—en un mapa. El usuario puede descubrir rápidamente las ubicaciones en las que se dan los problemas. El usuario puede así formular hipótesis sobre el motivo de que los problemas se produzcan en esas ubicaciones y contrastarlas cuantitativamente con la base de datos.

Otro ejemplo más: la información sobre los valores puede codificarse como colores y mostrarse con sólo un píxel de área de pantalla. Para detectar las asociaciones entre pares de elementos se puede utilizar una matriz bidimensional de píxeles en la que cada fila y cada columna representen un elemento. El porcentaje de transacciones que compran los dos elementos puede codificarse por la intensidad del color de los píxeles. Los elementos con asociaciones elevadas aparecerán en la pantalla como píxeles brillantes—fáciles de detectar contra el fondo más oscuro.

Los sistemas de visualización de datos no detectan de manera automática las estructuras, sino que proporcionan soporte del sistema para que los usuarios las detecten. Dado que los seres humanos son muy buenos en la detección de estructuras visuales, la visualización de los datos es un componente importante de la minería de datos.

18.5 Resumen

- Los sistemas de ayuda a la toma de decisiones analizan en línea los datos recogidos por los sistemas de procesamiento de transacciones para ayudar a los usuarios a adoptar decisiones de negocios. Dado que hoy en día la mayor parte de las organizaciones están intensamente informatizadas, se dispone de una enorme cantidad de información para la ayuda a la toma de decisiones. Los sistemas de ayuda a la toma de decisiones se presentan en varios formatos, incluidos los sistemas OLAP y los sistemas de minería de datos.

- Las herramientas de procesamiento analítico en línea (online analytical processing, OLAP) ayudan a los analistas a ver los datos resumidos de diferentes maneras, de forma que puedan obtener una perspectiva del funcionamiento de la organización.

  - Las herramientas OLAP trabajan con datos multidimensionales, caracterizados por los atributos de dimensiones y por los atributos de medida.
  - Los cubos de datos consisten en datos multidimensionales resumidos de diferentes maneras. El cálculo previo de los cubos ayuda a acelerar las consultas de resúmenes de datos.
  - El formato de las tabulaciones cruzadas permite que los usuarios vean simultáneamente dos dimensiones de los datos multidimensionales, junto con resúmenes de los datos.
  - La concreción, la abstracción y los cortes de los cubos de datos son algunas de las operaciones que los usuarios llevan a cabo con las herramientas OLAP.

- El componente OLAP de la norma SQL:1999 proporciona gran variedad de funcionalidades nuevas para el análisis de los datos, incluidas nuevas funciones de agregación; operaciones con cubos y de abstracción, funciones de clasificación; funciones para la creación de ventanas, que soportan la elaboración de resúmenes en ventanas móviles; y la realización de particiones, con la creación de ventanas y la clasificación aplicadas dentro de cada partición.

- Los almacenes de los datos ayudan a obtener y archivar datos operacionales importantes. Los almacenes se usan para la ayuda a la toma de decisiones y para el análisis de datos históricos, por ejemplo, para predecir tendencias. La limpieza de datos de orígenes de datos de entrada es a menudo una tarea importante en los almacenes de los datos. Los esquemas de los almacenes de datos tienden a ser multidimensionales, implicando una o varias tablas muy grandes de hechos y varias tablas de dimensiones mucho más pequeñas.

- La minería de datos es el proceso de análisis semiautomático de bases de datos de gran tamaño para hallar estructuras útiles. Hay gran número de aplicaciones de minería de datos, como la predicción de valores con base en los ejemplos ya pasados, la búsqueda de asociaciones entre las compras y la agrupación automática de personas y películas.

- La clasificación trata de la predicción de la clase de los ejemplos de prueba utilizando los atributos de los ejemplos de prueba con base en los atributos de los ejemplos de formación y en la clase real de los ejemplos de formación. La clasificación puede utilizarse, por ejemplo, para predecir el valor de crédito de los nuevos solicitantes o para predecir el rendimiento de los estudiantes que solicitan el ingreso en una universidad.

  Hay varios tipos de clasificadores:

  - Los clasificadores de árboles de decisión llevan a cabo la clasificación creando un árbol basado en los ejemplos de formación con hojas que tienen las etiquetas de las clases. Se recorre el árbol para cada ejemplo de prueba hasta hallar una hoja y la clase de esa hoja es la clase predicha.

    Se dispone de varias técnicas para crear árboles de decisión, la mayor parte de ellas basadas en heurística impaciente.
  - Los clasificadores bayesianos son más sencillos de crear que los clasificadores de árboles de decisión y funcionan mejor en caso de que haya valores de los atributos que falten o que sean nulos.

- Las reglas de asociación identifican los elementos que aparecen juntos con frecuencia, por ejemplo, los artículos que el mismo cliente suele comprar. Las correlaciones buscan las desviaciones respecto de los niveles de asociación esperados.
- Otros tipos de minería de datos son el agrupamiento, la minería de texto y la visualización de datos.

Términos de repaso

- Sistemas de ayuda a la toma de decisiones.
- Análisis estadístico.
- Datos multidimensionales.
  - ☐ Atributos de medida.
  - ☐ Atributos de dimensiones.
- Tabulaciones cruzadas.
- Cubo de datos.
- Procesamiento analítico en línea (online analytical processing, OLAP).
  - ☐ Pivotaje.
  - ☐ Cortes de cubos.
  - ☐ Abstracción y concreción.
- OLAP multidimensional (MOLAP).
- OLAP relacional (ROLAP).
- OLAP híbrido (HOLAP).
- Agregación ampliada.
  - ☐ Varianza.
  - ☐ Desviación estándar.
  - ☐ Correlación.
  - ☐ Regresión.
- Funciones de clasificación.
  - ☐ Clasificación.
  - ☐ Clasificación densa.
  - ☐ División mediante.
- Creación de ventanas.
- Almacenes de datos.
  - ☐ Recogida de datos.
  - ☐ Arquitectura dirigida por el origen.
  - ☐ Arquitectura dirigida por el destino.
  - ☐ Limpieza de datos.
    - — Mezcla-purga.
    - — Domiciliación.
  - ☐ Extraer, Transformar, Cargar (ETL).
- Esquemas de almacenamiento.
  - ☐ Tabla de hechos.
  - ☐ Tablas de dimensiones.
  - ☐ Esquema en estrella.
- Minería de datos.
- Predicción.
- Asociaciones.
- Clasificación.
  - ☐ Datos de formación.
  - ☐ Datos de prueba.
- Clasificadores de árboles de decisión.
  - ☐ Atributo de partición.
  - ☐ Condición de partición.
  - ☐ Pureza
    - — Medida de Gini.
    - — Medida de la entropía.
  - ☐ Ganancia de información.
  - ☐ Contenido de la información.
  - ☐ Tasa de ganancia de la información.
  - ☐ Atributo con valores continuos.
  - ☐ Atributo categórico.
  - ☐ División binaria.
  - ☐ División múltiple.
  - ☐ Exceso de ajuste.
- Clasificadores bayesianos.
  - ☐ Teorema de Bayes.
  - ☐ Clasificadores bayesianos ingenuos.
- Regresión.
  - ☐ Regresión lineal.
  - ☐ Ajuste de curvas.
- Reglas de asociación.
  - ☐ Población.
  - ☐ Soporte.
  - ☐ Confianza.
  - ☐ Conjuntos de elementos de gran tamaño.
- Otros tipos de asociación.
- Agrupamiento:
  - ☐ Jerárquico.
  - ☐ Aglomerativo.
  - ☐ Divisivo.
- Minería de texto.
- Visualización de datos.

Ejercicios prácticos

18.1 Muéstrese el modo de expresar **group by cube**(a, b, c, d) utilizando **rollup**; la respuesta sólo debe tener una cláusula **group by**.

18.2 Dada la relación $E(estudiante, asignatura, notas)$, escríbase una consulta para hallar los n mejores estudiantes por sus notas totales, utilizando la clasificación.

18.3 Escríbase una consulta para hallar saldos acumulativos, equivalente a la mostrada en el Apartado 18.2.5, pero sin utilizar las estructuras ampliadas para la creación de ventanas de SQL.

18.4 Considérese la relación *ventas* del Apartado 18.2. Escríbase una consulta en SQL para calcular la operación cubo para la relación, dada la relación de la Figura 18.2. No hay que utilizar el constructor **cube**.

18.5 Descríbanse las ventajas y los inconvenientes de una arquitectura dirigida por el origen para la recolección de datos en los almacenes de datos en comparación con una arquitectura dirigida por el destino.

18.6 Supóngase que hay dos reglas de clasificación, una que dice que la gente con sueldos entre 10.000 € y 20.000 € tienen una calificación de crédito de *buena*, y otra que dice que la gente con sueldos entre 20.000 € y 30.000 € tienen una calificación de crédito de *buena*. Indíquense las condiciones para las que se pueden reemplazar ambas reglas, sin pérdida de información, por una sola regla que diga que las personas con sueldos entre 10.000 € y 30.000 € tienen una calificación de crédito de *buena*.

18.7 Considérese el esquema de la Figura 18.7. Dada una consulta de SQL:1999 para resumir las cifras de ventas y los precios por tienda y por fecha, junto con las jerarquías para tienda y fecha.

Ejercicios

18.8 Para cada una de las funciones de agregación de SQL **sum, count, min**, y **max**, muéstrese el modo de calcular el valor agregado para el conjunto múltiple $S_1 \cup S_2$, dados los valores agregados para los conjuntos múltiples S_1 y S_2.

Basándose en lo anterior hay que dar las expresiones para calcular los valores agregados con el agrupamiento de un subconjunto S de los atributos de la relación $r(A, B, C, D, E)$, dados los valores agregados para agrupamiento de los atributos $T \supseteq S$ para las siguientes funciones de agregación:

a. **sum, count, min** y **max**
b. **avg**
c. Desviación estándar

18.9 Dese un ejemplo de un par de agrupamientos que no puedan expresarse utilizando una sola cláusula **group by** con **cube** y con **rollup**.

18.10 Dada la relación $r(a, b, c)$, muéstrese el modo de utilizar las características ampliadas de SQL para generar un histograma de c frente a a, dividiendo a en veinte particiones de igual tamaño (es decir, que cada partición contenga el cinco por ciento de las tuplas de r, ordenadas según a).

18.11 Considérese el atributo *saldo* de la relación *cuenta*. Escríbase una consulta en SQL para calcular un histograma de los valores de *saldo*, dividiendo el rango desde cero hasta el máximo saldo de una cuenta presente, en tres rangos iguales.

18.12 Créese un clasificador de árboles de decisión con divisiones binarias en cada nodo utilizando las tuplas de la relación $r(A, B, C)$ que se muestra más abajo como datos de formación; el atributo C denota la clase. Muéstrese el árbol final y, con cada nodo, muéstrese la mejor división para cada atributo junto con su valor de ganancia de la información.

$$(1, 2, a), (2, 1, a), (2, 5, b), (3, 3, b), (3, 6, b), (4, 5, b), (5, 5, c), (6, 3, b), (6, 7, c)$$

18.13 Supóngase que la mitad de las transacciones de una tienda de ropa adquieren vaqueros y un tercio de las transacciones de la tienda adquieren camisetas. Supóngase también que la mitad de las transacciones que adquieren vaqueros también adquieran camisetas. Hay que anotar todas las reglas de asociación (no triviales) que se puedan deducir de esta información, dando el soporte y la confianza de cada regla.

18.14 Considérese el problema de hallar conjuntos de artículos de gran tamaño.

- **a.** Descríbase el modo de hallar el soporte de un conjunto dado de conjuntos de elementos mediante una sola exploración de los datos. Supóngase que los conjuntos de artículos y la información asociada, como los recuentos, caben en la memoria.
- **b.** Supóngase que un conjunto de artículos tiene un soporte menor que j. Muéstrese que ningún superconjunto de este conjunto de artículos puede tener un soporte mayor o igual que j.

18.15 Créese un ejemplo pequeño de un conjunto de transacciones que demuestren que aunque muchas transacciones contengan dos artículos (es decir, el conjunto de artículos que contien los dos artículos tiene un gran soporte), la compra de uno de los artículos puede tener una correlación negativa con la compra del otro.

18.16 La organización de partes, capítulos, apartados y subapartados de un libro está relacionada con el agrupamiento. Explíquense la razón y la forma de agrupamiento.

18.17 Propóngase la forma de usar las técnicas de minería en un equipo de deporte usando su deporte preferido como ejemplo.

Notas bibliográficas

Gray et al. [1995] y Gray et al. [1997] describen el operador cubo de datos. Los algoritmos eficientes para calcular cubos de datos se describen en Agarwal et al. [1996], Harinarayan et al. [1996] y Ross y Srivastava [1997]. Las descripciones del soporte ampliado de la agregación en SQL:1999 puede hallarse en los manuales de los productos de sistemas de bases de datos como Oracle y DB2 de IBM. Las definiciones de las funciones estadísticas pueden hallarse en los libros de texto normales de estadística como Bulmer [1979] y Ross [1999].

Los libros de texto Poe [1995] y Mattison [1996] estudian los almacenes de datos. Zhuge et al. [1995] describe el mantenimiento de vistas en un entorno de almacenes de datos. Chaudhuri et al. [2003] describe las técnicas para el encaje difuso en la limpieza de datos, mientras que Sarawagi et al. [2002] describe un sistema para la desduplicación usando técnicas de aprendizaje activo.

Witten y Frank [1999] y Han y Kamber [2000] proporcionan cobertura del nivel de los libros de texto de la minería de datos. Mitchell [1997] es un libro de texto clásico sobre aprendizaje de las máquinas y trata con detalle las técnicas de clasificación. Fayyad et al. [1995] presenta un extenso conjunto de artículos sobre el descubrimiento de conocimiento y la minería de datos. Kohavi y Provost [2001] presenta un conjunto de artículos sobre aplicaciones de la minería de datos para el comercio electrónico.

Agrawal et al. [1993b] proporcionan una primera introducción a la minería de datos en las bases de datos. En Agrawal et al. [1992] y Shafer et al. [1996] se describen algoritmos para el cálculo de clasificadores con conjuntos de formación de gran tamaño; el algoritmo de creación del árbol de decisión descrito en este capítulo se basa en el algoritmo SPRINT de Shafer et al. [1996]. Agrawal et al. [1993a] introdujo la noción de reglas de asociación, mientras que Agrawal y Srikant [1994] presentó un algoritmo eficiente para la minería de reglas de asociación. Los algoritmos para la minería de diferentes formas de las reglas de asociación se describen en Srikant y Agrawal [1996a] y en Srikant y Agrawal [1996b]. Chakrabarti et al. [1998] describen las técnicas para la minería de estructuras temporales sorprendentes.

Las técnicas para la integración de cubos de datos se describen en Sarawagi [2000].

Durante mucho tiempo se ha estudiado el agrupamiento en el área de la estadística, y Jain y Dubes [1988] proporciona cobertura del agrupamiento del nivel de los libros de texto. Ng y Han [1994] describe las técnicas del agrupamiento espacial. Las técnicas de agrupamiento para conjuntos de datos de gran tamaño se describen en Zhang et al. [1996]. Breese et al. [1998] proporciona un análisis empírico de diferentes algoritmos para el filtrado cooperativo. Las técnicas para el filtrado cooperativo de los artículos de noticias se describen en Konstan et al. [1997].

El libro de texto Chakrabarti [2002] proporciona una descripción de recuperación de información, incluyendo un extenso tratamiento de las tareas de minería de datos relacionadas con los datos textuales y de hipertexto, tales como la clasificación y el agrupamiento. Chakrabarti [2000] proporciona una recopilación de técnicas de minería de hipertexto como la clasificación y el agrupamiento de hipertexto.

Herramientas

Se dispone de gran variedad de herramientas para cada una de las aplicaciones que se han estudiado en este capítulo. La mayor parte de los fabricantes de bases de datos proporcionan herramientas OLAP como parte de sus sistemas de bases de datos, o como aplicaciones complementarias. Entre ellas están las herramientas OLAP de Microsoft Corp., Oracle Express e Informix Metacube. La herramienta OLAP Arbor Essbase procede de un fabricante de software independiente. El sitio www.databeacon.com proporciona una demostración en línea de las herramientas OLAP de Databeacon para su empleo en la Web y en orígenes de datos de archivos de texto. Muchas empresas también ofrecen herramientas de análisis para aplicaciones específicas, como la gestión de las relaciones con los clientes.

Los principales fabricantes de bases de datos también ofrecen almacenes de datos acoplados a sus sistemas de bases de datos. Proporcionan ayuda para el modelado de datos, limpieza, carga y consultas. El sitio web www.dwinfocenter.org proporciona información de los almacenes de datos.

También hay una gran variedad de herramientas de minería de datos de propósito general que incluyen las herramientas de minería del SAS Institute, IBM Intelligent Miner y SGI Mineset. Se necesita una gran experiencia para aplicar las herramientas de minería de datos de propósito general para aplicaciones concretas. En consecuencia, se han desarrollado gran número de herramientas de minería para abordar aplicaciones especializadas. El sitio web www.kdnuggets.com proporciona un amplio directorio de software de minería de datos, soluciones, publicaciones, etc.

CAPÍTULO 19

Recuperación de información

Los datos de texto no están estructurados, a diferencia de los datos rígidamente estructurados de las bases de datos relacionales. El término **recuperación de información** suele hacer referencia a la consulta de datos de texto no estructurados. Los sistemas de recuperación de información tienen mucho en común con los sistemas de bases de datos, en especial, el almacenamiento y la recuperación de los datos del almacenamiento secundario. No obstante, el énfasis en el campo de los sistemas de información es diferente del de los sistemas de bases de datos y se centra en problemas como las consultas basadas en palabras clave; la relevancia de los documentos para la consulta y el análisis, la clasificación y el indexado de documentos.

19.1 Visión general

El campo de la **recuperación de información** se ha desarrollado en paralelo con el campo de las bases de datos. En el modelo tradicional usado en el campo de la recuperación de información, ésta se organiza en documentos, dando por supuesto que existe un gran número de documentos. Los datos contenidos en los documentos no están estructurados, no tienen ningún esquema asociado. El proceso de recuperación de información consiste en localizar los documentos importantes, de acuerdo con los datos suministrados por el usuario, como las palabras clave o los documentos de ejemplo.

Web ofrece una manera cómoda de llegar a las fuentes de información y de interactuar con ellas a través de Internet. No obstante, un problema persistente que sufre Web es la explosión de la información almacenada, con pocas indicaciones que ayuden al usuario a localizar la que es interesante. La recuperación de información ha desempeñado un papel crítico para hacer que Web sea una herramienta productiva y útil, especialmente para los investigadores.

Ejemplos tradicionales de sistemas de recuperación de información son los catálogos de bibliotecas en línea y los sistemas de gestión de la documentación en línea como los que almacenan los artículos de los periódicos. Los datos de estos sistemas se organizan como conjuntos de *documentos*; los artículos de los periódicos y las entradas de los catálogos (en los catálogos de las bibliotecas) son ejemplos de documentos. En el contexto de Web, se suele considerar cada página HTML como un documento.

Los usuarios de esos sistemas puede que deseen recuperar un documento concreto o una clase de documentos concreta. Los documentos deseados se suelen describir mediante un conjunto de **palabras clave**—por ejemplo, las palabras clave "sistema de bases de datos" se pueden usar para localizar los libros sobre sistemas de bases de datos y las palabras clave "valores" y "escándalo" se pueden usar para localizar artículos sobre escándalos de los mercados de valores. Cada documento tiene asociado un conjunto de palabras clave y se recuperan los documentos cuyas palabras clave contienen las facilitadas por los usuarios.

La recuperación de información basada en las palabras clave no sólo se puede usar para recuperar datos de texto, sino también para recuperar otros tipos de datos, como los datos de vídeo o de sonido, que

tengan asociadas palabras clave descriptivas. Por ejemplo, una película de vídeo puede tener asociadas palabras clave como el título, el director, los actores, el tipo de película, etc.

Existen varias diferencias entre este modelo y los modelos usados en los sistemas tradicionales de bases de datos.

- Los sistemas de bases de datos gestionan varias operaciones que no se abordan en los sistemas de recuperación de información. Por ejemplo, los sistemas de bases de datos gestionan las actualizaciones y los requisitos transaccionales asociados con el control de concurrencia y de durabilidad. Estos asuntos se consideran menos importantes en los sistemas de información. De manera parecida, los sistemas de bases de datos gestionan información estructurada organizada mediante modelos de datos relativamente complejos (como el modelo relacional o los modelos de datos orientados a los objetos), mientras que los sistemas de recuperación de información han usado tradicionalmente un modelo mucho más sencillo, en el que la información de la base de datos se organiza simplemente como un conjunto de documentos sin estructurar.
- Los sistemas de recuperación de información afrontan varios problemas que no se han abordado de manera adecuada en los sistemas de bases de datos. Por ejemplo, el campo de la recuperación de información ha tratado los problemas de la gestión de documentos sin estructurar, como la búsqueda aproximada mediante palabras clave y de la clasificación de los documentos por el grado estimado de relevancia de cada documento para la consulta.

Los sistemas de recuperación de información suelen permitir las expresiones de consulta formadas mediante palabras clave y las conectivas lógicas *y*, *o* y *no*. Por ejemplo, el usuario puede pedir todos los documentos que contengan las palabras clave "motocicleta *y* mantenimiento", o los documentos que contienen las palabras clave "computadora *o* microprocesador", o incluso los documentos que contienen la palabra clave "computadora *pero no* base de datos". Se da por supuesto que una consulta que contenga palabras clave sin ninguna de las conectivas indicadas usa *y* para conectar las palabras clave de manera implícita.

En la recuperación de **texto completo** se considera que todas las palabras de cada documento son palabras clave. Para los documentos no estructurados la recuperación de texto completo resulta fundamental, ya que puede que no haya información sobre las palabras del documento que son palabras clave. Se usará la palabra **término** para hacer referencia a las palabras de los documentos, ya que todas las palabras son palabras clave.

En su forma más sencilla los sistemas de recuperación de información buscan y devuelven todos los documentos que contienen todas las palabras clave de la consulta, si es que no tiene conectivas; las conectivas se manejan de la forma esperada. Los sistemas más sofisticados estiman la relevancia de cada documento para la consulta, de modo que se puedan mostrar ordenados según esta relevancia. Estos sistemas usan información sobre las apariciones de los términos, así como la información de los hipervínculos, para estimar la relevancia.

19.2 Clasificación por relevancia según los términos

El conjunto de los documentos que satisfacen una expresión de consulta puede ser muy grande; en concreto, hay miles de millones de documentos en Web y la mayor parte de las consultas por palabras clave en los motores de búsqueda de Web hallan centenares de millares de documentos que contienen esas palabras clave. La recuperación de texto completo agrava este problema: cada documento puede contener muchos términos, incluso términos que sólo se mencionan de pasada se tratan de manera equivalente a documentos en los que el término sí es importante. En consecuencia, puede que se recuperen documentos irrelevantes.

Por tanto, los sistemas de recuperación de información estiman la relevancia de cada documento para la consulta y sólo devuelven como respuestas los documentos con una clasificación más elevada. La clasificación por relevancia no es una ciencia exacta, pero hay algunos enfoques aceptados ampliamente.

19.2.1 Clasificación según TF-IDF

El primer punto que hay que abordar es, dado un término concreto t, averiguar la relevancia para ese término de un documento dado d. Un enfoque es usar el número de apariciones del término en el documento como medida de su relevancia, con la suposición de que es probable que los términos importantes se mencionen muchas veces en el documento. El mero recuento del número de apariciones de un término no suele ser un buen indicador: en primer lugar, el número de apariciones depende de la longitud del documento y, en segundo lugar, puede que un documento que contenga diez apariciones del término no tenga diez veces la relevancia de un documento que contenga una sola aparición.

Un modo de medir $TF(d, t)$, la relevancia del documento d para el término t, es

$$TF(d, t) = \log\left(1 + \frac{n(d, t)}{n(d)}\right)$$

donde $n(d)$ denota el número de términos del documento y $n(d, t)$ denota el número de apariciones del término t en el documento d. Obsérvese que esta métrica tiene en cuenta la longitud del documento. La relevancia crece con el número de apariciones del término en el documento, aunque no es directamente proporcional al número de apariciones.

Muchos sistemas refinan esta métrica usando otra información. Por ejemplo, si el término aparece en el título, o en la lista de autores o en el resumen, el documento se considera más importante para el término. De manera parecida, si la primera aparición del término se produce muy avanzado el documento, el documento se puede considerar menos importante que si aparece por primera vez al principio del documento. Los conceptos anteriores pueden formalizarse mediante extensiones de la fórmula que se ha mostrado para $TF(d, t)$. En la comunidad de recuperación de información la relevancia de un documento para un término se denomina **frecuencia del término** (Term Frequency, TF), independientemente de la fórmula concreta usada.

Una consulta C puede contener varias palabras clave. La relevancia de cada documento para una consulta con dos o más palabras clave se estima combinando las medidas de relevancia del documento para cada palabra clave. Una manera sencilla de combinar las medidas es sumarlas. No obstante, no todos los términos usados como palabras clave son iguales. Supóngase que una consulta usa dos términos, uno de los cuales aparece con frecuencia, como "base de datos", y otro que es menos frecuente, como "Silberschatz". Un documento que contenga "Silberschatz" pero no "base de datos" debe clasificarse por encima de otro que contenga "base de datos" pero no "Silberschatz".

Para solucionar este problema se asignan pesos a los términos usando la **frecuencia inversa de los documentos** (Inverse Document Frequency, IDF), definida como

$$IDF(t) = \frac{1}{n(t)}$$

donde $n(t)$ denota el número de documentos (entre los indexados por el sistema) que contienen el término t. La **relevancia** del documento d para el conjunto de términos C se define como

$$r(d, C) = \sum_{t \in C} TF(d, t) * IDF(t)$$

Esta medida puede refinarse aún más si se permite a los usuarios especificar los pesos $p(t)$ de los términos de la consulta, en cuyo caso los pesos especificados por los usuarios se tienen también en cuenta multiplicando $TF(t)$ por $p(t)$ en la fórmula anterior.

Este enfoque de usar la frecuencia de los términos y la frecuencia inversa de los documentos como medida de la relevancia de los documentos se denomina enfoque **TF-IDF**.

Casi todos los documentos de texto (en español) contienen palabras como "y", "o", "un", etc. y, por tanto, estas palabras resultan inútiles para propósitos de consulta, ya que su frecuencia inversa de documentos es extremadamente baja. Los sistemas de recuperación de información definen un conjunto de palabras, denominadas **palabras de parada**, que contiene aproximadamente cien de las palabras más frecuentes, e ignoran esas palabras al indexar los documentos. Esas palabras no se usan como palabras clave, y se descartan si se hallan entre las palabras proporcionadas por los usuarios.

Otro factor que se tiene en cuenta cuando una consulta contiene varios términos es la **proximidad** de los términos en el documento. Si los términos aparecen cercanos entre sí en el documento, el documento se clasificará en una posición más elevada que si aparecen muy separados. La fórmula de $r(d, C)$ puede modificarse para tener en cuenta la proximidad.

Dada una consulta C, el trabajo del sistema de recuperación de información es devolver documentos en orden descendente de relevancia para C. Dado que puede haber un número muy grande de documentos que sean relevantes, los sistemas de recuperación de información suelen devolver únicamente los primeros documentos con el grado de relevancia estimada más elevado y permiten que los usuarios soliciten más documentos de manera interactiva.

19.2.2 Recuperación basada en la semejanza

Algunos sistemas de recuperación de información permiten la **recuperación basada en la semejanza**. En este caso, el usuario puede dar al sistema el documento A y pedirle que recupere documentos que sean "semejantes" a A. La semejanza de un documento con otro puede definirse, por ejemplo, con base en los términos comunes. Un enfoque es hallar k términos de A con los valores más elevados de $TF(A, t) * IDF(t)$, y usar esos k términos como consulta para hallar la relevancia de otros documentos. Los términos de la consulta se pesan mediante $TF(A, t) * IDF(t)$.

Más genéricamente, la semejanza de los documentos se define mediante la métrica de **semejanza del coseno**. Sean los términos que aparecen en cualquiera de los dos documentos $t_1, t_2, \ldots, t_n$. Sea $r(d, t) = TF(d, t) * IDF(t)$. Entonces, la métrica de semejanza del coseno entre los documentos d y e se define como

$$\frac{\sum_{i=1}^{n} r(d, t_i) r(e, t_i)}{\sqrt{\sum_{i=1}^{n} r(d, t_i)^2} \sqrt{\sum_{i=1}^{n} r(e, t_i)^2}}$$

Se puede comprobar fácilmente que la métrica de la semejanza del coseno de un documento consigo mismo es 1, mientras que entre dos documentos que no compartan ningún término es 0.

La denominación "semejanza del coseno" procede del hecho de que la fórmula anterior calcula el coseno del ángulo entre dos vectores, cada uno de los cuales representa a uno de los documentos, definido de la manera siguiente. Supóngase que hay n palabras en total en todos los documentos que se vayan a considerar. Se define un espacio n-dimensional, con cada palabra como una de las dimensiones. El documento d se representa mediante un punto de este espacio, con el valor de la coordenada i-ésima del punto igual a $r(d, t_i)$. El vector del documento d conecta el origen (todas las coordenadas iguales a cero) con el punto que representa al documento. El modelo de documentos como puntos y vectores de un espacio n-dimensional se denomina **modelo de espacio vectorial**.

Si el conjunto de documentos semejantes al documento A de la consulta es grande, puede que el sistema sólo presente al usuario unos cuantos documentos semejantes, le permita escoger los más destacados e inicie una nueva búsqueda basada en la semejanza con A y con los documentos seleccionados. Es posible que el conjunto de documentos resultante sea lo que el usuario pretendía hallar. Esta idea se denomina **realimentación de la relevancia**.

La realimentación de la relevancia se puede usar también para ayudar a los usuarios a encontrar los documentos importantes de entre un conjunto grande de documentos que coinciden con las palabras clave de consulta dadas. En esta situación se puede permitir que los usuarios identifiquen uno o varios de los documentos devueltos como importantes; el sistema usará los documentos identificados para hallar otros semejantes. Es probable que el conjunto de documentos resultante sea lo que el usuario pretendía hallar. Una alternativa al enfoque de la realimentación de la relevancia es exigir a los usuarios que modifiquen la consulta añadiendo más palabras clave; la realimentación de la relevancia puede resultar más sencilla de usar, además de dar como respuesta un conjunto final de documentos más adecuado.

Para mostrar al usuario un conjunto representativo de documentos cuando el número total es muy grande, el sistema de búsqueda puede agrupar los documentos, basándose en la semejanza del coseno. La agrupación ya se ha descrito en el Apartado 18.4.5, y se han desarrollado varias técnicas para agrupar los conjuntos de documentos. Véanse las notas bibliográficas para hallar referencias sobre más información relativa a la agrupación.

19.3 Relevancia según los hipervínculos

Los primeros motores de búsqueda Web clasificaban los documentos usando sólo medidas de relevancia basadas en TF-IDF como las descritas en el Apartado 19.2. Sin embargo, estas técnicas presentaban algunas limitaciones cuando se usaban sobre conjuntos muy grandes de documentos, como el conjunto de todas las páginas Web. En concreto, muchas páginas Web tienen todas las palabras clave especificadas en las consultas típicas del motor de búsqueda; además, algunas de las páginas que los usuarios desean obtener como respuesta a menudo sólo tienen unas cuantas apariciones de los términos de la consulta y no consiguen una puntuación TF-IDF muy elevada.

No obstante, los investigadores pronto se dieron cuenta de que las páginas Web contienen información muy importante de la que carecen los documentos de texto sencillo, por ejemplo, los hipervínculos. Los hipervínculos se pueden aprovechar para obtener una mejor clasificación por relevancia; en concreto, la clasificación de relevancia de una página está muy influida por los hipervínculos que apuntan *a* esa página. En este apartado se estudia la manera en que se usan los hipervínculos para la clasificación de las páginas Web.

19.3.1 Clasificación por popularidad

La idea básica de la **clasificación por popularidad** (también denominada **clasificación por prestigio**) es hallar páginas que sean populares, y clasificarlas por encima de otras páginas que contengan las palabras clave especificadas. Dado que la mayor parte de las búsquedas pretenden hallar información de las páginas más populares, clasificar esas páginas en posiciones más elevadas suele ser una buena idea. Por ejemplo, el término "google" puede aparecer en gran número de páginas, pero la página google.com es el sitio más popular de entre las páginas que contienen el término "google". Por tanto, la página google.com debe clasificarse como la respuesta más importante de las consultas que consistan en el término "google".

Las medidas tradicionales de la relevancia de una página, como las basadas en TF-IDF que se vieron en el Apartado 19.2, pueden combinarse con la popularidad de la página para obtener una medida global de la relevancia de la página para la consulta. Las páginas con el valor más elevado de relevancia global se devuelven como primeras respuestas de la consulta.

Esto suscita la pregunta del modo de definir y averiguar la popularidad de una página. Una manera es hallar la cantidad de veces que se tiene acceso a la página y usar ese número como medida de la popularidad de ese sitio. Sin embargo, la obtención de esa información es imposible sin la cooperación del sitio, y no les es factible implementarla a los motores de búsqueda Web.

Una alternativa muy efectiva es usar los hipervínculos a la página como medida de su popularidad. Mucha gente tiene archivos de marcadores que contienen vínculos a sitios que usan con frecuencia. Se puede deducir que los sitios que aparecen en gran número de archivos de marcadores son muy populares. Los archivos de marcadores se suelen almacenar de manera privada y no suelen ser accesibles en Web. No obstante, muchos usuarios tienen páginas Web con vínculos a sus páginas Web favoritas. Muchos sitios Web también tienen vínculos a otros sitios relacionados, que también se pueden usar para deducir la popularidad de los sitios vinculados. Los motores de búsqueda de Web pueden capturar páginas Web (mediante un proceso denominado batida (crawling), que se describe en el Apartado 19.7) y analizarlas para hallar los vínculos entre unas y otras.

Una primera solución para estimar la popularidad de las páginas es usar el número de páginas que enlazan con ellas como medida de su popularidad. Sin embargo, esto en sí mismo tiene el inconveniente de que muchos sitios tienen varias páginas útiles, pero los vínculos externos a menudo sólo apuntan a la página raíz de cada sitio. La página raíz, a su vez, tiene vínculos con otras páginas del sitio. Se puede deducir, entonces, erróneamente, que esas otras páginas no son muy populares y tendrán una clasificación baja en las respuestas a las consultas.

Una alternativa es asociar la popularidad con los sitios, en vez de con las páginas. Todas las páginas de un sitio dado tienen la popularidad de ese sitio, y las páginas distintas de la página raíz de los sitios populares se benefician también de la popularidad de esos sitios. Sin embargo, surge la pregunta de qué constituye un sitio. Por lo general, el prefijo de la dirección de Internet del URL de una página constituye el sitio correspondiente a esa página. No obstante, hay muchos sitios que albergan gran número de páginas muy poco relacionadas entre sí, como los servidores de páginas iniciales de las universidades y

de los portales Web como groups.yahoo.com o tripod.com. Para estos sitios, la popularidad de una parte del sitio no implica la popularidad de otras partes del mismo sitio.

Una alternativa más sencilla es permitir la *transferencia de prestigio* desde las páginas populares a las páginas con las que se vinculan. De acuerdo con este esquema, a diferencia con el principio de "un hombre, un voto" de la democracia, el vínculo desde una página popular x a una página y se considera que confiere más prestigio a la página y que un vínculo desde la página no tan popular z[1].

Esta definición de popularidad es, de hecho, circular, ya que la popularidad de las páginas queda definida por la popularidad de otras páginas, y puede que haya ciclos de vínculos entre las páginas. No obstante, la popularidad de las páginas puede definirse mediante un sistema de ecuaciones lineales simultáneas, que pueden resolverse mediante las técnicas de tratamiento de matrices. Las ecuaciones lineales se definen de manera que tengan una solución única y bien definida.

Es interesante destacar que la idea básica subyacente a las clasificaciones de popularidad es, en realidad, bastante antigua y surgió por primera vez en la teoría de redes sociales desarrollada por los sociólogos de los años cincuenta del siglo veinte. En el contexto de las redes sociales el objetivo era definir el prestigio de las personas. Por ejemplo, el rey de España tiene un elevado prestigio, ya que gran cantidad de gente lo conoce. Si alguien es conocido por varias personas de prestigio, también tendrá un prestigio elevado, aunque no sea conocido por un número de personas tan grande. El uso de conjuntos de ecuaciones lineales para definir las medidas de popularidad también procede de estos trabajos.

19.3.2 PageRank

El motor de búsqueda Web Google introdujo **PageRank**, que es una medida de la popularidad de las páginas basada en la popularidad de las páginas que vinculan con ellas. El uso de la medida de popularidad PageRank para clasificar las respuestas a las consultas de resultados, que mejoran las de las técnicas de clasificación usadas anteriormente, hizo que Google pasara a ser el motor de búsqueda más usado en un periodo de tiempo bastante breve.

PageRank se puede comprender de manera intuitiva si se usa un **modelo de recorrido aleatorio**. Supóngase que una persona que navega por Web lleva a cabo un paseo aleatorio (transversal) de las páginas Web de la manera siguiente: El primer paso parte de una página Web aleatoria y, en cada paso, el paseante aleatorio emprende una de las acciones siguientes. Con una probabilidad δ el paseante salta a una página Web escogida aleatoriamente y, con una probabilidad de $1 - \delta$, escoge aleatoriamente uno de los vínculos externos de la página Web en la que se halla y sigue ese vínculo. El valor de PageRank de cada página es, por tanto, la probabilidad de que el paseante aleatorio visite la página en cualquier momento dado.

Téngase en cuenta que es más probable que se visiten las páginas a las que apuntan muchas páginas Web y, por tanto, tendrán un valor de PageRank más elevado. De manera parecida, las páginas a las que apuntan páginas con un valor de PageRank elevado también tendrán mayor probabilidad de que las visiten y, por tanto, tendrán un valor de PageRank más elevado.

PageRank se puede definir mediante un conjunto de ecuaciones lineales, de la manera siguiente. En primer lugar, se otorga a cada página Web un identificador entero. La matriz de las probabilidades de salto T se define con $T[i, j]$ definida como la probabilidad de que un paseante aleatorio que siga un vínculo externo de la página i siga el vínculo a la página j. Suponiendo que cada vínculo de i tiene igual probabilidad de que lo sigan, $T[i, j] = 1/N_i$, donde N_i es el número de vínculos externos de la página i. La mayor parte de las entradas de of T son 0 y se representa mejor mediante una lista de adyacentes. Por tanto, el valor de PageRank $P[j]$ para cada página j se puede definir como

$$P[j] = \delta/N + (1 - \delta) * \sum_{i=1}^{N} (T[i, j] * P[i])$$

donde δ es una constante entre 0 y 1 y N es el número de páginas; δ representa la probabilidad de que cada paso del paseo aleatorio sea un salto.

1. Esto es parecido, en cierto sentido, a conceder un peso extra al aval que personas famosas (como las estrellas de cine) conceden a ciertos productos, por lo que su importancia es discutible, aunque resulta efectivo y en la práctica se usa mucho.

El conjunto de ecuaciones generado de esta manera se suele resolver mediante técnicas iterativas, comenzando con cada $P[i]$ definida como $1/N$. Cada paso de la iteración calcula valores nuevos de cada $P[i]$ usando los valores de P de la iteración anterior. La iteración se detiene cuando la variación máxima entre iteraciones de cualquier valor de $P[i]$ queda por debajo de un valor de corte prefijado.

19.3.3 Otras medidas de popularidad

Las medidas básicas de popularidad como PageRank desempeñan un papel importante en la clasificación de las respuestas a las consultas, pero no son, de ningún modo, el único factor. La puntuación TF-IDF de cada página se usa para evaluar su relevancia para las palabras clave de la consulta y se deben combinar con la clasificación de popularidad. También se deben tener en cuenta otros factores, para superar las limitaciones de PageRank y de otras medidas de popularidad relacionadas.

Un inconveniente del algoritmo de PageRank es que asigna una medida de popularidad que no tiene en cuenta las palabras clave de la consulta. Por ejemplo, es probable que la página google.com tenga un valor de PageRank muy elevado, ya que muchos sitios contienen vínculos con ella. Supóngase que contiene una palabra mencionada de pasada, como "Stanford" (la página de búsqueda avanzada de Google contenía de hecho la palabra Stanford, al menos hasta principios de 2005). La búsqueda con la palabra clave Stanford devolvería google.com como respuesta mejor clasificada, por delante de respuestas más relevantes, como la página Web de la Universidad de Stanford.

Una solución de este problema muy usada es emplear palabras clave en el texto ancla de los vínculos a una página para evaluar para qué temas es importante la página. El texto ancla de los vínculos consiste en el texto que aparece dentro de la etiqueta a href de HTML. Por ejemplo, el texto ancla del vínculo

 Stanford University

es "Stanford University". Si muchos vínculos a stanford.edu contienen la palabra Stanford en su texto ancla, se puede considerar muy importante para la palabra la palabra clave Stanford. El texto cercano al texto ancla también se puede tomar en consideración; por ejemplo, un sitio Web puede contener el texto "Ésta es la página Web de la Universidad de Stanford", pero puede haber usado sólo la palabra "ésta" como texto ancla en el vínculo al sitio Web de la Universidad de Stanford.

La popularidad basada en el texto ancla se combina con otras medidas de la popularidad y con las medidas TF-IDF para obtener una clasificación global de las respuestas a las consultas. Téngase en cuenta que la mayor parte de los motores de búsqueda no revelan la manera en que calculan las clasificaciones de relevancia; consideran que revelar sus técnicas de clasificación permitiría a sus competidores alcanzarlos y facilitaría la labor de "contaminación de los motores de búsqueda", lo que daría lugar a una menor calidad de los resultados. La contaminación de los motores de búsqueda se describe con más detalle posteriormente en este apartado.

Un enfoque alternativo a la toma en cuenta de las palabras clave al definir la popularidad es calcular una medida de la popularidad que use *sólo* las páginas que contienen las palabras clave de la consulta, en lugar de calcular la popularidad usando todas las páginas Web disponibles. Este enfoque resulta más costoso, ya que el cálculo de las clasificaciones de popularidad tiene que llevarse a cabo de manera dinámica cuando se recibe la consulta, mientras que PageRank se calcula una vez de manera estática y se reutiliza para todas las consultas. Los motores de búsqueda en Web que manejan miles de millones de consultas diarias no se pueden permitir emplear tanto tiempo en contestar cada consulta. En consecuencia, aunque este enfoque puede dar mejores respuestas, no se usa mucho.

El algoritmo HITS se basaba en la idea anterior de hallar primero las páginas que contienen las palabras clave de la consulta y calcular luego una medida de la popularidad usando sólo ese conjunto de páginas relacionadas. Además, introdujo el concepto de *nodos* y de *autoridades*. Un **nodo** es una página que almacena vínculos con muchas páginas; no contiene información real sobre el asunto, pero apunta a páginas que sí que la contienen. Por el contrario, una **autoridad** es una página que contiene información real sobre un asunto, aunque puede que no almacene vínculos con muchas páginas relacionadas. Cada página recibe un valor de prestigio como nodo (*prestigio-nodo*) y otro valor de prestigio como autoridad (*prestigio-autoridad*). Las definiciones de prestigio, como ya se ha visto, son cíclicas y vienen dadas por un conjunto de ecuaciones lineales simultáneas. Las páginas obtienen mayor prestigio-nodo si apuntan a muchas páginas con un elevado prestigio-autoridad, mientras que reciben un elevado prestigio-

autoridad si apuntan a ellas muchas páginas con un elevado prestigio-nodo. Dada una consulta, las páginas con prestigio-autoridad más elevado se clasifican por encima de las demás páginas. Véanse las notas bibliográficas para hallar referencias que ofrezcan más detalles.

La **contaminación de los motores de búsqueda** hace referencia a la práctica de crear páginas Web, o conjuntos de páginas Web, diseñados para obtener una clasificación de relevancia elevada para algunas consultas, aunque los sitios no sean realmente populares. Por ejemplo, puede que un sitio sobre viajes desee obtener una clasificación elevada para las consultas con la palabra clave "viaje". Puede obtener puntuaciones TF-IDF elevadas mediante la repetición de la palabra "viaje" muchas veces en su página[2]. Incluso los sitios sin relación con los viajes, como los sitios pornográficos, pueden hacer lo mismo, y obtener clasificaciones elevadas para las consultas sobre la palabra viaje. De hecho, este tipo de contaminación de TF-IDF era frecuente en los primeros días de la búsqueda en Web, y hubo una batalla constante entre este tipo de sitios y los motores de búsqueda que intentaban detectar la contaminación e impedir que consiguieran clasificaciones elevadas.

Los esquemas de clasificación de la popularidad como PageRank hacen más difícil la labor de contaminación de los motores de búsqueda, ya que la mera repetición de palabras para obtener puntuaciones TF-IDF elevada ya no es suficiente. No obstante, incluso estas técnicas se pueden contaminar, mediante la creación de un conjunto de páginas Web que se apunten entre sí, lo que incrementa su clasificación de popularidad. Se han propuesto técnicas como el uso de los sitios en lugar de las páginas como unidad de clasificación (con las probabilidades normalizadas de salto correspondientes) para evitar algunas técnicas de contaminación. La guerra entre los contaminadores de los motores de búsqueda y los motores de búsqueda continúa incluso hoy en día.

El enfoque de nodos y autoridades del algoritmo HITS es más susceptible de contaminación. El contaminador puede crear una página Web que contenga vínculos con autoridades auténticas de un asunto dado y obtener en consecuencia una puntuación de nodo elevada. Además, Web del contaminador incluye vínculos a páginas que desea popularizar, que puede que no tengan ninguna relevancia para ese asunto. Como a estas páginas las apunta una página con una puntuación de nodo elevada, obtienen una puntuación de autoridad elevada pero inmerecida.

19.4 Sinónimos, homónimos y ontologías

Considérese el problema de la localización de documentos sobre el mantenimiento de motocicletas, usando la consulta "mantenimiento de motocicletas". Supóngase que las palabras clave de cada documento son las palabras del título y el nombre de los autores. El documento titulado *Reparación de motocicletas* no se recuperaría, ya que la palabra "mantenimiento" no aparece en el título.

Se puede resolver este problema haciendo uso de los **sinónimos**. Cada palabra puede tener definido un conjunto de sinónimos, y la aparición de una palabra puede ser sustituida por la *disyunción* de todos sus sinónimos (incluida la propia palabra). Así, la consulta "motocicleta *y* reparación" puede sustituirse por "motocicleta *y* (reparación *o* mantenimiento)". Esta consulta sí hallaría el documento deseado.

Las consultas basadas en las palabras clave también se ven afectadas por el problema contrario, el de los **homónimos**, es decir, las palabras con varios significados. Por ejemplo, la palabra "objeto" tiene significados diferentes como nombre y como verbo. La palabra "tabla" puede hacer referencia a una pieza de madera o a una tabla de una base de datos relacional.

De hecho, un riesgo de usar los sinónimos para ampliar las consultas es que los sinónimos pueden tener, a su vez, significados diferentes. Por ejemplo, "sustento" es sinónimo de uno de los significados de la palabra "mantenimiento", pero tiene un significado diferente del pretendido por el usuario en la consulta "mantenimiento de motocicletas". Se recuperarán los documentos que utilicen un significado implícito alternativo. El usuario se preguntará el motivo de que el sistema considerara que alguno de los documentos recuperados (que, por ejemplo, usa la palabra "sustento") era importante, si no contiene ni las palabras clave especificadas por el usuario ni palabras cuyo significado implícito en el documento sea

2. Las palabras repetidas en las páginas Web pueden confundir a los usuarios; los contaminadores pueden abordar este problema entregando páginas diferentes a los motores de búsqueda y al resto de los usuarios, para el mismo URL, o haciendo invisibles las palabras repetidas, por ejemplo, aplicando a esas palabras un formato de fuente blanca pequeña sobre fondo blanco.

sinónimo de las palabras clave especificadas. Por tanto, no es buena idea usar sinónimos para ampliar las consultas sin comprobarlos antes con el usuario.

Un enfoque mejor del problema anterior es que el sistema comprenda el *concepto* que representa cada palabra del documento y, de manera parecida, comprenda el concepto que busca el usuario y devuelva los documentos que aborden los conceptos en los que está interesado el usuario. Los sistemas que soportan las **consultas basadas en conceptos** tienen que analizar cada documento para deshacer la ambigüedad de cada palabra del documento y sustituirla por el concepto que representa; la ruptura de la ambigüedad suele hacerse examinando las palabras circundantes en el documento. Por ejemplo, si el documento contiene palabras como base de datos o consulta, es probable que la palabra tabla sea sustituida por el concepto "tabla: datos", mientras que, si el documento contiene palabras como muebles, silla o madera cerca de la palabra tabla, ésta sea sustituida por el concepto "tabla: carpintería". La ruptura de la ambigüedad basada en las palabras cercanas suele ser más difícil con las consultas de los usuarios, ya que las consultas contienen muy pocas palabras, por lo que los sistemas de consultas basadas en los conceptos suelen ofrecer varios conceptos alternativos al usuario, que escoge uno o varios antes de que la búsqueda se reanude.

Las consultas basadas en los conceptos tienen varias ventajas; por ejemplo, una consulta en un idioma puede recuperar documentos en otros idiomas, siempre y cuando se hallen relacionados con el mismo concepto. Se pueden usar luego mecanismos de traducción automática si el usuario no comprende el idioma en que está escrito el documento. No obstante, la sobrecarga del procesamiento de los documentos para deshacer la ambigüedad de las palabras es muy elevada si se trabaja con miles de millones de documentos. Por tanto, los motores de búsqueda de Internet no suelen soportar las consultas basadas en los conceptos. No obstante, se han creado sistemas de consultas basadas en los conceptos y se han usado para otros conjuntos documentales de gran tamaño.

Las consultas basadas en los conceptos se pueden ampliar aún más aprovechando las jerarquías de conceptos. Por ejemplo, supóngase que alguien formula la consulta "animales voladores"; los documentos que contengan información sobre los "mamíferos voladores" son, ciertamente, importantes, ya que los mamíferos son animales. Sin embargo, los dos conceptos no son iguales y la mera coincidencia de conceptos no permite que se devuelva el documento como respuesta. Los sistemas de consultas basadas en los conceptos pueden soportar la recuperación de documentos basada en las jerarquías de conceptos.

Las **ontologías** son estructuras jerárquicas que reflejan las relaciones entre los conceptos. La más frecuente es la relación **es/son**; por ejemplo, los leopardos son mamíferos, y los mamíferos son animales. También son posibles otras relaciones, como *parte-de*; por ejemplo, el ala del avión es parte del avión.

El sistema WordNet define gran variedad de conceptos con palabras asociadas (denominadas *synset*, conjunto de sinónimos—synonym set—, en la terminología WordNet). Las palabras asociadas a un synset son sinónimos del concepto; por supuesto, cada palabra puede ser sinónima de varios conceptos diferentes. Además de los sinónimos, WordNet define homónimos y otras relaciones. En concreto, las relaciones es y parte—de que define conectan conceptos y definen de hecho ontologías. El proyecto Cyc fue otro intento de crear ontologías.

Además de las ontologías que abarcan todo el idioma, se han definido ontologías para áreas concretas para tratar la terminología importante para esas áreas. Por ejemplo, se han creado ontologías para normalizar los términos usados en algunos negocios; se trata de un paso importante en la creación de una infraestructura normalizada para el tratamiento del procesamiento de pedidos y otros flujos de datos internos de las organizaciones.

También es posible crear ontologías que vinculen varios idiomas. Por ejemplo, se han creado WordNets para diferentes idiomas, y los conceptos comunes entre los diferentes idiomas se pueden vincular entre sí. Este tipo de sistemas se puede usar para la traducción de textos. En el contexto de la recuperación de información, las ontologías multilingües se pueden usar para implementar búsquedas basadas en los conceptos en documentos escritos en varios idiomas.

19.5 Creación de índices de documentos

Una estructura de índices efectiva es importante para el procesamiento eficiente de las consultas en los sistemas de recuperación de información. Se pueden localizar los documentos que contienen las palabras clave especificadas de manera efectiva usando un **índice invertido**, que relaciona cada palabra

clave C_i con el conjunto S_i de (los identificadores de) los documentos que contienen C_i. Para dar soporte a la clasificación por relevancia basada en la proximidad de las palabras clave estos índices pueden proporcionar no sólo los identificadores de los documentos, sino también una lista de las ubicaciones dentro del documento en las que aparecen las palabras clave. Estos índices hay que almacenarlos en disco, y cada lista S_i puede abarcar varias páginas de disco. Para minimizar el número de operaciones de E/S para la recuperación de cada lista S_i, el sistema intentará guardar cada lista S_i en un conjunto de páginas consecutivas del disco, de modo que toda la lista se pueda recuperar con una sola búsqueda en el disco. Se pueden usar índices de árbol B^+ para asignar cada palabra clave C_i a su lista invertida asociada S_i.

La operación *y* halla los documentos que contienen todas las palabras clave del conjunto de palabras clave especificado $C_1, C_2, \ldots, C_n$. La operación *y* se implementa recuperando primero los conjuntos de identificadores de documentos $S_1, S_2, \ldots, S_n$ de todos los documentos que contienen las palabras clave respectivas. La intersección $S_1 \cap S_2 \cap \cdots \cap S_n$, de los conjuntos de documentos da los identificadores de los documentos del conjunto de documentos deseado. La operación *o* da el conjunto de todos los documentos que contienen al menos una de las palabras clave $C_1, C_2, \ldots, C_n$. La operación *o* se implementa calculando la unión, $S_1 \cup S_2 \cup \cdots \cup S_n$, de los conjuntos. La operación *no* halla los documentos que no contienen la palabra clave especificada C_i. Dado un conjunto de identificadores S, se pueden eliminar los documentos que contienen la palabra clave especificada C_i tomando la diferencia $S - S_i$, donde S_i es el conjunto de identificadores de los documentos que contienen la palabra clave C_i.

Dado un conjunto de palabras clave de una consulta, muchos sistemas de recuperación de información no insisten en que los documentos recuperados contengan todas las palabras clave (a menos que se utilice de manera explícita una operación *y*). En ese caso, se recuperan todos los documentos que contienen como mínimo una de las palabras clave (como en la operación *o*), pero se clasifican de acuerdo con la medida de su relevancia.

Para usar para la clasificación la frecuencia de los términos la estructura de índices también debe conservar el número de veces que aparece cada término en cada documento. Para reducir este esfuerzo se puede usar una representación comprimida con sólo unos pocos bits, que aproxima la frecuencia de los términos. El índice también debe almacenar la frecuencia de documentos de cada término (es decir, el número de documentos en que aparece cada término).

La lista S_i se puede ordenar según la clasificación de popularidad (y, en segundo lugar, para los documentos con la misma clasificación de popularidad, por identificador de documento). Luego se puede usar una simple mezcla para calcular las operaciones *y* y *o*. Los resultados con clasificación de popularidad más elevada aparecerán cerca de la cabeza de las listas. En el caso de la operación *y*, si se ignora la contribución de TF-IDF a la puntuación de relevancia y se exige simplemente que el documento contenga las palabras clave dadas, la mezcla puede detenerse una vez obtenidas C respuestas, si el usuario sólo pide las C primeras respuestas.

19.6 Medida de la efectividad de la recuperación

Cada palabra clave puede estar incluida en gran número de documentos; por tanto, una representación compacta resulta fundamental para mantener bajas las necesidades de espacio del índice. Así, los conjuntos de documentos de cada palabra clave se conservan en forma comprimida. Para ahorrar espacio de almacenamiento a veces se almacena el índice de modo que la recuperación es aproximada; puede que no se recuperen unos pocos documentos de relevancia (lo que se denomina un **rechazo falso** o un **falso negativo**), o puede que se recuperen unos pocos documentos sin relevancia (lo que se denomina un **falso positivo**). Una buena estructura de índice no tiene *ningún* rechazo falso, pero puede que permita algunos falsos positivos; el sistema puede filtrarlos posteriormente mirando las palabras clave que contienen realmente. En el indexado de Webs los falsos positivos tampoco son deseables, ya que puede que el documento real no sea accesible rápidamente para su filtrado.

Se usan dos métricas para medir la calidad con que los sistemas de recuperación de información pueden contestar las consultas. La primera, la **precisión**, mide el porcentaje de los documentos recuperados que son verdaderamente importantes para la consulta. La segunda, la **recuperación** (*recall*), mide el porcentaje de los documentos importantes para la consulta que se ha recuperado. Lo ideal sería que ambas fueran del cien por cien.

La precisión y la recuperación también son medidas importantes para la comprensión de la calidad de una determinada estrategia de clasificación de los documentos. Las estrategias de clasificación pueden dar lugar a falsos negativos y a falsos positivos, pero en un sentido más sutil.

- Pueden producirse falsos negativos al clasificar los documentos porque los documentos importantes obtengan clasificaciones bajas. Si el sistema capturara todos los documentos hasta incluir los que tienen clasificaciones muy bajas, habría muy pocos falsos negativos. Sin embargo, los usuarios rara vez miran más allá de las primeras decenas de documentos devueltos y, por tanto, puede que pasen por alto documentos importantes porque no estén clasificados entre los primeros. Lo que sea exactamente un falso negativo depende del número de documentos que se examinen. Por tanto, en lugar de tener un solo número como medida de la recuperación, se puede medir la recuperación como función del número de documentos capturados.
- Pueden producirse falsos positivos porque documentos sin relevancia obtengan clasificaciones más elevadas que algunos documentos importantes. Esto también depende del número de documentos que se examinen. Una posibilidad es medir la precisión como función del número de documentos extraídos.

Una opción mejor y más intuitiva para la medida de la precisión es medirla como función de la recuperación. Con esta medida combinada tanto la precisión como la recuperación pueden calcularse en función del número de documentos, si hace falta.

Por ejemplo, se puede decir que con una recuperación del cincuenta por ciento la precisión es del setenta y cinco por ciento, mientras que con una recuperación del setenta y cinco por ciento la precisión ha caído hasta el sesenta por ciento. En general, se puede dibujar una gráfica que relacione la precisión con la recuperación. Estas medidas pueden calcularse para las diferentes consultas y promediarse para un conjunto de consultas en las pruebas de homologación de la calidad de las consultas.

Otro problema más de la medida de la precisión y de la recuperación consiste en el modo de definir los documentos que son realmente importantes y los que no lo son. De hecho, decidir si un documento es importante o no lo es exige la comprensión del lenguaje natural y de las pretensiones de la consulta. Los investigadores, por tanto, han creado conjuntos de documentos y de consultas y han marcado manualmente los documentos como importantes o no importantes para las consultas. Se pueden ejecutar diferentes sistemas de clasificación con estos conjuntos de documentos para medir la precisión y la recuperación medias de varias consultas.

19.7 Motores de búsqueda en Web

Los programas denominados **Web crawlers** (batidores Web) localizan y reúnen información de Web. Siguen de manera recursiva los hipervínculos presentes en los documentos conocidos para hallar otros documentos. Los batidores recuperan los documentos y añaden la información hallada en ellos a índices combinados; generalmente, los documentos no se almacenan, aunque algunos motores de búsqueda guardan en la caché una copia del documento para ofrecer a sus clientes un acceso más rápido a los documentos.

Dado que el número de documentos de Web es muy grande, no es posible recorrer toda Web en un periodo corto de tiempo; y, de hecho, todos los motores de búsqueda cubren únicamente algunas partes de Web, no toda ella, y sus batidores pueden tardar semanas o meses en llevar a cabo un solo recorrido de todas las páginas que abarcan. Suele haber muchos procesos que se ejecutan en varias máquinas, implicadas en los recorridos. Una base de datos almacena el conjunto de vínculos (o de sitios) que hay que recorrer; esa base de datos asigna los vínculos que parten de este conjunto a cada proceso batidor. Los vínculos nuevos hallados durante cada recorrido se añaden a la base de datos, y se pueden recorrer posteriormente si no se recorren de manera inmediata. Las páginas halladas durante cada recorrido también se pasan al sistema de cálculo de prestigio y de indexado, que puede que se ejecute en una máquina diferente. Hay que volver a capturar las páginas (es decir, volver a seguir los vínculos) de manera periódica para obtener información actualizada y descartar los sitios que ya no existen, de modo que la información del índice de búsqueda se mantenga razonablemente actualizada.

Los propios sistemas de cálculo de prestigio y de indexado se ejecutan en paralelo en varias máquinas. No es buena idea añadir las páginas al mismo índice que se usa para las consultas, ya que eso exige el

control de concurrencia del índice, y afecta al rendimiento de las consultas y de las actualizaciones. En lugar de eso, se usa una copia del índice para responder a las consultas mientras otra copia se actualiza con las páginas recién recorridas. A intervalos periódicos se intercambian las copias y se actualiza la más antigua mientras la copia nueva se usa para las consultas.

Para soportar tasas de consultas muy elevadas se pueden mantener los índices en la memoria principal y usar varias máquinas; el sistema encamina de manera selectiva las consultas a las diferentes máquinas para equilibrar la carga de trabajo entre ellas. Los motores de búsqueda más populares suelen tener decenas de miles de máquinas que llevan a cabo las diferentes tareas de exploración, indexado y respuesta a las consultas de los usuarios.

19.8 Recuperación de información y datos estructurados

Aunque los sistemas de recuperación de información se diseñaron originalmente para hallar documentos de texto relacionados con las consultas, hay una necesidad creciente de sistemas que intenten comprender los documentos (en un grado limitado) y contestar preguntas de acuerdo con esa comprensión (limitada). Un enfoque es la creación de información estructurada a partir de documentos no estructurados y responder a preguntas de acuerdo con esa información estructurada. Otro enfoque aplica las técnicas del lenguaje natural para hallar documentos relacionados con la pregunta (planteada en lenguaje natural) y devolver fragmentos importantes de los documentos como respuesta a esa pregunta.

19.8.1 Extracción de información

Los sistemas de **extracción de información** pasan la información en forma de texto a otra forma más estructurada. Por ejemplo, un anuncio inmobiliario puede describir los atributos de una casa en forma de texto, como "casa en La Moraleja con dos dormitorios y tres baños, un millón de euros", de la que el sistema de extracción de la información puede extraer atributos como el número de dormitorios, el de baños, el coste y la zona. Esta información así extraída se puede usar para responder mejor a las consultas. Una organización que mantenga una base de datos de información sobre empresas puede usar un sistema de extracción de la información para extraer de manera automática la información de los artículos de los periódicos; la información extraída se relacionará con las modificaciones en los atributos que interesan, como las renuncias, los ceses o los nombramientos de los ejecutivos de las empresas. Se han creado varios sistemas para la extracción de información para aplicaciones especializadas. Usan técnicas lingüísticas y reglas definidas por los usuarios para dominios concretos (como los anuncios inmobiliarios).

19.8.2 Consulta de datos estructurados

Los datos estructurados se representan sobre todo en forma relacional o mediante XML. Se han creado varios sistemas para soportar la consulta de palabras clave en datos relacionales y de XML. Un problema frecuente en estos sistemas es la búsqueda de nodos (tuplas o elementos de XML) que contengan las palabras clave especificadas y la de caminos que los conecten (o ancestros comunes, en el caso de los datos de XML).

Por ejemplo, la consulta "Gómez Castellana" en una base de datos bancaria puede determinar que el nombre Gómez aparezca en una tupla de *cliente* y el nombre Castellana en una tupla de *sucursal*, y un camino que pase por la relación *impositor* que conecte esas dos tuplas. Estas consultas se pueden usar para la exploración y consulta ad hoc de los datos, cuando el usuario no conoce el esquema exacto y no desea realizar el esfuerzo de escribir una consulta de SQL o de XQuery para definir lo que está buscando. En realidad, no resulta razonable suponer que los usuarios legos escriban consultas en lenguajes de consultas estructurados, ya que la búsqueda mediante palabras clave resulta bastante natural.

Como las consultas no están completamente definidas, pueden tener muchos tipos diferentes de respuestas, que hay que clasificar. Se han propuesto varias técnicas para clasificar las respuestas en entornos de este tipo, de acuerdo con la longitud de los caminos de conexión y con base en técnicas para la asignación de direcciones y pesos a los bordes. También se han propuesto técnicas para la asignación de clasificaciones de popularidad a las tuplas y a los elementos de XML, de acuerdo con vínculos como

las claves externas y los vínculos IDREF. Véanse las notas bibliográficas para obtener más información sobre la búsqueda mediante palabras clave en los datos relacionales y de XML.

19.8.3 Respuesta a las preguntas

Los sistemas de recuperación de información se centran en la búsqueda de documentos importantes para cada consulta. Sin embargo, la respuesta a una consulta dada puede hallarse sólo en parte de un documento, o en partes pequeñas de varios documentos. Los sistemas de **respuesta a las preguntas** intentan ofrecer respuestas directas a las preguntas planteadas por los usuarios. Por ejemplo, una pregunta de la forma "¿Quién mató a Cánovas?" se responde mejor con una línea que diga "Antonio Cánovas del Castillo fue asesinado por el anarquista Angiolillo en 1897". Téngase en cuenta que la respuesta no contiene realmente las palabras "mató" ni "quién", pero el sistema deduce que "quién" puede responderse con un nombre, y "mató" está relacionado con "asesinó".

Los sistemas de respuesta a las preguntas que se centran en la información de Web suelen generar una o más consultas de palabras clave a partir de la pregunta formulada, formulan las consultas de palabras clave a los motores de búsqueda por Web y analizan los documentos devueltos para hallar fragmentos de esos documentos que respondan la pregunta. Se usan varias técnicas lingüísticas y heurísticas para generar las consultas de palabras clave y para hallar los fragmentos importantes de los documentos. El motor de búsqueda MSN de Microsoft soporta la respuesta a preguntas y usa la enciclopedia Encarta como fuente de información principal.

Los sistemas de respuesta a las preguntas de la generación actual tienen una potencia limitada, ya que no comprenden realmente ni la pregunta ni los documentos usados para responderla. No obstante, resultan útiles para cierto tipo de tareas de respuesta a preguntas sencillas.

19.9 Directorios

El usuario típico de una biblioteca puede usar un catálogo para hallar el libro que busca. Cuando recupera el libro del estante, no obstante, es probable que *hojee* otros libros que se hallen cerca. Las bibliotecas organizan los libros de modo que los títulos relacionados se guarden cerca unos de otros. Por tanto, puede que un libro que esté físicamente cerca del libro deseado también sea interesante, lo que hace que merezca la pena para los usuarios hojear esos libros.

Para guardar cerca unos de otros los libros relacionados las bibliotecas usan una **jerarquía de clasificación**. Los libros de ciencia se clasifican juntos. Dentro de este conjunto de libros hay una clasificación más detallada, que organiza por un lado los libros de informática, por otro los de matemáticas, etc. Dado que hay una relación entre las matemáticas y la informática, hay conjuntos importantes de libros que se guardan físicamente cerca. En otro nivel diferente de la jerarquía de la clasificación los libros de informática se dividen en subáreas, como los sistemas operativos, los lenguajes y los algoritmos. La Figura 19.1

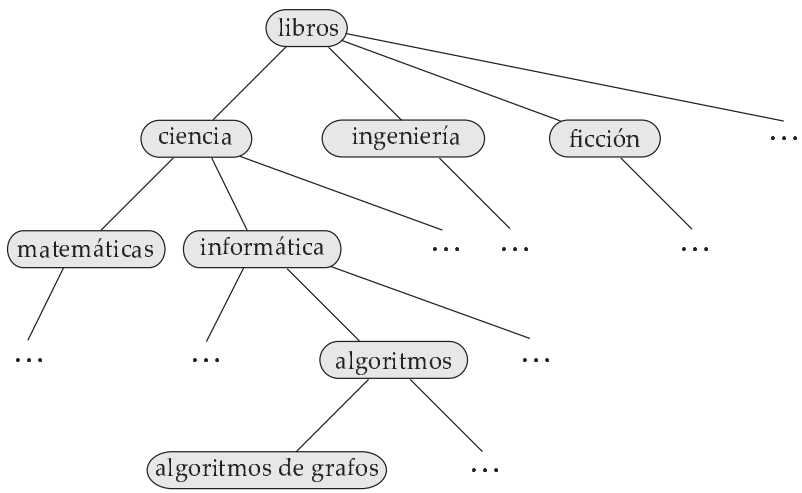


Figura 19.1 Jerarquía de clasificación para el sistema de una biblioteca.

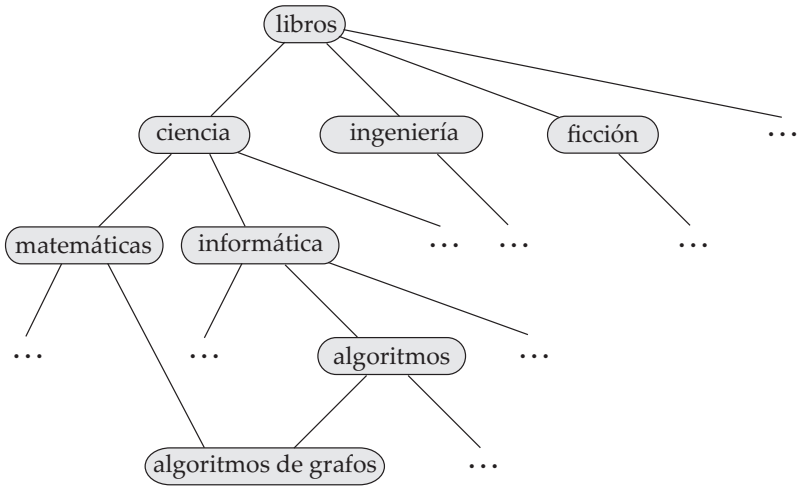


Figura 19.2 GAD de clasificación del sistema de recuperación de información de una biblioteca.

muestra una jerarquía de clasificación que pueden usar las bibliotecas. Como los libros sólo se pueden guardar en un sitio, cada libro de la biblioteca se clasifica exactamente en un punto de la jerarquía de clasificación.

En los sistemas de recuperación de información no hace falta almacenar cerca los documentos relacionados. No obstante, estos sistemas necesitan *organizar lógicamente los documentos* para permitir su exploración. Por tanto, estos sistemas pueden usar una jerarquía de clasificación parecida a la que usan las bibliotecas y, al mostrar un documento concreto, también pueden mostrar una breve descripción de los documentos que se hallan cercanos en la jerarquía.

En los sistemas de recuperación de información no hace falta guardar cada documento en un solo punto de la jerarquía. Los documentos que traten de matemáticas para informáticos pueden clasificarse en matemáticas y en informática. Lo que se guarda en cada punto es un identificador del documento (es decir, un puntero hacia el documento), y resulta fácil capturar el contenido del documento mediante su identificador.

Como consecuencia de esa flexibilidad, no sólo se puede clasificar cada documento en dos posiciones, sino que también puede aparecer una subárea de la jerarquía de la clasificación en dos áreas diferentes. La clase de documento "algoritmo de grafos" puede aparecer tanto en matemáticas como en informática. Por tanto, la jerarquía de clasificación es ahora un grafo acíclico dirigido (GAD), como puede verse en la Figura 19.2. Los documentos de algoritmos de grafos pueden aparecer en una sola posición del GAD, pero se puede llegar a ellos por varios caminos.

Un **directorio** no es más que una estructura de clasificación GAD. Cada hoja del directorio almacena vínculos con documentos del tema representado por la hoja. Los nodos internos también pueden contener vínculos, por ejemplo, con documentos que no se pueden clasificar en ninguno de los nodos hijo.

Para hallar información sobre un asunto los usuarios empiezan en la raíz del directorio y siguen los caminos por el GAD hasta alcanzar el nodo que representa el asunto deseado. Mientras avanzan por el directorio los usuarios no sólo pueden hallar documentos sobre el asunto en el que están interesados, sino que también pueden hallar documentos relacionados y clases relacionadas en la jerarquía de clasificación. Los usuarios pueden conseguir información nueva explorando los documentos (o las subclases) de las clases relacionadas.

La organización de la enorme cantidad de información disponible en Web en una estructura de directorio es una enorme tarea.

- El primer problema es la determinación de cuál debe ser exactamente la jerarquía del directorio.
- El segundo problema es, dado un documento, decidir los nodos del directorio que son categorías importantes para el documento.

Para afrontar el primer problema los portales como Yahoo tienen equipos de "bibliotecarios de Internet" que sugieren la jerarquía de la clasificación y la perfeccionan continuamente. El *proyecto de directorio abierto* (Open Directory Project) es un gran esfuerzo cooperativo con diferentes voluntarios que son responsables de organizar las diferentes ramas del directorio.

El segundo problema también puede afrontarse manualmente con bibliotecarios, o bien los conservadores del sitio Web pueden ser responsables de decidir el lugar de la jerarquía en que deben ubicarse sus sitios.

También hay técnicas para decidir de manera automática la ubicación de los documentos de acuerdo con el cálculo de su semejanza con documentos que ya se hayan clasificado.

19.10 Resumen

- Los sistemas de recuperación de información se usan para almacenar y consultar datos de texto como los documentos. Usan un modelo de datos más sencillo que el de los sistemas de bases de datos, pero ofrecen posibilidades de búsqueda más potentes dentro de ese modelo restringido.

  Las consultas intentan localizar los documentos de interés especificando, por ejemplo, conjuntos de palabras clave. La consulta que el usuario tiene en mente no se suele poder formular de manera precisa; por tanto, los sistemas de recuperación de información ordenan las respuestas con base en su posible relevancia.

- La clasificación por relevancia emplea varios tipos de información como:
  - Frecuencia de los términos: la relevancia de cada término para cada documento.
  - Frecuencia inversa de los documentos.
  - Clasificación por popularidad.

- La semejanza de los documentos se usa para recuperar documentos parecidos a un documento de ejemplo. La métrica del coseno se usa para definir la semejanza y se basa en el modelo del espacio vectorial.

- PageRank y la clasificación de nodos y autoridades son dos maneras de asignar prestigio a las páginas con base en los vínculos de cada página. La medida de PageRank se puede comprender de manera intuitiva usando un modelo de recorrido aleatorio. La información del texto ancla también se usa para calcular un concepto de popularidad por palabra clave.

- La contaminación de los motores de búsqueda intenta conseguir una clasificación elevada (no merecida).

- Los sinónimos y los homónimos complican la tarea de recuperación de información. Las consultas basadas en los conceptos intentan hallar los documentos que contienen los conceptos especificados, independientemente de las palabras exactas (o del idioma) en que se especifiquen. Las ontologías se usan para relacionar los conceptos entre sí, usando relaciones como *es* o *parte-de*.

- Los índices invertidos se usan para responder a las consultas de palabras clave.

- La precisión y la recuperación son dos medidas de la efectividad de los sistemas de recuperación de información.

- Los motores de búsqueda Web la recorren para hallar páginas, analizarlas para calcular las medidas de prestigio e indexarlas.

- Se han desarrollado técnicas para extraer información estructurada de los datos de texto, para llevar a cabo consultas basadas en palabras clave sobre datos estructurados y para dar respuestas directas a preguntas sencillas planteadas en lenguaje natural.

- Las estructuras de directorio se usan para clasificar los documentos con otros documentos semejantes.

Términos de repaso

- Sistemas de recuperación de información.
- Búsqueda por palabras clave.
- Recuperación de texto completo.
- Término.
- Clasificación por relevancia.
  - ☐ Frecuencia de los términos.
  - ☐ Frecuencia inversa de los documentos.
  - ☐ Relevancia.
  - ☐ Proximidad.
- Recuperación basada en la semejanza.
  - ☐ Modelo del espacio vectorial.
  - ☐ Métrica de semejanza del coseno.
  - ☐ Realimentación de la relevancia.
- Palabras de parada.
- Relevancia cuando se usan hipervínculos.
  - ☐ Popularidad y prestigio.
  - ☐ Transferencia del prestigio.
- PageRank.
  - ☐ Modelo de recorrido aleatorio.
- Relevancia basada en el texto ancla.
- Clasificación de nodos y autoridades.
- Contaminación de los índices de búsqueda.
- Sinónimos.
- Homónimos.
- Conceptos.
- Consultas basadas en conceptos.
- Ontologías.
- WordNet.
- Índice invertido.
- Falso rechazo.
- Falso negativo.
- Falso positivo.
- Precisión.
- Recuperación.
- Batidores Web.
- Extracción de información.
- Consulta de datos estructurados.
- Respuesta a preguntas.
- Directorios.
- Jerarquía de clasificación.

Ejercicios prácticos

19.1 Calcúlese la relevancia (mediante las definiciones correspondientes de la frecuencia de los términos y de la frecuencia inversa de los documentos) de cada una de los Ejercicios prácticos de este capítulo para la consulta "relación SQL".

19.2 Supóngase que se desea hallar documentos que contengan como mínimo c palabras clave de un conjunto dado de n. Supóngase también que se dispone de un índice de palabras clave que da una lista (ordenada) de identificadores de los documentos que contienen una palabra clave dada. Dese un algoritmo eficiente para hallar el conjunto de documentos deseado.

19.3 Sugiérase la manera de implementar la técnica iterativa para el cálculo de PageRank dado que la matriz T (incluso en la representación de lista de adyacentes) no cabe en la memoria.

19.4 Sugiérase la manera en que se puede indexar un documento que contiene una palabra (como "leopardo") de modo que las consultas que utilicen un concepto más general (como "carnívoro" o "mamífero") lo recuperen de manera eficiente. Se puede suponer que la jerarquía de conceptos no es muy profunda, por lo que cada concepto sólo tiene unas cuantas generalizaciones (cada concepto, no obstante, puede tener gran número de generalizaciones). También se puede suponer que se dispone de una función que devuelve el concepto de cada palabra del documento.

Sugiérase también la manera de que las consultas que utilicen conceptos especializados puedan recuperar conceptos más generales.

19.5 Supóngase que se mantienen en bloques listas invertidas, donde cada bloque denota la mejor clasificación de popularidad y las puntuaciones de TF-IDF de los documentos de los demás bloques. Sugiérase la manera en que la mezcla de las listas invertidas puede detenerse de manera temprana si el usuario sólo desea las primeras C respuestas.

Ejercicios

19.6 Usando una definición sencilla de la frecuencia de cada término como número de apariciones de cada término en el documento, dense las puntuaciones TF-IDF de cada término del conjunto de documentos que consiste en este ejercicio y en el siguiente.

19.7 Créese un pequeño ejemplo de cuatro documentos de pequeño tamaño, cada uno con su valor de PageRank, y créense cuatro listas invertidas de los documentos ordenados por PageRank. No hace falta calcular PageRank, basta con suponer un valor para cada página.

19.8 Supóngase que se desea llevar a cabo una consulta mediante palabras clave sobre un conjunto de tuplas de una base de datos, donde cada tupla tiene unos pocos atributos, cada uno de los cuales sólo contiene unas cuantas palabras. ¿El concepto de frecuencia de los términos tiene sentido en este contexto? ¿Y el de frecuencia inversa de los documentos? Explíquense las respuestas. Sugiérase también la manera en que se puede definir la semejanza de dos tuplas usando los conceptos de TF-IDF.

19.9 Los sitios Web que desean conseguir publicidad pueden unirse a un anillo de Webs, en los que crean vínculos con otros sitios del anillo, a cambio de que otros sitios del anillo creen vínculos con ellos. ¿Cuál es el efecto de estos anillos en las técnicas de clasificación por popularidad como PageRank?

19.10 El motor de búsqueda Google ofrece una característica por la cual los sitios Web pueden mostrar anuncios proporcionados por Google. Los anuncios proporcionados se basan en el contenido de cada página. Sugiérase la manera en que Google puede escoger los anuncios que debe proporcionar a cada página, dado el contenido de la página.

19.11 Una manera de crear una versión de PageRank específica para palabras clave es modificar el salto aleatorio de modo que sólo sean posibles los saltos a páginas que contengan la palabra clave deseada. Por tanto, las páginas que no contengan esa palabra clave pero estén cercanas (en términos de vínculos) a las que sí la contienen obtendrán también una clasificación no nula para esa palabra clave.

- **a.** Dense las ecuaciones que definen esta versión específica para palabras clave de PageRank.
- **b.** Dese una fórmula para calcular la relevancia de cada página para una consulta que contenga varias palabras clave.

19.12 La idea de clasificación por popularidad mediante hipervínculos se puede ampliar a los datos relacionales y de XML, usando las claves externas y los bordes IDREF en lugar de los hipervínculos. Sugiérase la manera de que este esquema de clasificación resulte útil para las aplicaciones siguientes:

- **a.** Una base de datos bibliográfica, que tiene vínculos de los artículos con sus autores y de cada artículo con los artículos a los que hace referencia.
- **b.** Una base de datos de ventas que tiene vínculos de cada registro de ventas con los productos que se han vendido.

Sugiérase también el motivo de que la clasificación por prestigio pueda dar resultados menos significativos en bases de datos de películas que registran los actores que trabajaron en cada película.

19.13 Explíquese la diferencia entre un falso positivo y un rechazo falso. Si es fundamental que las consultas de recuperación de información no pasen por alto ninguna información importante, explicar si es aceptable tener falsos positivos o falsos rechazos. Explicar el motivo.

Notas bibliográficas

Chakrabarti [2002], Grossman y Frieder [2004], Witten et al. [1999], y Baeza-Yates y Ribeiro-Neto [1999] ofrecen descripciones propias de libros de texto de la recuperación de información. Chakrabarti [2002] ofrece un tratamiento detallado de las batidas Web, de las técnicas de clasificación y de la agrupación y otras técnicas de obtención de datos relacionadas con la recuperación de información. El indexado

de los documentos se trata con detalle en Witten et al. [1999]. Jones y Willet [1997] es una colección de artículos sobre recuperación de información. Salton [1989] es uno de los primeros libros de texto sobre los sistemas de recuperación de información.

Brin y Page [1998] describen la anatomía del motor de búsqueda de Google, incluida la técnica PageRank, mientras que una técnica de clasificación basada en nodos y autoridades denominada HITS se describe en Kleinberg [1999]. Bharat y Henzinger [1998] presentan una mejora de la técnica de clasificación HITS. Estas técnicas, así como otras técnicas de clasificación basadas en la popularidad (y técnicas para evitar la contaminación de los motores de búsqueda) se describen con detalle en Chakrabarti [2002]. Chakrabarti et al. [1999] abordan el recorrido enfocado de Web para hallar páginas relacionadas con un asunto concreto. Chakrabarti [1999] ofrece un resumen del descubrimiento de recursos Web.

El sistema Citeseer (citeseer.ist.psu.edu) mantiene una base de datos de gran tamaño de publicaciones (artículos), con vínculos de citas entre las publicaciones, y usa estas citas para clasificar las publicaciones. Incluye una técnica para ajustar la clasificación por citas de acuerdo con la antigüedad de la publicación, para compensar el hecho de que las citas a una publicación dada aumentan a medida que pasa el tiempo; sin ese ajuste, los documentos más antiguos tienden a obtener clasificaciones más elevadas de las que merecen realmente.

La extracción de información y la respuesta a preguntas tienen una historia bastante larga en la comunidad de la inteligencia artificial. Jackson y Moulinier [2002] ofrecen un tratamiento de libro de texto de la técnica de procesamiento del lenguaje natural con énfasis en la extracción de la información. Soderland [1999] describe la extracción de la información mediante el sistema WHISK, mientras que Appelt y Israel [1999] ofrecen un tutorial sobre la extracción de información.

La Conferencia anual de Recuperacion de Texto (Text Retrieval Conference, TREC) tiene varios temas, incluida la recuperación de documentos, la respuesta a preguntas, la búsqueda genómica, etc. Cada tema define un problema y la infraestructura para comprobar la calidad de las soluciones a ese problema. Los detalles sobre TREC se pueden hallar en trec.nist.gov. La información sobre el tema de las respuestas a preguntas se puede hallar en trec.nist.gov/data/qa.html.

Se puede hallar más información sobre WordNet en wordnet.princeton.edu y en globalwordnet.org. El objetivo del sistema Cyc era la representación formal de grandes cantidades de conocimiento humano. Su base de conocimiento contiene gran número de términos, y de asertos sobre cada término. Cyc incluye también soporte para la comprensión del lenguaje natural y para la ruptura de la ambigüedad. La información sobre el sistema Cyc pueden hallarse en cyc.com y en opencyc.org.

Agrawal et al. [2002], Bhalotia et al. [2002] y Hristidis y Papakonstantinou [2002] tratan la consulta por palabras clave de los datos relacionales. La consulta por palabras clave de los datos de XML se aborda en Florescu et al. [2000] y Guo et al. [2003], entre otros.

Herramientas

Google (www.google.com) es actualmente el motor de búsqueda más popular, pero hay otros motores de búsqueda, como MSN Search (search.msn.com) y Yahoo Search (search.yahoo.com). El sitio searchenginewatch.com ofrece gran variedad de información sobre los motores de búsqueda. Yahoo (www.yahoo.com) y el Proyecto de Directorio Abierto (Open Directory Project, dmoz.org) ofrecen jerarquías de clasificación para los sitios Web.

PARTE 7

Arquitectura de sistemas

La arquitectura de los sistemas de bases de datos está enormemente influida por el sistema informático subyacente en el que se ejecuta el sistema de bases de datos. Los sistemas de bases de datos pueden ser centralizados, o cliente-servidor, donde una máquina que hace de servidor ejecuta trabajos de múltiples máquinas clientes. Los sistemas de bases de datos también pueden diseñarse para explotar las arquitecturas paralelas de computadoras. Las bases de datos distribuidas abarcan muchas máquinas separadas geográficamente.

En el Capítulo 20 se empieza tratando las arquitecturas de los sistemas de bases de datos que se ejecutan en sistemas servidores, los cuales se utilizan en arquitecturas centralizadas y cliente-servidor. En este capítulo se tratan los diferentes procesos que juntos implementan la funcionalidad de la base de datos. Después, se estudian las arquitecturas paralelas de computadoras y las arquitecturas paralelas de bases de datos diseñadas para diferentes tipos de computadoras paralelas. Finalmente, el capítulo trata asuntos arquitectónicos para la construcción de un sistema distribuido de bases de datos.

En el Capítulo 21 se describe la forma en que varias acciones de una base de datos, en particular el procesamiento de consultas, se pueden implementar para explotar el procesamiento paralelo.

En el Capítulo 22 se presentan varias cuestiones que surgen en una base de datos distribuida, y se describe cómo tratar cada cuestión. Estas cuestiones incluyen cómo almacenar datos, cómo asegurar la atomicidad de las transacciones que se ejecutan en varios emplazamientos, cómo realizar el control de concurrencia y cómo proporcionar alta disponibilidad ante la presencia de fallos. En este capítulo también se estudian el procesamiento distribuido de consultas y los sistemas de directorio.

CAPÍTULO 20

Arquitecturas de los sistemas de bases de datos

La arquitectura de un sistema de bases de datos está influida en gran medida por el sistema informático subyacente en el que se ejecuta, en concreto por aspectos de la arquitectura de la computadora como la conexión en red, el paralelismo y la distribución:

- La conexión en red de varias computadoras permite que algunas tareas se ejecuten en un sistema servidor y que otras se ejecuten en los sistemas clientes. Esta división de trabajo ha conducido al desarrollo de *sistemas de bases de datos cliente–servidor*.
- El procesamiento paralelo dentro de una computadora permite acelerar las actividades del sistema de base de datos, proporcionando a las transacciones unas respuestas más rápidas, así como la capacidad de ejecutar más transacciones por segundo. Las consultas pueden procesarse de manera que se explote el paralelismo ofrecido por el sistema informático subyacente. La necesidad del procesamiento paralelo de consultas ha conducido al desarrollo de los *sistemas de bases de datos paralelos*.
- La distribución de datos a través de las distintas sedes de una organización permite que estos datos residan donde han sido generados o donde son más necesarios, pero continuar siendo accesibles desde otros lugares o departamentos diferentes. El hecho de guardar varias copias de la base de datos en diferentes sitios permite que puedan continuar las operaciones sobre la base de datos aunque algún sitio se vea afectado por algún desastre natural como una inundación, un incendio o un terremoto. Se han desarrollado los *sistemas distribuidos de bases de datos* para manejar datos distribuidos geográfica o administrativamente a lo largo de múltiples sistemas de bases de datos.

En este capítulo se estudia la arquitectura de los sistemas de bases de datos comenzando con los tradicionales sistemas centralizados y tratando, más adelante, los sistemas de bases de datos cliente–servidor, paralelos y distribuidos.

20.1 Arquitecturas centralizadas y cliente–servidor

Los sistemas de bases de datos centralizados son aquellos que se ejecutan en un único sistema informático sin interaccionar con ninguna otra computadora. Tales sistemas comprenden el rango desde los sistemas de bases de datos monousuario ejecutándose en computadoras personales hasta los sistemas de bases de datos de alto rendimiento ejecutándose en grandes sistemas. Por otro lado, los sistemas cliente–servidor tienen su funcionalidad dividida entre el sistema servidor y múltiples sistemas clientes.

20.1.1 Sistemas centralizados

Una computadora moderna de propósito general consiste en una o unas pocas unidades centrales de procesamiento y un número determinado de controladores para los dispositivos que se encuentran conectados a través de un bus común, el cual proporciona acceso a la memoria compartida (Figura 20.1). Las CPU (unidades centrales de procesamiento) poseen memorias caché locales donde se almacenan copias de ciertas partes de la memoria para acelerar el acceso a los datos. Cada controlador de dispositivo se encarga de un tipo específico de dispositivos (por ejemplo, una unidad de disco, una tarjeta de sonido o un monitor). Las CPU y los controladores de dispositivos pueden ejecutarse concurrentemente compitiendo así por el acceso a la memoria. La memoria caché reduce la disputa por el acceso a la memoria ya que la CPU necesita acceder a la memoria compartida un número de veces menor.

Se distinguen dos formas de utilizar las computadoras: como sistemas monousuario o multiusuario. En la primera categoría se encuentran las computadoras personales y las estaciones de trabajo. Un **sistema monousuario** típico es una unidad de sobremesa utilizada por una única persona que dispone de una sola CPU, de uno o dos discos fijos y que trabaja con un sistema operativo que sólo permite un único usuario. Por el contrario, un **sistema multiusuario** típico tiene más discos y más memoria, puede disponer de varias CPU y trabaja con un sistema operativo multiusuario. Se encarga de dar servicio a un gran número de usuarios que están conectados al sistema a través de terminales.

Normalmente, los sistemas de bases de datos diseñados para funcionar sobre sistemas monousuario no suelen proporcionar muchas de las facilidades que ofrecen los sistemas multiusuario. En particular no tienen control de concurrencia, el cual no es necesario cuando solamente un usuario puede generar modificaciones. Las facilidades de recuperación en estos sistemas o no existen o son primitivas; por ejemplo, realizar una copia de seguridad de la base de datos antes de cualquier modificación. La mayoría de estos sistemas no admiten SQL y proporcionan un lenguaje de consulta muy simple que, en algunos casos, es una variante de QBE. En cambio, los sistemas de bases de datos diseñados para sistemas multiusuario soportan todas las características de las transacciones que se han estudiado antes.

Aunque hoy en día las computadoras de propósito general disponen de varios procesadores, utilizan **paralelismo de grano grueso**, disponiendo de unos pocos procesadores (normalmente dos o cuatro) que comparten la misma memoria principal. Las bases de datos que se ejecutan en tales máquinas habitualmente no intentan dividir una consulta simple entre los distintos procesadores, sino que ejecutan cada consulta en un único procesador posibilitando la concurrencia de varias consultas. Así, estos sistemas soportan una mayor productividad, es decir, permiten ejecutar un mayor número de transacciones por segundo, a pesar de que cada transacción individualmente no se ejecute más rápido.

Las bases de datos diseñadas para las máquinas monoprocesador ya disponen de multitarea permitiendo que varios procesos se ejecuten a la vez en el mismo procesador, usando tiempo compartido, mientras que de cara al usuario parece que los procesos se están ejecutando en paralelo. De esta manera,

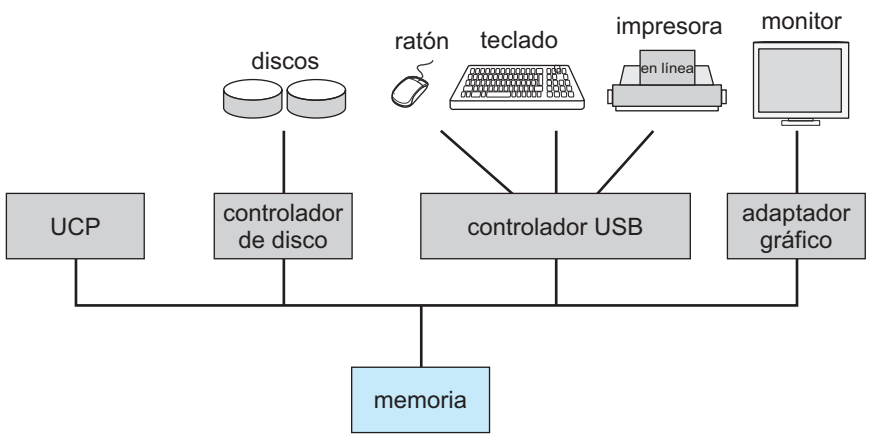


Figura 20.1 Un sistema centralizado.

desde un punto de vista lógico, las máquinas paralelas de grano grueso parecen ser idénticas a las máquinas monoprocesador, y pueden adaptarse fácilmente los sistemas de bases de datos diseñados para máquinas de tiempo compartido para que puedan ejecutarse sobre máquinas paralelas de grano grueso.

Por el contrario, las **máquinas paralelas de grano fino** disponen de un gran número de procesadores y los sistemas de bases de datos que se ejecutan sobre ellas intentan hacer paralelas las tareas simples (consultas, por ejemplo) que solicitan los usuarios. En el Apartado 20.3 se estudia la arquitectura de los sistemas de bases de datos paralelos.

20.1.2 Sistemas cliente–servidor

Como las computadoras personales son cada vez más rápidas, más potentes y más baratas, los sistemas se han ido distanciando de la arquitectura centralizada. Los terminales conectados a un sistema central han sido suplantados por computadoras personales. De igual forma, la interfaz de usuario, que solía estar gestionada directamente por el sistema central, está pasando a ser gestionada, cada vez más, por las computadoras personales. Como consecuencia, los sistemas centralizados actúan hoy como **sistemas servidores** que satisfacen las peticiones generadas por los *sistemas clientes*. En la Figura 20.2 se representa la estructura general de un sistema cliente–servidor.

La funcionalidad proporcionada por los sistemas de bases de datos se puede dividir a grandes rasgos en dos partes: la fachada y el sistema subyacente. El sistema subyacente gestiona el acceso a las estructuras, la evaluación y optimización de consultas, el control de concurrencia y la recuperación. La fachada de un sistema de base de datos está formado por herramientas como la interfaz de usuario con SQL, interfaces de formularios, diseñadores de informes y herramientas para la recopilación y análisis de datos. La interfaz entre la fachada y el sistema subyacente puede ser SQL o una aplicación.

Las normas como *ODBC* y *JDBC*, que se estudiaron en el Capítulo 3, se desarrollaron para hacer de interfaz entre clientes y servidores. Cualquier cliente que utilice interfaces ODBC o JDBC puede conectarse a cualquier servidor que proporcione esta interfaz.

Ciertas aplicaciones como las hojas de cálculo y los paquetes de análisis estadístico utilizan la interfaz cliente–servidor directamente para acceder a los datos del servidor subyacente. De hecho, proporcionan interfaces visibles especiales para diferentes tareas.

Los sistemas que trabajan con una gran cantidad de usuarios adoptan una arquitectura de tres capas, que se vio anteriormente en la Figura 1.7 (Capítulo 1), donde la fachada es el explorador Web que se comunica con un servidor de aplicaciones. El servidor de aplicaciones actúa como cliente del servidor de bases de datos.

Algunos sistemas de procesamiento de transacciones proporcionan una interfaz de **llamada a procedimientos remotos para transacciones** para conectar los clientes con el servidor. Estas llamadas aparecen para el programador como llamadas normales a procedimientos, pero todas las llamadas a procedimientos remotos hechas desde un cliente se engloban en una única transacción al servidor final. De este modo, si la transacción se cancela, el servidor puede deshacer los efectos de las llamadas a procedimientos remotos individuales.

20.2 Arquitecturas de sistemas servidores

Los sistemas servidores pueden dividirse en servidores de transacciones y servidores de datos.

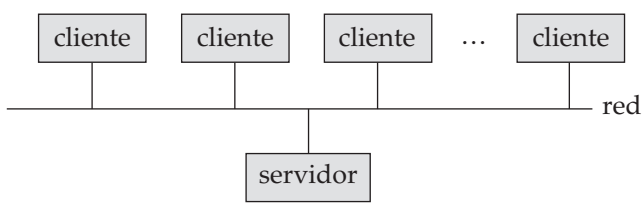


Figura 20.2 Estructura general de un sistema cliente–servidor.

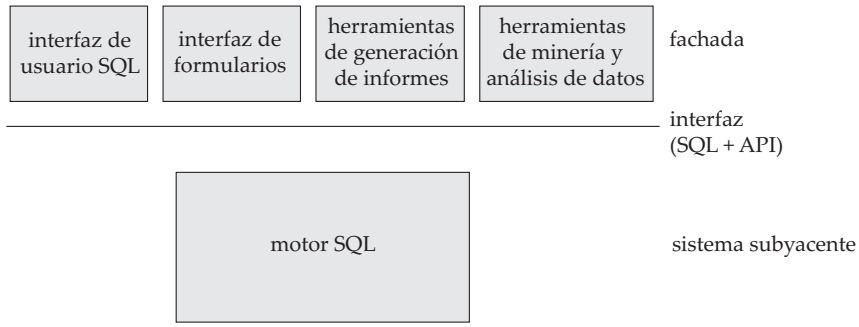


Figura 20.3 Funcionalidades de la fachada y del sistema subyacente.

- Los sistemas **servidores de transacciones**, también llamados sistemas **servidores de consultas**, proporcionan una interfaz a través de la cual los clientes pueden enviar peticiones para realizar una acción que el servidor ejecutará y cuyos resultados se devolverán al cliente. Normalmente, las máquinas cliente envían las transacciones a los sistemas servidores, lugar en el que estas transacciones se ejecutan, y los resultados se devuelven a los clientes que son los encargados de visualizar los datos. Las peticiones se pueden especificar utilizando SQL o mediante la interfaz de una aplicación especializada.
- Los sistemas **servidores de datos** permiten a los clientes interaccionar con los servidores realizando peticiones de lectura o modificación de datos en unidades tales como archivos o páginas. Por ejemplo, los servidores de archivos proporcionan una interfaz de sistema de archivos a través de la cual los clientes pueden crear, modificar, leer y borrar archivos. Los servidores de datos de los sistemas de bases de datos ofrecen muchas más funcionalidades; soportan unidades de datos de menor tamaño que los archivos—como páginas, tuplas u objetos. Proporcionan facilidades de indexación de los datos así como facilidades de transacción de modo que los datos nunca se quedan en un estado inconsistente si falla una máquina cliente o un proceso.

De éstas, la arquitectura del servidor de transacciones es, con mucho, la arquitectura más ampliamente utilizada. En los Apartados 20.2.1 y 20.2.2 se desarrollarán las arquitecturas de los servidores de transacciones y de los servidores de datos.

20.2.1 Estructura de procesos del servidor de transacciones

Hoy en día, un sistema servidor de transacciones típico consiste en múltiples procesos accediendo a los datos en una memoria compartida, como en la Figura 20.4. Los procesos que forman parte del sistema de bases de datos incluyen:

- **Proceso servidor**. Son procesos que reciben consultas del usuario (transacciones), las ejecutan, y devuelven los resultados. Las consultas deben enviarse a los procesos servidor desde la interfaz de usuario, o desde un proceso de usuario que ejecuta SQL incorporado, o a través de JDBC, ODBC o protocolos similares. Algunos sistemas de bases de datos utilizan un proceso distinto para cada sesión de usuario, y otros utilizan un único proceso de la base de datos para todas las sesiones del usuario, pero con múltiples hebras de forma que se pueden ejecutar concurrentemente múltiples consultas (una **hebra** es parecida a un proceso; sin embargo, varias hebras se pueden ejecutar concurrentemente como parte de un mismo proceso, y todas ellas en el mismo espacio de memoria virtual). Algunos sistemas de bases de datos utilizan una arquitectura híbrida, con procesos múltiples, cada uno de ellos con varias hebras.
- **Proceso gestor de bloqueos**. Este proceso implementa una función de gestión de bloqueos que incluye concesión de bloqueos, liberación de bloqueos y detección de interbloqueos.
- **Proceso escritor de bases de datos**. Existe uno o más procesos que vuelcan al disco los bloques de memoria intermedia modificados de forma continua.

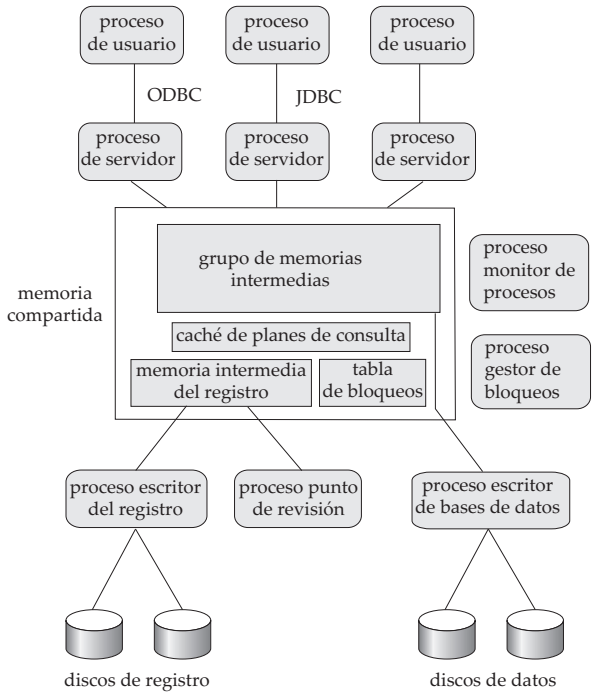


Figura 20.4 Estructura de la memoria compartida y de los procesos.

- **Proceso escritor del registro**. Este proceso genera entradas del registro en el almacenamiento estable a partir de la memoria intermedia del registro. Los procesos servidor simplifican la adición de entradas a la memoria intermedia del registro en memoria compartida y, si es necesario forzar la escritura del registro, le piden al proceso escritor del registro que vuelque las entradas del registro.
- **Proceso punto de revisión**. Este proceso realiza periódicamente puntos de revisión.
- **Proceso monitor de procesos**. Este proceso observa otros procesos y, si cualquiera de ellos falla, realiza acciones de recuperación para el proceso, tales como cancelar cualquier transacción que estuviera ejecutando el proceso fallido, y reinicia el proceso.

La memoria compartida contiene todos los datos compartidos, como:

- Grupo de memorias intermedias.
- Tabla de bloqueos.
- Memoria intermedia del registro, que contiene las entradas del registro que esperan a ser volcadas en el almacenamiento estable.
- Planes de consulta en caché, que se pueden reutilizar si se envía de nuevo la misma consulta.

Todos los procesos de la base de datos pueden acceder a los datos de la memoria compartida. Ya que múltiples procesos pueden leer o realizar actualizaciones en las estructuras de datos en memoria compartida, debe haber un mecanismo que asegure que sólo uno de ellos está modificando una estructura de datos en un momento dado, y que ningún proceso está leyendo una estructura de datos mientras otros la escriben. Tal **exclusión mutua** se puede implementar por medio de funciones del sistema operativo llamadas semáforos. Implementaciones alternativas, con menos sobrecargas, utilizan **instrucciones atómicas** especiales soportadas por el hardware de la computadora; un tipo de instrucción atómica comprueba una posición de la memoria y la establece a uno automáticamente. Se pueden encontrar más detalles sobre la excusión mutua en cualquier libro de texto de un sistema operativo estándar. Los mecanismos de exclusión mutua también se utilizan para implementar pestillos.

Para evitar la sobrecarga del paso de mensajes, en muchos sistemas de bases de datos los procesos servidor implementan el bloqueo actualizando directamente la tabla de bloqueos (que está en memoria compartida), en lugar de enviar mensajes de solicitud de bloqueo a un proceso administrador de bloqueos. El procedimiento de solicitud de bloqueos ejecuta las acciones que realizaría el proceso administrador de bloqueos para procesar una solicitud de bloqueo. Las acciones de la solicitud y la liberación de bloqueos son como las del Apartado 16.1.4, pero con dos diferencias significativas:

- Dado que varios procesos servidor pueden acceder a la memoria compartida, se asegurará la exclusión mutua en la tabla de bloqueos.
- Si no se puede obtener un bloqueo inmediatamente a causa de un conflicto de bloqueos, el código de la solicitud de bloqueo sigue observando la tabla de bloqueos hasta percatarse de que se ha concedido el bloqueo. El código de liberación de bloqueo actualiza la tabla de bloqueos para indicar a qué proceso se le ha concedido el bloqueo.

  Para evitar repetidas comprobaciones de la tabla de bloqueos, el código de solicitud de bloqueo puede utilizar los semáforos del sistema operativo para esperar una notificación de una concesión de bloqueo. El código de liberación de bloqueo debe utilizar entonces el mecanismo de semáforos para notificar a las transacciones que están esperando que sus bloqueos hayan sido concedidos.

Incluso si el sistema gestiona las solicitudes de bloqueo por medio de memoria compartida, aún utiliza el proceso administrador de bloqueos para la detección de interbloqueos.

20.2.2 Servidores de datos

Los sistemas servidores de datos se utilizan en redes de área local en las cuales se alcanza una alta velocidad de conexión entre los clientes y el servidor. Las máquinas clientes son comparables al servidor en cuanto a poder de procesamiento y ejecutan tareas de cómputo intensivo. En este entorno tiene sentido enviar los datos a las máquinas clientes, realizar allí todo el procesamiento (que puede durar un tiempo) y después enviar los datos de vuelta al servidor. Obsérvese que esta arquitectura necesita que los clientes posean todas las funcionalidades del sistema subyacente. Las arquitecturas de los servidores de datos se han hecho particularmente populares en los sistemas de bases de datos orientadas a objetos.

En esta arquitectura surgen algunos aspectos interesantes, ya que el coste en tiempo de comunicación entre el cliente y el servidor es alto comparado al de acceso a una memoria local (milisegundos frente a menos de 100 nanosegundos).

- **Envío de páginas o envío de elementos.** La unidad de comunicación de datos puede ser de grano grueso, como una página, o de grano fino, como una tupla (o, en el contexto de los sistemas de bases de datos orientados a objetos, un objeto). Se empleará el término **elemento** para referirse tanto a tuplas como a objetos.

  Si la unidad de comunicación de datos es un único elemento, la sobrecarga por la transferencia de mensajes es alta comparada con el número de datos transmitidos. En lugar de ello, cuando se necesita un elemento, cobra sentido la idea de enviar junto a él otros que probablemente vayan a emplearse en un futuro próximo. Se denomina **preextracción** a la acción de buscar y enviar elementos antes de que sea estrictamente necesario. Si varios elementos residen en un página, el envío de páginas puede considerarse como una forma de preextracción ya que, cuando un proceso desee acceder a un único elemento de la página, se enviarán todos los elementos de esa página.
- **Bloqueo.** La concesión del bloqueo de los elementos de datos que el servidor envía a los clientes la realiza habitualmente el propio servidor. Un inconveniente del envío de páginas es que los clientes pueden recibir bloqueos de grano grueso—el bloqueo de una página bloquea implícitamente todos los elementos que residen en ella. El cliente adquiere implícitamente bloqueos sobre todos los elementos preextraídos incluso aunque no esté accediendo a algunos de ellos. De esta forma, es posible que se detenega innecesariamente el procesamiento de otros clientes que necesiten bloquear estos elementos. Se han propuesto algunas técnicas para la **liberación de**

bloqueos en las que el servidor puede pedir a los clientes que le devuelvan el control sobre los bloqueos de los elementos preextraídos. Si el cliente no necesita el elemento preextraído puede devolver los bloqueos sobre ese elemento al servidor para que éstos puedan ser asignados a otros clientes.

- **Caché de datos**. Los datos que se envían al cliente en favor de una transacción se pueden **alojar en una caché** del cliente incluso una vez completada la transacción, si dispone de suficiente espacio de almacenamiento libre. Las transacciones sucesivas en el mismo cliente pueden hacer uso de los datos en caché. Sin embargo, se presenta el problema de la **coherencia de caché**: si una transacción encuentra los datos en la caché, debe asegurarse de que esos datos están al día ya que, después de haber sido almacenados en la caché, pueden haber sido modificados por otro cliente. Así, debe establecerse una comunicación con el servidor para comprobar la validez de los datos y poder adquirir un bloqueo sobre ellos.
- **Caché de bloqueos**. Los bloqueos también pueden ser almacenados en la memoria caché del cliente si los datos están prácticamente divididos entre los clientes, de manera que un cliente rara vez necesite los datos de otros clientes. Supóngase que se encuentran en la memoria caché tanto el elemento de datos que se busca como el bloqueo requerido para acceder al mismo. Entonces, el cliente puede acceder al elemento de datos sin necesidad de comunicar nada al servidor. No obstante, el servidor debe seguir el rastro de los bloqueos en caché; si un cliente solicita un bloqueo al servidor, éste debe **comunicar** a todos los bloqueos sobre el elemento de datos que se encuentren en las memorias caché de otros clientes. La tarea se vuelve más complicada cuando se tienen en cuenta los posibles fallos de la máquina. Esta técnica se diferencia de la liberación de bloqueos en que la caché de bloqueo se realiza a través de transacciones; de otra forma, las dos técnicas serían similares.

Las referencias bibliográficas proporcionan más información sobre los sistemas cliente–servidor de bases de datos.

20.3 Sistemas paralelos

Los sistemas paralelos mejoran la velocidad de procesamiento y de E/S porque la CPU y los discos funcionan en paralelo. Cada vez son más comunes las máquinas paralelas, lo que hace que cada vez sea más importante el estudio de los sistemas paralelos de bases de datos. La fuerza que ha impulsado a los sistemas paralelos de bases de datos ha sido la demanda de aplicaciones que han de manejar bases de datos extremadamente grandes (del orden de terabytes—esto es, 1012 bytes) o que tienen que procesar un número enorme de transacciones por segundo (del orden de miles de transacciones por segundo). Los sistemas de bases de datos centralizados o cliente–servidor no son suficientemente potentes para soportar tales aplicaciones.

En el procesamiento paralelo se realizan muchas operaciones simultáneamente mientras que en el procesamiento secuencial, los distintos pasos computacionales han de ejecutarse en serie. Una máquina paralela de **grano grueso** consiste en un pequeño número de potentes procesadores; una máquina **masivamente paralela** o de **grano fino** utiliza miles de procesadores más pequeños. Hoy en día, la mayoría de las máquinas de gama alta ofrecen un cierto grado de paralelismo de grano grueso: son comunes las máquinas con dos o cuatro procesadores. Las computadoras masivamente paralelas se distinguen de las máquinas paralelas de grano grueso porque son capaces de soportar un grado de paralelismo mucho mayor. Ya se encuentran en el mercado computadoras paralelas con cientos de CPU y discos.

Para evaluar el rendimiento de los sistemas de bases de datos existen dos medidas principales: (1) la **productividad**, número de tareas que pueden completarse en un intervalo de tiempo determinado, y (2) el **tiempo de respuesta**, cantidad de tiempo que necesita para completar una única tarea a partir del momento en que se envíe. Un sistema que procese un gran número de pequeñas transacciones puede mejorar la productividad realizando muchas transacciones en paralelo. Un sistema que procese transacciones largas puede mejorar el tiempo de respuesta y la productividad realizando en paralelo las distintas subtareas de cada transacción.

20.3.1 Ganancia de velocidad y ampliabilidad

La ganancia de velocidad y la ampliabilidad son dos aspectos importantes en el estudio del paralelismo. La **ganancia de velocidad** se refiere a la ejecución en menos tiempo de una tarea dada mediante el incremento del grado de paralelismo. La **ampliabilidad** se refiere al manejo de transacciones más largas mediante el incremento del grado de paralelismo.

Considérese un sistema paralelo con un cierto número de procesadores y discos que está ejecutando una aplicación de base de datos. Supóngase ahora que se incrementa el tamaño del sistema añadiéndole más procesadores, discos y otros componentes. El objetivo es realizar el procesamiento de la tarea en un tiempo inversamente proporcional al número de procesadores y discos del sistema. Supóngase que el tiempo de ejecución de una tarea en la máquina más grande es T_G y que el tiempo de ejecución de la misma tarea en la máquina más pequeña es T_P. La ganancia de velocidad debida al paralelismo se define como T_P/T_G. Se dice que un sistema paralelo tiene una **ganancia de velocidad lineal** si la ganancia de velocidad es N cuando el sistema más grande tiene N veces más recursos (CPU, discos, etc.) que el sistema más pequeño. Si la ganancia de velocidad es menor que N se dice que el sistema tiene una **ganancia de velocidad sublineal**. En la Figura 20.5 se muestra la ganancia de velocidad lineal y sublineal.

La ampliabilidad está relacionada con la capacidad para procesar tareas más largas en el mismo tiempo mediante el incremento de los recursos del sistema. Sea Q una tarea y sea Q_N una tarea N veces más grande que Q. Supóngase que T_P es el tiempo de ejecución de la tarea Q en una máquina dada M_P y que T_G es el tiempo de ejecución de la tarea Q_N en una máquina paralela *MG*, la cual es N veces más grande que M_P. La ampliabilidad se define como T_P/T_G. Se dice que el sistema paralelo M_G tiene una **ampliabilidad lineal** sobre la tarea Q si $T_G = T_P$. Si $T_G > T_P$ se dice que el sistema tiene una **ampliabilidad sublineal**. En la Figura 20.6 se muestra la ampliabilidad lineal y sublineal (donde los recursos aumentan proporcionalmente al tamaño del problema). La manera de medir el tamaño de las tareas da lugar a dos tipos de ampliabilidad relevantes en los sistemas paralelos de bases de datos:

- En la **ampliabilidad por lotes** aumenta el tamaño de la base de datos, y las tareas son trabajos más largos cuyos tiempos de ejecución dependen del tamaño de la base de datos. Recorrer una relación cuyo tamaño es proporcional al tamaño de la base de datos sería un ejemplo de tales tareas. Así, la medida del tamaño del problema es el tamaño de la base de datos. La ampliabilidad por lotes también se utiliza en aplicaciones científicas tales como la ejecución de una consulta con una resolución N veces mayor o la realización de una simulación N veces más larga.
- En la **ampliabilidad de transacciones** aumenta la velocidad con la que se envían las transacciones a la base de datos y el tamaño de la base de datos crece proporcionalmente a la tasa de transacciones. Este tipo de ampliabilidad es el relevante en los sistemas de procesamiento de transacciones en los que las transacciones son modificaciones pequeñas—por ejemplo, un abono o retirada de fondos de una cuenta—y cuantas más cuentas se creen, más crece la tasa de transacciones. Este procesamiento de transacciones se adapta especialmente bien a la ejecución en paralelo, ya que

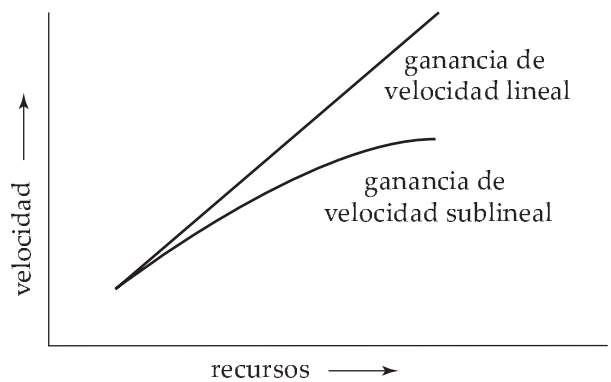


Figura 20.5 Ganancia de velocidad respecto al incremento de los recursos.

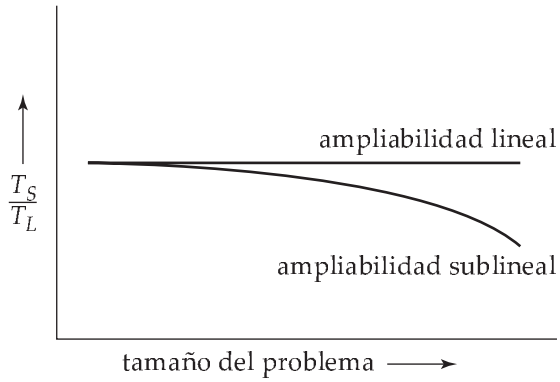


Figura 20.6 Ampliabilidad respecto al crecimiento del tamaño del problema y de los recursos.

las transacciones pueden ejecutarse concurrente e independientemente en procesadores distintos y cada transacción dura más o menos el mismo tiempo, aunque crezca la base de datos.

La ampliabilidad es normalmente el factor más importante para medir la eficiencia de un sistema paralelo de bases de datos. El objetivo del paralelismo en los sistemas de bases de datos suele ser asegurar que la ejecución del sistema continuará realizándose a una velocidad aceptable, incluso en el caso de que aumente el tamaño de la base de datos o el número de transacciones. El incremento de la capacidad del sistema mediante el incremento del paralelismo proporciona a una empresa un modo de crecimiento más suave que el de reemplazar un sistema centralizado por una máquina más rápida (suponiendo incluso que esta máquina existiera). Sin embargo, cuando se utiliza la ampliabilidad debe atenderse también a los valores del rendimiento absoluto; una máquina con un ampliabilidad lineal puede tener un rendimiento más bajo que otra con ampliabilidad sublineal simplemente porque la última sea mucho más rápida que la primera.

Existen algunos factores que trabajan en contra de la eficiencia del paralelismo y pueden atenuar tanto la ganancia de velocidad como la ampliabilidad.

- **Costes de inicio**. El inicio de un único proceso lleva asociado un coste. En una operación paralela compuesta por miles de procesos el *tiempo de inicio* puede llegar a ser mucho mayor que el tiempo real de procesamiento, lo que influye negativamente en la ganancia de velocidad.
- **Interferencia**. Como los procesos que se ejecutan en un sistema paralelo acceden con frecuencia a recursos compartidos, pueden sufrir un cierto retardo como consecuencia de la *interferencia* de cada nuevo proceso en la competencia con los procesos existentes por el acceso a los recursos más comunes, como el bus del sistema, los discos compartidos o incluso los bloqueos. Este fenómeno afecta tanto a la ganancia de velocidad como a la ampliabilidad.
- **Sesgo**. Al dividir cada tarea en un cierto número de pasos paralelos se reduce el tamaño del paso medio. Es más, el tiempo de servicio de la tarea completa vendrá determinado por el tiempo de servicio del paso más lento. Normalmente es difícil dividir una tarea en partes exactamente iguales, entonces se dice que la forma de distribución de los tamaños es *sesgada*. Por ejemplo, si se divide una tarea de tamaño 100 en 10 partes y la división está sesgada, puede haber algunas tareas de tamaño menor que 10 y otras de tamaño superior a 10; si el tamaño de una tarea fuera 20, la ganancia de velocidad que se obtendría al ejecutar las tareas en paralelo sólo valdría 5 en vez de lo que cabría esperarse, 10.

20.3.2 Redes de interconexión

Los sistemas paralelos están constituidos por un conjunto de componentes (procesadores, memoria y discos) que pueden comunicarse entre sí a través de una **red de interconexión**. La Figura 20.7 muestra tres tipos de redes de interconexión utilizados frecuentemente:

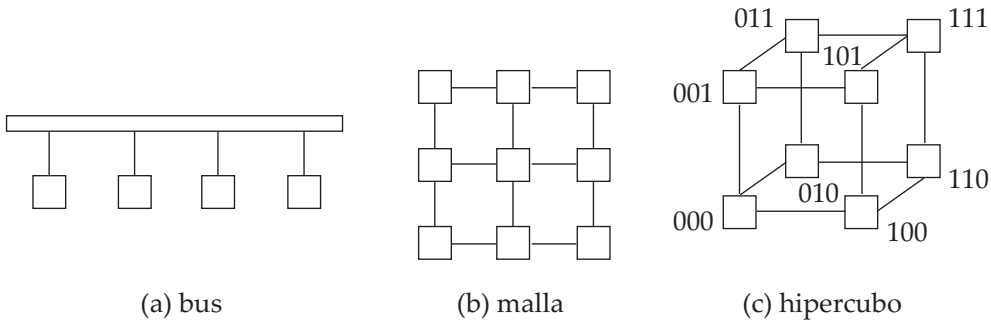


Figura 20.7 Redes de interconexión.

- **Bus**. Todos los componentes del sistema pueden enviar o recibir datos de un único bus de comunicaciones. Este tipo de interconexión se muestra en la Figura 20.7a. El bus puede ser una red Ethernet o una interconexión paralela. Las arquitecturas de bus trabajan bien para un pequeño número de procesadores. Sin embargo, como el bus sólo puede gestionar la comunicación de un único componente en cada momento, las arquitecturas de bus son menos apropiadas según aumenta el paralelismo.
- **Malla**. Los componentes se organizan como los nodos de una retícula de modo que cada componente está conectado con todos los nodos adyacentes. En una malla bidimensional cada nodo está conectado con cuatro nodos adyacentes, mientras que una malla tridimensional cada nodo está conectado con seis nodos adyacentes. La Figura 20.7b muestra una malla bidimensional. Los nodos entre los que no existe una conexión directa pueden comunicarse mediante el envío de mensajes a través de una secuencia de nodos intermedios que sí dispongan de conexión directa. A medida que aumenta el número de componentes también aumenta el número de enlaces de comunicación por lo que la capacidad de comunicación de una malla es mejor cuanto mayor es el paralelismo.
- **Hipercubo**. Se asigna a cada componente un número binario de modo que dos componentes tienen una conexión directa si sus correspondientes representaciones binarias difieren en un sólo bit. Así, cada uno de los n componentes está conectado con otros $\log(n)$ componentes. La Figura 20.7c muestra un hipercubo con 8 vértices. Puede demostrarse que, en un hipercubo, un mensaje de un componente puede llegar a cualquier otro componente de la red de interconexión atravesando a lo sumo $\log(n)$ enlaces. Por el contrario, en una malla un componente puede estar a $2(\sqrt{n}-1)$ enlaces de otros componentes (o a $\sqrt{n}$ enlaces de distancia si la malla de interconexión conecta entre sí los bordes opuestos). De esta manera, el retardo de la comunicación en un hipercubo es significativamente menor que en una malla.

20.3.3 Arquitecturas paralelas de bases de datos

Existen varios modelos de arquitecturas para las máquinas paralelas. En la Figura 20.8 se muestran algunos de los más importantes (en la figura, M quiere decir memoria, P procesador y los discos se dibujan como cilindros):

- **Memoria compartida**. Todos los procesadores comparten una memoria común (Figura 20.8a).
- **Disco compartido**. Todos los procesadores comparten un conjunto de discos común (Figura 20.8b). Algunas veces los sistemas de disco compartido se denominan **agrupaciones**.
- **Sin compartimiento**. Los procesadores no comparten ni memoria ni disco (Figura 20.8c).
- **Jerárquica**. Este modelo es un híbrido de las arquitecturas anteriores (Figura 20.8d).

En los Apartados 20.3.3.1 hasta el 20.3.3.4 se abordará cada uno de estos modelos.

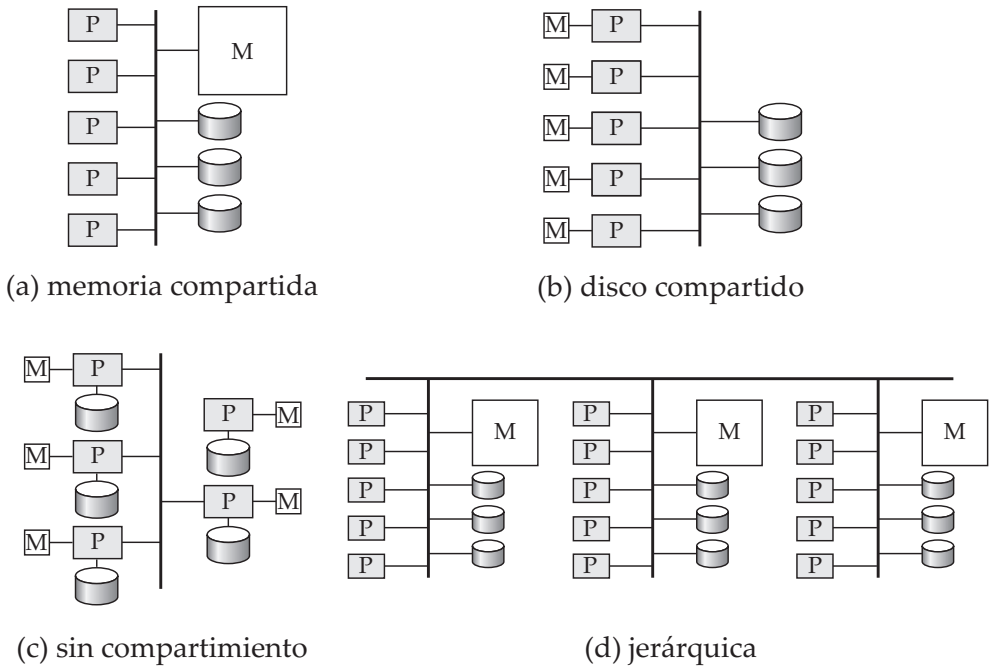


Figura 20.8 Arquitecturas paralelas de bases de datos.

Las técnicas utilizadas para acelerar el procesamiento de transacciones en sistemas servidores de datos, como la caché de datos y bloqueos y la liberación de bloqueos, tratadas en el Apartado 20.2.2, también se pueden utilizar en bases de datos paralelas de discos compartidos además de en bases de datos paralelas sin compartimiento. De hecho, son muy importantes para el procesamiento eficiente de transacciones en tales sistemas.

20.3.3.1 Memoria compartida

En una arquitectura de **memoria compartida** los procesadores y los discos tienen acceso a una memoria común, normalmente a través de un bus o de una red de interconexión. El beneficio de la memoria compartida es la extremada eficiencia en cuanto a la comunicación entre procesadores—cualquier procesador puede acceder a los datos de la memoria compartida sin necesidad de la intervención del software. Un procesador puede enviar mensajes a otros procesadores utilizando escrituras en la memoria de modo que la velocidad de envío es mucho mayor (normalmente es inferior a un microsegundo) que la que se alcanza con un mecanismo de comunicación. El inconveniente de las máquinas con memoria compartida es que la arquitectura no puede ir más allá de 32 o 64 procesadores porque el bus o la red de interconexión se convertirían en un cuello de botella (ya que está compartido por todos los procesadores). Llega un momento en el que no sirve de nada añadir más procesadores ya que éstos emplean la mayoría de su tiempo esperando su turno para utilizar el bus y así poder acceder a la memoria.

Las arquitecturas de memoria compartida suelen dotar a cada procesador de una memoria caché muy grande para evitar las referencias a la memoria compartida siempre que sea posible. No obstante, en la caché no podrán estar todos los datos y no podrá evitarse el acceso a la memoria compartida. Además, las cachés necesitan mantener la coherencia; esto es, si un procesador realiza una escritura en una posición de memoria, los datos de dicha posición de memoria se deberían actualizar en o eliminar de cualquier procesador donde estuvieran los datos en caché. El mantenimiento de la coherencia de la caché aumenta la sobrecarga cuando aumenta el número de procesadores. Por estas razones las máquinas con memoria compartida no pueden extenderse llegado un punto; las máquinas actuales con memoria compartida no pueden soportar más de 64 procesadores.

20.3.3.2 Disco compartido

En el modelo de **disco compartido** todos los procesadores pueden acceder directamente a todos los discos a través de una red de interconexión, pero los procesadores tienen memorias privadas. Las arquitecturas de disco compartido ofrecen dos ventajas respecto de las de memoria compartida. Primero, el bus de la memoria deja de ser un cuello de botella ya que cada procesador dispone de memoria propia. Segundo, esta arquitectura ofrece una forma barata para proporcionar una cierta **tolerancia ante fallos**: si falla un procesador (o su memoria) los demás procesadores pueden hacerse cargo de sus tareas ya que la base de datos reside en los discos, a los cuales tienen acceso todos los procesadores. Como se describía en el Capítulo 11, utilizando una arquitectura RAID también puede conseguirse que el subsistema de discos sea tolerante ante fallos por sí mismo. La arquitectura de disco compartido tiene aceptación en bastantes aplicaciones.

El problema principal de los sistemas de discos compartidos es, de nuevo, la ampliabilidad. Aunque el bus de la memoria no es un cuello de botella muy grande, la interconexión con el subsistema de discos es ahora el nuevo cuello de botella; esto es especialmente grave en situaciones en las que la base de datos realiza un gran número de accesos a los discos. Los sistemas de discos compartidos pueden soportar un mayor número de procesadores en comparación con los sistemas de memoria compartida, pero la comunicación entre los procesadores es más lenta (hasta unos pocos milisegundos si se carece de un hardware de propósito especial para comunicaciones) ya que se realiza a través de una red de interconexión.

20.3.3.3 Sin compartimiento

En un sistema **sin compartimiento** cada nodo de la máquina consta de un procesador, memoria y uno o más discos. Los procesadores de un nodo pueden comunicarse con un procesador de otro nodo utilizando una red de interconexión de alta velocidad. Un nodo funciona como el servidor de los datos almacenados en los discos que posee. El modelo sin compartimiento salva el inconveniente de requerir que todas las operaciones de E/S vayan a través de una única red de interconexión, ya que las referencias a los discos locales son servidas por los discos locales de cada procesador; solamente van por la red las peticiones, los accesos a discos remotos y las relaciones de resultados. Es más, habitualmente las redes de interconexión para los sistemas sin compartimiento se diseñan para ser ampliables por lo que su capacidad de transmisión crece a medida que se añaden nuevos nodos. Como consecuencia, las arquitecturas sin compartimiento son más ampliables y pueden soportar con facilidad un gran número de procesadores. El principal inconveniente de los sistemas sin compartimiento es el coste de comunicación y de acceso a discos remotos, coste que es mayor que el que se produce en las arquitecturas de memoria o disco compartido, ya que el envío de datos provoca la intervención del software en ambos extremos.

20.3.3.4 Jerárquica

La **arquitectura jerárquica** combina las características de las arquitecturas de memoria compartida, de disco compartido y sin compartimiento. A alto nivel el sistema está formado por nodos que están conectados mediante una red de interconexión y que no comparten ni memoria ni discos. Así, el nivel más alto es una arquitectura sin compartimiento. Cada nodo del sistema podría ser en realidad un sistema de memoria compartida con algunos procesadores. Alternativamente, cada nodo podría ser un sistema de disco compartido y cada uno de estos sistemas de disco compartido podría ser a su vez un sistema de memoria compartida. De esta manera, un sistema podría construirse como una jerarquía con una arquitectura de memoria compartida con pocos procesadores en la base, en lo más alto una arquitectura sin compartimiento y quizá una arquitectura de disco compartido en el medio. En la Figura 20.8d se muestra una arquitectura jerárquica con nodos de memoria compartida conectados entre sí con una arquitectura sin compartimiento. Hoy en día los sistemas paralelos comerciales de bases de datos pueden ejecutarse sobre varias de estas arquitecturas.

Los intentos de reducción de complejidad de programación de estos sistemas han dado lugar a las arquitecturas de **memoria virtual distribuida**, en las que hay una única memoria compartida desde el punto de vista lógico, pero hay varios sistemas de memoria disjuntos desde el punto de vista físico; se

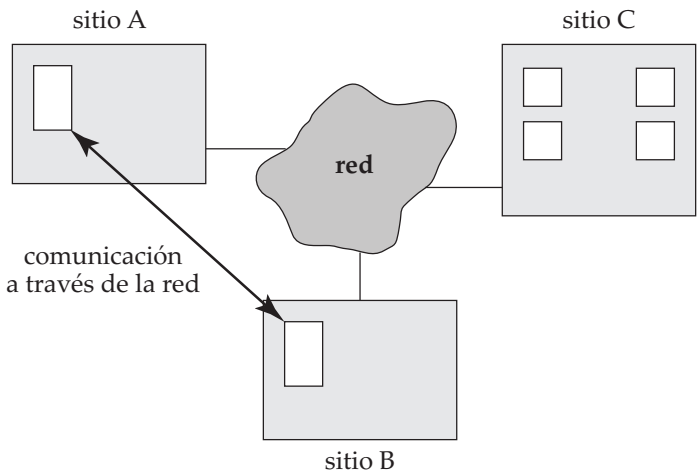


Figura 20.9 Un sistema distribuido.

obtiene una única vista del área de memoria virtual de estas memorias disjuntas mediante un hardware de asignación de memoria virtual en conjunción con un software extra. Dado que las velocidades de acceso son diferentes, dependiendo de si la página está disponible localmente o no, esta arquitectura también se denomina **arquitectura de memoria no uniforme** (NUMA, Nonuniform Memory Architecture).

20.4 Sistemas distribuidos

En un **sistema distribuido de bases de datos** se almacena la base de datos en varias computadoras. Los medios de comunicación como las redes de alta velocidad o las líneas telefónicas pueden poner en contacto las distintas computadoras de un sistema distribuido. No comparten ni memoria ni discos. Las computadoras de un sistema distribuido pueden variar en tamaño y función pudiendo abarcar desde las estaciones de trabajo a los grandes sistemas.

Dependiendo del contexto en el que se mencionen existen diferentes nombres para referirse a las computadoras que forman parte de un sistema distribuido, tales como **sitios** o **nodos**. Para enfatizar la distribución física de estos sistemas se emplean principalmente el término **sitio**. En la Figura 20.9 se muestra la estructura general de un sistema distribuido.

Las principales diferencias entre las bases de datos paralelas sin compartimientos y las bases de datos distribuidas son que las bases de datos distribuidas normalmente se encuentran en varios lugares geográficos distintos, se administran de forma separada y poseen una interconexión más lenta. Otra gran diferencia es que en un sistema distribuido existen dos tipos de transacciones: locales y globales. Una **transacción local** es aquella que accede a los datos del único sitio en el cual se inició la transacción. Por otra parte, una **transacción global** es la que, o bien accede a los datos situados en un sitio diferente de aquél en el que se inició la transacción, o bien accede a datos de varios sitios distintos.

Existen varias razones para construir sistemas distribuidos de bases de datos, incluyendo el compartimiento de los datos, la autonomía y la disponibilidad.

- **Datos compartidos**. La principal ventaja de construir un sistema distribuido de bases de datos es poder disponer de un entorno donde los usuarios puedan acceder desde una única ubicación a los datos que residen en otras ubicaciones. Por ejemplo, en un sistema de banca distribuida, donde cada sucursal almacena datos relacionados con dicha sucursal, es posible que un usuario de una de las sucursales acceda a los datos de otra sucursal. Sin esta capacidad, un usuario que quisiera transferir fondos de una sucursal a otra tendría que recurrir a algún mecanismo externo que pudiera enlazar los sistemas existentes.

- **Autonomía**. La principal ventaja de compartir datos por medio de distribución de datos es que cada ubicación es capaz de mantener un grado de control sobre los datos que se almacenan localmente. En un sistema centralizado, el administrador de bases de datos de la ubicación central controla la base de datos. En un sistema distribuido, existe un administrador de bases de datos global responsable de todo el sistema. Una parte de estas responsabilidades se delegan al administrador de bases de datos local de cada sitio. Dependiendo del diseño del sistema distribuido de bases de datos, cada administrador puede tener un grado diferente de **autonomía local**. La posibilidad de autonomía local es a menudo una de las grandes ventajas de las bases de datos distribuidas.
- **Disponibilidad**. Si un sitio de un sistema distribuido falla, los sitios restantes pueden seguir trabajando. En particular, si los elementos de datos están **replicados** en varios sitios, una transacción que necesite un elemento de datos en particular puede encontrarlo en varios sitios. De este modo, el fallo de un sitio no implica necesariamente la caída del sistema.

  El sistema puede detectar el fallo de un sitio y es posible que sea necesario aplicar acciones apropiadas para la recuperación del fallo. El sistema no debe seguir utilizando los servicios del sitio que falló. Finalmente, cuando el sitio que falló se recupera o se repara, debe haber mecanismos disponibles para integrarlo sin problemas de nuevo en el sistema.

  Aunque la recuperación ante un fallo es más compleja en los sistemas distribuidos que en los sistemas centralizados, la capacidad que tienen muchos sistemas de continuar trabajando a pesar del fallo en uno de los sitios produce una mayor disponibilidad. La disponibilidad es crucial para los sistemas de bases de datos que se utilizan en aplicaciones de tiempo real. Que, por ejemplo, una línea aérea pierda el acceso a los datos puede provocar la pérdida de potenciales compradores de billetes en favor de la competencia.

20.4.1 Un ejemplo de una base de datos distribuida

Considérese un sistema bancario compuesto por cuatro sucursales situadas en cuatro ciudades diferentes. Cada sucursal posee su propia computadora con una base de datos que alberga todas las cuentas abiertas en dicha sucursal. Así, cada una de estas instalaciones se considera un sitio. También hay un único sitio que mantiene la información relativa a todas las sucursales del banco. Cada sucursal dispone (entre otras) de una relación *cuenta*(*Esquema_cuenta*), donde

$$Esquema_cuenta = (número_cuenta, nombre_sucursal, saldo)$$

El sitio que contiene información acerca de las cuatro sucursales mantiene la relación *sucursal*(*Esquema_sucursal*), donde

$$Esquema_sucursal = (nombre_sucursal, ciudad_sucursal, activos)$$

Existen otras relaciones en los distintos sitios que serán ignoradas para los propósitos del ejemplo.

Para ilustrar la diferencia entre los dos tipos de transacciones considérese la transacción que suma 50 € a la cuenta C-177 situada en la sucursal de Cercedilla. La transacción se considera local si ésta comenzó en la sucursal de Cercedilla; en otro caso, se considera global. Una transacción que transfiere 50 € desde la cuenta C-177 a la cuenta C-305, que se encuentra en la sucursal de Guadarrama, es una transacción global ya que como resultado de su ejecución se accede a datos de dos sitios diferentes.

En un sistema distribuido de bases de datos ideal, los sitios deberían compartir un esquema global común (aunque algunas relaciones se puedan almacenar sólo en algunos sitios), todos los sitios deberían ejecutar el mismo software de gestión de bases de datos distribuidas, y los sitios deberían conocer la existencia de los demás. Si una base de datos distribuida se construye partiendo de cero, realmente debería ser posible lograr los objetivos anteriores. Sin embargo, en la realidad, una base de datos distribuida se tiene que construir enlazando múltiples sistemas de bases de datos que ya existen, cada uno con su propio esquema y posiblemente ejecutando diferente software de gestión de bases de datos. A veces, tales sistemas reciben el nombre de **sistemas de bases de datos múltiples** o **sistemas de bases de datos distribuidos y heterogéneos**. En el Apartado 22.8 se estudian dichos sistemas y se muestra cómo conseguir un cierto grado de control global a pesar de la heterogeneidad de los sistemas que lo componen.

20.4.2 Aspectos de la implementación

La atomicidad de las transacciones es un aspecto importante de la construcción de un sistema distribuido de bases de datos. Si una transacción se ejecuta a lo largo de dos sitios, a menos que los diseñadores del sistema sean cuidadosos, puede comprometerse en un sitio y cancelarse en otro, lo que conduciría a un estado de inconsistencia. Los protocolos de compromiso de transacciones aseguran que tales situaciones no se produzcan. El *protocolo de compromiso de dos fases* (C2F) es el más utilizado de estos protocolos.

La idea básica de C2F es que cada sitio ejecuta la transacción justo hasta antes del compromiso, y entonces deja la decisión del compromiso a un único sitio coordinador; se dice que en ese punto la transacción está en estado *preparada* en el sitio. El coordinador decide comprometer la transacción sólo si la transacción alcanza el estado preparada en cada sitio donde se ejecutó; en otro caso (por ejemplo, si la transacción se canceló en algún sitio), el coordinador decide cancelar la transacción. Todos los sitios donde la transacción se ejecutó deben acatar la decisión del coordinador. Si un sitio falla cuando una transacción se encuentra en estado preparada, cuando el sitio se recupere del fallo debería estar en posición de comprometer o cancelar la transacción dependiendo de la decisión del coordinador. El protocolo C2F se describe en detalle en el Apartado 22.4.1.

El control de concurrencia es otra característica de una base de datos distribuida. Como una transacción puede acceder a elementos de datos de varios sitios, los administradores de transacciones de varios sitios pueden necesitar coordinarse para implementar el control de concurrencia. Si se utiliza bloqueo (como casi siempre sucede en la práctica), el bloqueo se puede realizar de forma local en los sitios que contienen los elementos de datos accedidos, pero también existe posibilidad de un interbloqueo que involucre a transacciones originadas en múltiples sitios. Por lo tanto, es necesario llevar la detección de interbloqueos a lo largo de múltiples sitios. Los fallos son más comunes en los sistemas distribuidos dado que no sólo las computadoras pueden fallar, sino que también pueden fallar los enlaces de comunicaciones. La réplica de los elementos de datos, que es la clave para el funcionamiento continuado de las bases de datos distribuidas cuando ocurren fallos, complica aún más el control de concurrencia. El Apartado 22.5 proporciona más detalles sobre el control de concurrencia en bases de datos distribuidas.

Los modelos estándar de transacciones, basados en múltiples acciones llevadas a cabo por una única unidad de programa, son a menudo inapropiadas para realizar tareas que cruzan los límites de las bases de datos que no pueden o no cooperarán para implementar protocolos como C2F. Para estas tareas se utilizan generalmente técnicas alternativas, basadas en *mensajería persistente* para las comunicaciones.

Los modelos estándar de transacciones, basados en múltiples acciones llevadas a cabo por una única unidad de programa, son a menudo inapropiadas para realizar tareas que cruzan los límites de las bases de datos que no pueden o no cooperarán para implementar protocolos como el C2F. Para estas tareas se utilizan generalmente técnicas alternativas, basadas en *mensajería persistente* para las comunicaciones.

Cuando las tareas a realizar son complejas, involucrando múltiples bases de datos y múltiples interacciones con humanos, la coordinación de las tareas y asegurar las propiedades de las transacciones para las tareas se vuelve más complicado. Los *sistemas de gestión de flujos de trabajo* son sistemas diseñados para ayudar en la realización de dichas tareas. El Apartado 22.4.3 describe la mensajería persistente, mientras que el Apartado 25.2 describe los sistemas de gestión de flujos de trabajo.

En caso de que una empresa tenga que escoger entre una arquitectura distribuida y una arquitectura centralizada para implementar una aplicación, el arquitecto del sistema debe sopesar las ventajas frente a las desventajas de la distribución de datos. Ya se han examinado las ventajas de utilizar bases de datos distribuidas. El principal inconveniente de los sistemas distribuidos de bases de datos es la complejidad añadida que es necesaria para garantizar la coordinación apropiada entre los sitios. Esta creciente complejidad tiene varias facetas:

- **Coste de desarrollo del software**. La implementación de un sistema distribuido de bases de datos es más difícil y, por tanto, más costoso.
- **Mayor probabilidad de errores**. Como los sitios que constituyen el sistema distribuido operan en paralelo es más difícil asegurarse de la corrección de los algoritmos, del funcionamiento especial durante los fallos de parte del sistema así como de la recuperación. Son probables errores extremadamente sutiles.

- **Mayor sobrecarga de procesamiento**. El intercambio de mensajes y el cómputo adicional necesario para conseguir la coordinación entre los distintos sitios constituyen una forma de sobrecarga que no surge en los sistemas centralizados.

Existen varios enfoques acerca del diseño de las bases de datos distribuidas que abarcan desde los diseños completamente distribuidos hasta los que incluyen un alto grado de centralización. Se estudiarán en el Capítulo 22.

20.5 Tipos de redes

Las bases de datos distribuidas y los sistemas cliente–servidor se construyen en torno a las redes de comunicación. Existen básicamente dos clases de redes: las **redes de área local** y las **redes de área amplia**. La diferencia principal entre ambas es la forma en que están distribuidas geográficamente. Las redes de área local están compuestas por procesadores distribuidos en áreas geográficas pequeñas tales como un único edificio o varios edificios adyacentes. Por su parte, las redes de área amplia se componen de un número determinado de procesadores autónomos que están distribuidos a lo largo de una extensa área geográfica (como puede ser España o el mundo entero). Estas diferencias implican importantes variaciones en la velocidad y en la fiabilidad de la red de comunicación y quedan reflejadas en el diseño del sistema operativo distribuido.

20.5.1 Redes de área local

Las **redes de área local** (LAN, Local Area Network) (Figura 20.10) surgen a principios de los 70 como una forma de comunicación y de compartimiento de datos entre varias computadoras. La gente se dio cuenta de que en muchas empresas era más económico tener muchas computadoras pequeñas, cada una de ellas con sus propias aplicaciones, que un enorme y único sistema. La conexión de estas pequeñas computadoras formando una red parece un paso natural porque, probablemente, cada pequeña computadora necesite acceder a un conjunto complementario de dispositivos periféricos (como discos e impresoras) y porque en una empresa suele ser necesaria el compartimiento de algunos datos.

Las LAN se utilizan generalmente en un entorno de oficina. Todos los puestos de estos sistemas están próximos entre sí por lo que los enlaces de comunicación suelen poseer una mayor velocidad y una tasa de errores más baja que la que se da en las redes de área amplia. Los enlaces más comunes en una red de área local son el par trenzado, el cable coaxial, la fibra óptica y, cada vez más, las conexiones inalámbricas. La velocidad de comunicación varía entre unos pocos megabits por segundo (en las redes de área

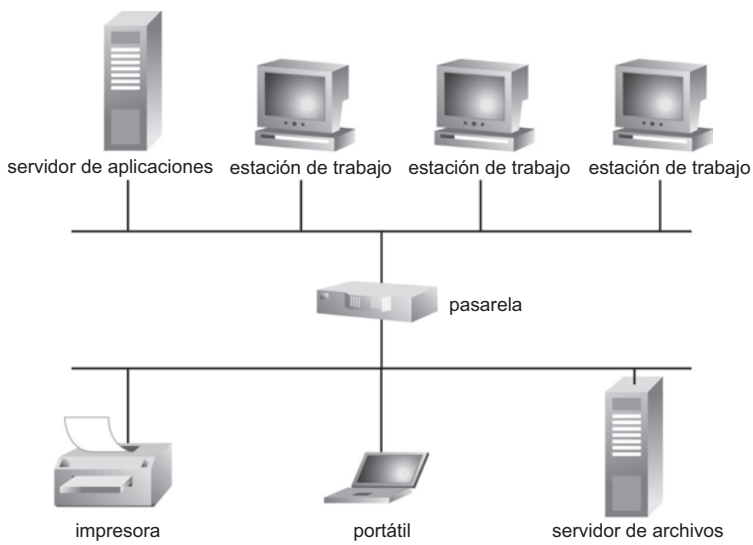


Figura 20.10 Red de área local.

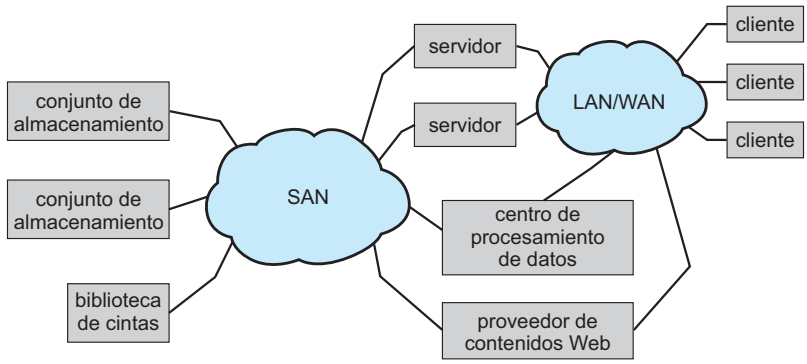


Figura 20.11 Red de área de almacenamiento.

local inalámbricas), a un gigabit por segundo para Gigabit Ethernet. La Ethernet estándar funciona a 10 megabits por segundo, mientras que Fast Ethernet llega a 100 megabits por segundo.

Una **red de área de almacenamiento** (SAN, Storage Area Network) es un tipo especial de red de área local de alta velocidad destinada a conectar numerosos bancos de dispositivos de almacenamiento (discos) a las computadoras que utilizan los datos (véase la Figura 20.11).

Así, las redes de área de almacenamiento ayudan a construir *sistemas de discos compartidos* a gran escala. El motivo para utilizar redes de área de almacenamiento para conectar múltiples computadoras a grandes bancos de dispositivos de almacenamiento es esencialmente el mismo que para las bases de datos de disco compartido:

- Ampliabilidad añadiendo más computadoras.
- Alta disponibilidad, ya que los datos son accesibles incluso si falla alguna computadora.

Las organizaciones RAID se utilizan en dispositivos de almacenamiento para asegurar la alta disponibilidad de los datos, permitiendo continuar al procesamiento incluso si falla algún disco. Las redes de área de almacenamiento se construyen normalmente con redundancia, como múltiples caminos entre nodos, así si un componente como un enlace o una conexión a la red falla, la red continúa funcionando.

20.5.2 Redes de área amplia

Las **redes de área amplia** (WAN, Wide Area Networks) surgen a finales de los 60 principalmente como un proyecto de investigación académica para proporcionar una comunicación eficiente entre varios lugares permitiendo que una gran comunidad de usuarios pudiera compartir hardware y software de una manera conveniente y económica. A principios de los años 60 se desarrollaron sistemas que permitían que terminales remotos se conectaran a una computadora central a través de la línea telefónica, pero no eran verdaderas WAN. *Arpanet* fue la primera WAN que se diseñó y se desarrolló. El trabajo en Arpanet comenzó en 1968. Arpanet ha crecido de tal forma que ha pasado de ser una red experimental de cuatro puestos a una red de redes extendida por todo el mundo, **Internet**, abarcando a cientos de millones de sistemas de computación. Los enlaces típicos de Internet son las líneas de fibra óptica y, a veces, los canales vía satélite. Las transferencias de datos para los enlaces de área amplia varían normalmente entre los pocos megabits por segundo y los cientos de gigabits por segundo. El último enlace, hasta el puesto del usuario, se basa a menudo en la tecnología de *línea de suscriptor digital* (DSL, Digital Subscriber Line) (que soporta unos pocos megabits por segundo), o módem de cable (que soporta 10 megabits por segundo) o conexiones de módem telefónico sobre líneas telefónicas (que soportan hasta 56 kilobits por segundo). Las redes WAN pueden pertenecer a dos tipos:

- En las WAN de **conexión discontinua**, como las basadas en conexiones por radio, las computadoras están conectadas a la red sólo durante intervalos del tiempo.
- En las WAN de **conexión continua**, como Internet, las computadoras están conectadas a la red continuamente.

Las redes que no están continuamente conectadas no suelen permitir las transacciones entre distintos sitios, pero pueden almacenar copias locales de los datos remotos y actualizarlas periódicamente (por ejemplo, todas las noches). Para las aplicaciones en las que la consistencia no es un factor crítico, como ocurre en el compartimiento de documentos, los sistemas de software de grupo como Lotus Notes permiten realizar localmente las actualizaciones de los datos remotos y propagar más tarde dichas actualizaciones al sitio remoto. Debe detectarse y resolverse el riesgo potencial de conflicto entre varias actualizaciones realizadas en sitios diferentes. Más adelante, en el Apartado 24.5.4, se describe un mecanismo para detectar actualizaciones conflictivas; el mecanismo de resolución de actualizaciones conflictivas es, sin embargo, dependiente de la aplicación.

20.6 Resumen

- Los sistemas centralizados de bases de datos se ejecutan completamente en una única computadora. Con el crecimiento de las computadoras personales y las redes de área local, se ha ido desplazando hacia el lado del cliente la funcionalidad de la fachada de la base de datos de modo que los sistemas servidores provean la funcionalidad del sistema subyacente. Los protocolos de interfaz cliente–servidor han ayudado al crecimiento de los sistemas de bases de datos cliente–servidor.
- Los servidores pueden ser servidores de transacciones o servidores de datos, aunque el uso de los servidores de transacciones excede ampliamente el uso de los servidores de datos para proporcionar servicios de bases de datos.
  - Los servidores de transacciones tienen múltiples procesos, ejecutándose posiblemente en múltiples procesadores. Dado que estos procesos tienen acceso a los datos comunes, como la memoria intermedia de la base de datos, los sistemas almacenan dichos datos en memoria compartida. Además de los procesos que gestionan consultas, hay procesos del sistema que realizan tareas como la gestión de los bloqueos y del registro y los puntos de revisión.
  - Los sistemas servidores de datos suministran datos sin formato a los clientes. Tales sistemas se esfuerzan en minimizar la comunicación entre clientes y servidores usando caché de datos y de bloqueos en los clientes. Los sistemas paralelos de bases de datos utilizan optimizaciones similares.
- Los sistemas paralelos de bases de datos consisten en varios procesadores y varios discos conectados a través de una red de interconexión de alta velocidad. La ganancia de velocidad mide cuánto puede incrementarse la velocidad de procesamiento al incrementarse el paralelismo dada una transacción. La ampliabilidad mide lo bien que se gestiona un mayor número de transacciones cuando se incrementa el paralelismo. La interferencia, el sesgo y los costes de inicio actúan como barreras para obtener la ganancia de velocidad y la ampliabilidad ideales.
- Las arquitecturas paralelas de bases de datos pueden clasificarse en arquitecturas de memoria compartida, de disco compartido, sin compartimiento o jerárquicas. Estas arquitecturas tienen distintos compromisos entre la ampliabilidad y la velocidad de comunicación.
- Un sistema de bases de datos distribuido es un conjunto de bases de datos parcialmente independientes que (idealmente) comparten un esquema común y coordinan el procesamiento de transacciones que acceden a datos remotos. Los sistemas se comunican entre sí a través de una red de comunicación que gestiona el encaminamiento y las estrategias de conexión.
- Existen dos tipos principales de redes de comunicación: las redes de área local y las de área amplia. Las redes de área local conectan nodos que están distribuidos sobre áreas geográficas pequeñas tales como un único edificio o varios edificios adyacentes. Las redes de área amplia conectan nodos a lo largo de una extensa área geográfica. Actualmente, la red de área amplia más extensa que se utiliza es Internet.

  Las redes de área de almacenamiento son un tipo especial de redes de área local diseñadas para proporcionar interconexión rápida entre grandes bancos de dispositivos de almacenamiento y múltiples computadoras.

Términos de repaso

- Sistemas centralizados.
- Sistemas servidores.
- Paralelismo de grano grueso.
- Paralelismo de grano fino.
- Estructura de proceso de la base de datos.
- Exclusión mutua.
- Hebra.
- Procesos del servidor:
  - ☐ Proceso administrador de bloqueos.
  - ☐ Proceso escritor de bases de datos.
  - ☐ Proceso escritor del registro.
  - ☐ Proceso punto de revisión.
  - ☐ Proceso monitor de procesos.
- Sistemas cliente–servidor.
- Servidor de transacciones.
- Servidor de consultas.
- Servidor de datos.
  - ☐ Preextracción.
  - ☐ Liberación de bloqueos.
  - ☐ Caché de datos.
  - ☐ Coherencia de caché.
  - ☐ Caché de bloqueos.
  - ☐ Comunicar.
- Sistemas paralelos.
- Productividad.
- Tiempo de respuesta.
- Ganancia de velocidad:
  - ☐ Lineal.
  - ☐ Sublineal.
- Ampliabilidad:
  - ☐ Lineal.
  - ☐ Sublineal.
  - ☐ Por lotes.
  - ☐ De transacciones.
- Costes de inicio.
- Interferencia.
- Sesgo.
- Redes de interconexión.
  - ☐ Bus.
  - ☐ Malla.
  - ☐ Hipercubo.
- Arquitecturas paralelas de bases de datos.
  - ☐ Memoria compartida.
  - ☐ Disco compartido (agrupaciones).
  - ☐ Sin compartimiento.
  - ☐ Jerárquica.
- Tolerancia ante fallos.
- Memoria virtual distribuida.
- Arquitectura de memoria no uniforme.
- Sistemas distribuidos.
- Bases de datos distribuidas.
  - ☐ Sitios (nodos).
  - ☐ Transacción local.
  - ☐ Transacción global.
  - ☐ Autonomía local.
- Sistema con múltiples bases de datos.
- Tipos de redes.
  - ☐ Redes de área local (LAN).
  - ☐ Redes de área amplia (WAN).
  - ☐ Redes de área de almacenamiento (SAN).

Ejercicios prácticos

20.1 En lugar de almacenar estructuras compartidas en memoria compartida, una arquitectura alternativa podría ser almacenarlas en la memoria local de un proceso especial, y acceder a los datos compartidos mediante comunicación entre procesos con el proceso. ¿Cuál sería la desventaja de dicha arquitectura?

20.2 La máquina que hace de servidor en los sistemas cliente–servidor típicos es mucho más potente que los clientes, es decir, su procesador es más rápido, puede tener varios procesadores, tiene más memoria y tiene discos de mayor capacidad. En vez de esto, considérese el caso en el que los clientes y el servidor tuvieran exactamente la misma potencia. ¿Tendría sentido construir un sistema cliente–servidor en ese caso? ¿Por qué? ¿Qué caso se ajustaría mejor a una arquitectura servidora de datos?

20.3 Considérese un sistema de base de datos orientada a objetos sobre una arquitectura cliente–servidor en la que el servidor actúa como servidor de datos.

a. ¿Cuál es el efecto de la velocidad de interconexión entre el cliente y el servidor en los casos de envío de páginas y de objetos?
b. Si se utiliza envío de páginas, la caché de datos en el cliente puede organizarse como una caché de objetos o una caché de páginas. La caché de páginas almacena los datos en unidades de páginas mientras que la caché de objetos almacena los datos en unidades de objetos. Supóngase que los objetos son más pequeños que una página. Descríbase una ventaja de la caché de objetos frente a la caché de páginas.

20.4 Supóngase una transacción escrita en C con código SQL incorporado que ocupa el 80 % del tiempo en ejecutar el código SQL y sólo el 20 % restante en el código C. ¿Qué ganancia de velocidad puede esperarse si sólo se hace paralelo el código SQL? Justifíquese la respuesta.

20.5 Considérese una red basada en líneas de acceso telefónico en la que los sitios se comunican periódicamente, por ejemplo, todas las noches. Estas redes suelen tener un servidor y varios clientes. Los sitios que actúan como clientes están conectados sólo con el servidor e intercambian los datos con el resto de clientes almacenándolos en el servidor y recuperando los almacenados por otros clientes en el servidor. ¿Cuál es la ventaja de tal arquitectura frente a una en la que un sitio pueda intercambiar datos con otro mediante acceso telefónico directo?

Ejercicios

20.6 ¿Por qué es relativamente fácil trasladar una base de datos desde una máquina con un único procesador a otra con varios procesadores si no es necesario hacer paralelas las consultas individuales?

20.7 Las arquitecturas servidoras de transacciones son populares entre las bases de datos relacionales cliente–servidor, en las que las transacciones son cortas. Por el contrario, las arquitecturas servidoras de datos son populares entre los sistemas cliente–servidor de bases de datos orientadas a objetos, donde las transacciones son relativamente largas. Dense dos razones por las que los servidores de datos puedan ser populares entre las bases de datos orientadas a objetos y no los sean entre las bases de datos relacionales.

20.8 ¿Qué es la liberación de bloqueos y bajo qué condiciones es necesaria? ¿Por qué no es necesario si la unidad de envío de datos es un elemento?

20.9 Supóngase una persona a cargo de las operaciones de la base de datos de una empresa cuyo trabajo principal es el de procesar transacciones. Supóngase además que la empresa crece rápidamente cada año y que el sistema informático actual se ha quedado pequeño. Cuando se escoja una nueva computadora paralela, ¿qué factor será más importante: la ganancia de velocidad, la ampliabilidad por lotes o la ampliabilidad de transacciones? ¿Por qué?

20.10 Los sistemas de bases de datos se implementan normalmente como un conjunto de procesos (o hebras) que comparten un área de memoria.

a. ¿Cómo se controla el acceso a la memoria compartida?
b. ¿Es apropiado el bloqueo de dos fases para secuenciar el acceso a las estructuras de datos en la memoria compartida? Explíquese la respuesta.

20.11 En un sistema de procesamiento de transacciones, ¿cuáles son los factores que trabajan en contra de la ampliabilidad lineal? ¿Cuál de esos factores es probablemente el más importante en cada una de estas arquitecturas: memoria compartida, disco compartido y sin compartimiento?

20.12 La velocidad de los procesadores se ha incrementado mucho más rápidamente que las velocidades de acceso a memoria. ¿Qué impacto tiene esto sobre el número de procesadores que pueden compartir de modo efectivo una memoria común?

20.13 Considérese un banco que dispone de un conjunto de sitios en los que se ejecuta un sistema de base de datos. Supóngase que la transferencia electrónica de dinero entre ellos es el único modo de interacción de las bases de datos. ¿Puede tal sistema ser calificado como distribuido? ¿Por qué?

Notas bibliográficas

Hennessy et al. [2002] proporcionan una excelente introducción al área de la arquitectura de computadoras.

Gray y Reuter [1993] dan una descripción en su libro del procesamiento de transacciones incluyendo la arquitectura de cliente–servidor y los sistemas distribuidos. Las notas bibliográficas del Capítulo 4 proporcionan referencias a más infomración sobre ODBC, JDBC y otras interfaces de programación para aplicaciones.

Carey et al. [1991] y Franklin et al. [1993] describen técnicas de caché de datos para los sistemas de bases de datos cliente–servidor. Biliris y Orenstein [1994] tratan los sistemas de gestión de almacenamiento de objetos incluyendo aspectos relacionados con los sistemas cliente–servidor. En Franklin et al. [1992] y Mohan y Narang [1994] pueden encontrarse algunas técnicas para la recuperación en sistemas cliente–servidor.

DeWitt y Gray [1992] describen los sistemas paralelos de bases de datos incluyendo sus propias arquitecturas y medidas de rendimiento. Duncan [1990] introduce una vista de las arquitecturas paralelas de computadoras. Dubois y Thakkar [1992] es una colección de artículos sobre las arquitecturas ampliables de memoria compartida. Las agrupaciones DEC con Rdb constituyen uno de los primeros usuarios comerciales de la arquitectura de bases de datos de disco compartido Rdb ahora es propiedad de Oracle y se denomina Oracle Rdb. Digital Equipment Corporation (DEC) es ahora propiedad de Compaq. La máquina de base de datos Teradata fue uno de los primeros sistemas comerciales que utilizaron la arquitectura sin compartimiento de bases de datos. También se construyeron sobre arquitecturas sin compartimiento los prototipos de investigación Grace y Gamma.

El libro de texto de Ozsu y Valduriez [1999] trata los sistemas distribuidos de bases de datos. Se pueden encontrar más referencias sobre los sistemas de bases de datos paralelos y distribuidos en las notas bibliográficas de los Capítulos 21 y 22, respectivamente.

Comer y Droms [2003] y Thomas [1996] describen las redes de computadoras e Internet. Tanenbaum [2002] y Halsall [1996] proporcionan revisiones generales de las redes de computadoras.

CAPÍTULO 21

Bases de datos paralelas

En este capítulo se estudian los algoritmos fundamentales de los sistemas paralelos de bases de datos basados en el modelo de datos relacional. En concreto, este capítulo se centra en la ubicación de los datos en varios discos y en la evaluación en paralelo de las operaciones relacionales, dos conceptos que han sido esenciales para el éxito de las bases de datos paralelas.

21.1 Introducción

Quince años atrás los sistemas paralelos de bases de datos habían sido casi descartados, incluso por algunos de sus más firmes partidarios. Actualmente los venden con éxito casi todas las marcas de bases de datos. Este cambio ha sido impulsado por varios desafíos:

- Los requisitos transaccionales de las empresas han aumentado con el creciente empleo de las computadoras. Además, el crecimiento de Web ha creado muchos sitios con millones de visitantes, y la creciente cantidad de datos obtenidos de esos visitantes ha generado en muchas empresas bases de datos enormes.
- Las empresas utilizan volúmenes crecientes de datos—como los relativos a lo que se compra, los enlaces Web que se pulsan o la hora a la que se realizan las llamadas telefónicas—para planificar sus actividades y sus tarifas. Las consultas utilizadas para estos fines se denominan consultas de **ayuda a la toma de decisiones** y pueden llegar a necesitar varios terabytes de datos. Los sistemas con un solo procesador no son capaces de tratar tales volúmenes de datos a la velocidad necesaria.
- La naturaleza orientada a conjuntos de las consultas a las bases de datos se presta de modo natural a la paralelización. Diferentes sistemas comerciales y experimentales han demostrado la potencia y la dimensionabilidad del procesamiento paralelo de las consultas.
- Al abaratarse los microprocesadores, las máquinas paralelas se han popularizado y se han vuelto relativamente baratas.

Como se ha estudiado en el Capítulo 20, el paralelismo se utiliza para mejorar la velocidad, pues las consultas se ejecutan más rápido debido a que disponen de más recursos, como procesadores y discos. El paralelismo también se utiliza para proporcionar dimensionabilidad, pues la creciente carga de trabajo se trata sin incrementar el tiempo de respuesta, mediante un aumento del grado de paralelismo.

En el Capítulo 20 se esbozaron las diferentes arquitecturas de los sistemas paralelos de bases de datos: de memoria compartida, de discos compartidos, sin compartimiento y arquitecturas jerárquicas. En resumen, en las arquitecturas de memoria compartida todos los procesadores comparten memoria y discos; en las arquitecturas de disco compartido los procesadores tienen memorias independientes

pero comparten los discos; en las arquitecturas sin compartimiento los procesadores no comparten ni la memoria ni los discos; y las arquitecturas jerárquicas tienen nodos que no comparten entre sí ni la memoria ni los discos, pero cada nodo tiene internamente una arquitectura de memoria o de disco compartido.

21.2 Paralelismo de E/S

En su forma más sencilla, el término **paralelismo de E/S** se refiere a la división de las relaciones entre varios discos para reducir el tiempo necesario de su recuperación. La forma más habitual de división de datos en un entorno de bases de datos paralelas es la *división horizontal*. En la **división horizontal**, las tuplas de cada relación se dividen (o desagrupan) entre varios discos, de modo que cada tupla resida en uno distinto. Se han propuesto varias estrategias de división.

21.2.1 Técnicas de división

Se presentan tres estrategias básicas para la división de datos. Supóngase que hay n discos, $D_0, D_1, \ldots, D_{n-1}$, entre los cuales se van a dividir los datos.

- **Turno rotatorio**. La relación se explora en un orden cualquiera y la i-ésima tupla se envía al disco numerado $D_{i \bmod n}$. El esquema de turno rotatorio asegura una distribución homogénea de las tuplas entre los discos; es decir, cada disco tiene aproximadamente el mismo número de tuplas que los demás.
- **División por asociación**. En esta estrategia de desagrupación uno o varios atributos del esquema de la relación dada se designan como atributos de la división. Se escoge una función de asociación cuyo rango sea $\{0, 1, \ldots, n-1\}$. Cada tupla de la relación original se asocia en términos de los atributos de la división. Si la función de asociación devuelve i, la tupla se ubica en el disco D_i.
- **División por rangos**. Esta estrategia distribuye rangos contiguos de valores de los atributos a cada disco. Se escoge un atributo de división, A, como **vector de división**. La relación se divide de la manera siguiente. Sea $[v_0, v_1, \ldots, v_{n-2}]$ el vector de división, tal que , si $i < j$, entonces $v_i < v_j$. Considérese una tupla t tal que $t[A] = x$. Si $x < v_0$ entonces t se ubica en el disco D_0. Si $x \geq v_{n-2}$, entonces t se ubica en el disco D_{n-1}. Si $v_i \leq x < v_{i+1}$, entonces t se ubica en el disco D_{i+1}.

  Por ejemplo, en una división por rangos con tres discos numerados del cero al dos se pueden asignar las tuplas con valores menores que cinco al disco cero, las de valores entre cinco y cuarenta al disco uno, y las que tienen valores mayores que cuarenta al disco dos.

21.2.2 Comparación de las técnicas de división

Una vez dividida una relación entre varios discos, se puede recuperar en paralelo utilizándolos todos. De modo parecido, cuando se está dividiendo una relación, se puede escribir en paralelo en varios discos. De esta manera, las velocidades de transferencia de lectura o de escritura de la relación completa son mucho mayores con paralelismo de E/S que sin él. Sin embargo, la lectura de la relación completa, o *exploración de la relación* es sólo uno de los tipos de acceso a los datos. El acceso a los datos puede clasificarse de la manera siguiente:

1. Exploración de la relación completa.
2. Localización de tuplas de manera asociativa, (por ejemplo, *nombre_empleado* = "García"); estas consultas, denominadas **consultas concretas**, buscan tuplas que tengan un valor concreto para un atributo determinado.
3. Localización de todas las tuplas cuyo valor de un atributo dado se halle en un rango especificado (por ejemplo, $10.000 < sueldo < 20.000$); estas consultas se denominan **consultas de rango**.

Las diferentes técnicas de división permiten estos tipos de acceso con diferentes niveles de eficacia:

- **Turno rotatorio**. El esquema se adapta perfectamente a las aplicaciones que desean leer secuencialmente la relación completa en cada consulta. Con este esquema tanto las consultas concretas como las de rango son difíciles de procesar, ya que se deben emplear en la búsqueda todos y cada uno de los n discos.
- **División por asociación**. Este esquema se adapta mejor a las consultas concretas basadas en el atributo de división. Por ejemplo, si se divide una relación en términos del atributo *número_teléfono*, se puede responder a la consulta "Buscar el registro del empleado con *número_teléfono* = 5553333" aplicando la función de división por asociación a 5553333 y buscando luego en ese disco. Dirigir la consulta a un solo disco ahorra el coste inicial de comenzar una consulta en varios discos y deja a los demás discos libres para procesar otras consultas.

  La división por asociación también resulta útil para las exploraciones secuenciales de toda la relación. Si la función de asociación es una buena función aleatoria y los atributos de división constituyen una clave de la relación, el número de tuplas en cada uno de los discos es aproximadamente el mismo, sin mucha varianza. Por tanto, el tiempo empleado para explorar la relación es aproximadamente $1/n$ del necesario para explorar la relación en un sistema de disco único.

  El esquema, sin embargo, no se adapta bien a las búsquedas concretas en términos de atributos que no sean de división. La división basada en asociación tampoco resulta muy adecuada para las respuestas a consultas de rangos, dado que, generalmente, las funciones de asociación no conservan la proximidad dentro de ellos. Por tanto, hace falta explorar todos los discos para responder a las consultas de rango.
- **División por rangos**. Este esquema se adapta bien a las consultas concretas y de rango basadas en el atributo de división. Para las consultas concretas se puede analizar el vector de división para encontrar el disco en el que reside la tupla. En las consultas de rango se consulta el vector de división para hallar el rango de discos en que pueden residir las tuplas. En ambos casos la búsqueda se limita exactamente a aquellos discos que pueden tener tuplas de interés.

  Una ventaja es que, si sólo hay unas cuantas tuplas en el rango consultado, la consulta se suele enviar a un solo disco, en vez de a todos. Dado que se pueden utilizar otros discos para responder a otras consultas, la división por rangos da lugar a una mayor productividad de consultas al tiempo que se mantiene un buen tiempo de respuesta. Por otro lado, si hay muchas tuplas en el rango consultado (como ocurre cuando el rango consultado es una fracción mayor del dominio de la relación), es necesario recuperar muchas tuplas de pocos discos, lo que origina un cuello de botella de E/S (punto caliente) en esos discos. En este ejemplo de **sesgo de ejecución** todo el procesamiento tiene lugar en una partición (o en sólo unas pocas). Por el contrario, la división por asociación y la de turno rotatorio emplearían todos los discos para esas consultas, lo que proporcionaría un tiempo de respuesta menor para aproximadamente la misma productividad.

El tipo de división también afecta a otras operaciones relacionales, como las reuniones (véase el Apartado 21.5). De este modo, la elección de la técnica de división también depende de las operaciones que haya que ejecutar. En general, se prefieren las divisiones por asociación y en rangos al turno rotatorio.

En un sistema con muchos discos, el número de discos en los que se divide una relación puede escogerse de la manera siguiente. Si una relación sólo contiene unas pocas tuplas que caben en un solo bloque de disco, es mejor asignarla a un solo disco. Las relaciones grandes se dividen preferiblemente entre todos los discos disponibles. Si una relación consta de m bloques de disco y hay n discos disponibles en el sistema, se deberán asignar a la relación $\mathbf{min}(m, n)$ discos.

21.2.3 Tratamiento del sesgo

La distribución de las tuplas al dividir una relación (excepto mediante el turno rotatorio) puede estar **sesgada**, con un porcentaje alto de tuplas ubicado en algunas divisiones y porcentajes menores en otras. Los diferentes tipos de sesgo se clasifican como:

- Sesgo de los valores de los atributos.
- Sesgo de la división.

El **sesgo de los valores de los atributos** se refiere al hecho de que algunos valores aparezcan en los atributos de división de muchas tuplas. Todas las tuplas con el mismo valor del atributo de división terminan en la misma partición, lo que da lugar al sesgo. El **sesgo de la división** se refiere al hecho de que puede haber un desequilibrio de carga en la división, aunque no haya sesgo en los atributos.

El sesgo de los valores de los atributos puede dar lugar a una división sesgada independientemente de que se utilice división por rangos o por asociación. Si no se escoge cuidadosamente el vector de división, la división por rangos puede dar lugar a sesgo de división. El sesgo de división es menos probable en la división por asociación, si se ha escogido una buena función de asociación.

Como se indicó en el Apartado 20.3.1, incluso un sesgo pequeño puede dar lugar a una disminución significativa del rendimiento. El sesgo es un problema que se agrava al aumentar el grado de paralelismo. Por ejemplo, si una relación de mil tuplas se divide en diez partes y la división está sesgada, puede haber algunas particiones de tamaño menor que cien y otras de tamaño mayor que cien; incluso si una partición llega a tener tamaño doscientos, la aceleración que se obtendría al acceder en paralelo a las particiones sólo sería de cinco, en lugar del valor de diez que cabría esperar. Si la misma relación tiene que dividirse en cien partes, cada partición tendrá de media diez tuplas. Si una partición llega a tener cuarenta tuplas (lo que es posible dado el gran número de particiones) el aumento de velocidad que se obtendría al acceder a ellas en paralelo sería de veinticinco, en vez de cien. Por tanto, se puede ver que la pérdida de aceleración debida al sesgo aumenta con el paralelismo.

Se puede construir un **vector de división por rangos equilibrado** mediante ordenación. La relación primero se ordena según los atributos de división. A continuación se explora de forma ordenada. Después de que se haya leído cada $1/n$ de la relación, se añade el valor del atributo de división de la siguiente tupla al vector de división. En este caso, n denota el número de particiones que hay que crear. En caso de que haya muchas tuplas con el mismo valor para el atributo de división, la técnica puede seguir generando algo de sesgo. El inconveniente principal de este método es la sobrecarga de E/S debida a la ordenación inicial.

La sobrecarga de E/S debida a la construcción de vectores de división por rangos equilibrados se puede reducir creando y almacenando una tabla de frecuencias, o **histograma**, de los valores de los atributos para todos los atributos de cada relación. La Figura 21.1 muestra un ejemplo de un histograma para un atributo de tipo entero que toma valores en el rango de uno a veinticinco. Los histogramas ocupan poco espacio, por lo que en el catálogo se pueden almacenar histogramas de varios atributos diferentes. Resulta sencillo crear una función de división por rangos equilibrada dado un histograma de los atributos de división. Si no se almacena el histograma es posible calcularlo de manera aproximada tomando muestras de la relación. Estas muestras son las tuplas de un subconjunto de los bloques de disco elegido aleatoriamente.

Otro enfoque para minimizar el efecto del sesgo, especialmente con la división por rangos, es el empleo de *procesadores virtuales*. En el enfoque de los **procesadores virtuales** se simula que el número de *procesadores virtuales* es múltiplo del número de procesadores reales. Se puede usar cualquiera de las técnicas de división y de evaluación de consultas que se estudiarán posteriormente en este capítulo, sin embargo asignando las tuplas a los procesadores virtuales en lugar de a los reales. Los procesadores vir-

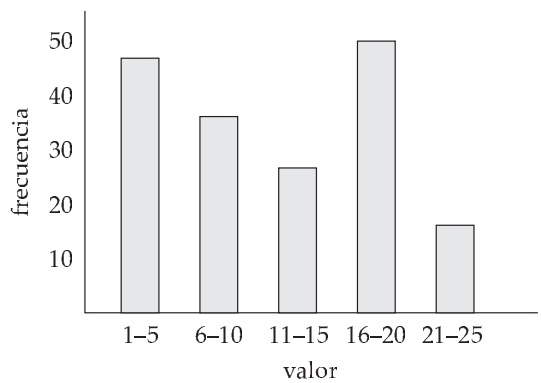


Figura 21.1 Ejemplo de histograma.

tuales, a su vez, se hacen corresponder con los procesadores reales, generalmente mediante una división por turno rotatorio.

La idea es que, incluso si uno de los rangos tuviera muchas más tuplas que los otros debido al sesgo, éstas se podrían repartir entre varios rangos de procesadores virtuales. La asignación por turno rotatorio de los procesadores virtuales a procesadores reales distribuiría el trabajo adicional entre varios procesadores reales, de forma que ningún procesador tuviera que asumir toda la carga.

21.3 Paralelismo entre consultas

En el **paralelismo entre consultas** se ejecutan en paralelo entre sí diferentes consultas o transacciones. La productividad de las transacciones puede aumentarse con esta forma de paralelismo. Sin embargo, el tiempo de respuesta de cada transacción no es menor que si se ejecutara aisladamente. Por ello, la aplicación principal del paralelismo entre consultas es la ampliación de los sistemas de procesamiento de transacciones para permitir un número mayor de transacciones por segundo.

El paralelismo entre consultas es la forma más sencilla de paralelismo que se permite en los sistemas de bases de datos—especialmente en los sistemas paralelos de memoria compartida. Los sistemas de bases de datos diseñados para sistemas con un único procesador pueden utilizarse en arquitecturas paralelas de memoria compartida con pocos cambios o con ninguno, dado que incluso los sistemas secuenciales de bases de datos permiten el procesamiento concurrente. Las transacciones que se habrían realizado de manera concurrente en tiempo compartido en una máquina secuencial se realizan en paralelo en la arquitectura paralela de memoria compartida.

Permitir el paralelismo entre consultas es más complicado en las arquitecturas de disco compartido y sin compartimiento. Los procesadores tienen que realizar algunas tareas, como los bloqueos y el registro histórico, de forma coordinada, y eso exige que se intercambien mensajes. Los sistemas de bases de datos con arquitectura paralela también deben asegurarse de que dos procesadores no actualicen simultáneamente los mismos datos de manera independiente. Además, cuando un procesador accede a los datos o los actualiza, el sistema de bases de datos debe garantizar que tenga su última versión en la memoria intermedia. El problema de asegurar que la versión sea la última disponible se denomina problema de **coherencia de caché**.

Se dispone de varios protocolos para garantizar la coherencia caché; a menudo los protocolos de coherencia de la caché se integran con los de control de concurrencia de modo que se reduce la sobrecarga. Los protocolos de este tipo para sistemas de disco compartido son de la manera siguiente:

1. Antes de cualquier acceso de lectura o de escritura a una página, la transacción la bloquea en modo compartido o exclusivo, según corresponda. Inmediatamente después de obtener el bloqueo compartido o exclusivo de la página, la transacción lee también su copia más reciente del disco compartido.

2. Antes de que una transacción libere el bloqueo exclusivo de una página, la traslada al disco compartido; posteriormente libera el bloqueo.

Este protocolo garantiza que, cuando una transacción establece un bloqueo compartido o exclusivo sobre una página, obtenga la copia correcta de la página.

Otros protocolos más complejos evitan la lectura y escritura reiteradas del disco exigidas por el protocolo anterior. Estos protocolos no escriben las páginas en el disco cuando se liberan los bloqueos exclusivos. Cuando se obtiene un bloqueo compartido o exclusivo, si la versión más reciente de la página se halla en la memoria intermedia de algún procesador, se obtiene de allí. Hay que diseñar los protocolos para tratar peticiones concurrentes. Los protocolos de disco compartido pueden extenderse a las arquitecturas sin compartimiento mediante este esquema. Cada página tiene un **procesador local** P_i y se almacena en el disco D_i. Cuando otros procesadores desean leer la página o escribir en ella, envían las peticiones a su procesador local P_i, dado que no pueden comunicarse directamente con el disco. Las otras acciones son iguales que en los protocolos de disco compartido.

Los sistemas Oracle y Oracle Rdb son ejemplos de sistemas paralelos de bases de datos de disco compartido que permiten el paralelismo entre consultas.

21.4 Paralelismo en consultas

El **paralelismo en consultas** se refiere a la ejecución en paralelo de una única consulta en varios procesadores y discos. El empleo del paralelismo en consultas es importante para acelerar las consultas de ejecución prolongada. El paralelismo entre consultas no ayuda en esta tarea, dado que cada consulta se ejecuta de manera secuencial.

Para ilustrar la evaluación en paralelo de una consulta, considérese una que exija que se ordene una relación. Supóngase que la relación se ha dividido entre varios discos mediante la división por rangos basada en algún atributo y que se solicita la ordenación basada en el atributo de división. La operación de ordenación se puede implementar ordenando cada partición en paralelo y luego se concatenan las particiones ordenadas para obtener la relación ordenada final.

Por tanto, se pueden hacer paralelas las consultas haciendo paralelas las operaciones que las forman. Hay otra fuente de paralelismo para la evaluación de las consultas: el *árbol de operadores* de la consulta puede contener varias operaciones. La evaluación del árbol de operadores se puede hacer paralela evaluando en paralelo las operaciones que no tengan ninguna dependencia entre sí. Además, como se mencionó en el Capítulo 13, puede que se logre encauzar el resultado de una operación hacia otra. Las dos operaciones pueden ejecutarse en paralelo en procesadores separados, uno que genera el resultado que consume el otro, incluso simultáneamente con su generación.

En resumen, hay dos maneras de ejecutar en paralelo una sola consulta:

- **Paralelismo en operaciones**. Se puede acelerar el procesamiento de la consulta haciendo paralela la ejecución de cada una de sus operaciones individuales ordenación, selección, proyección y reunión. El paralelismo en operaciones se considera en el Apartado 21.5.
- **Paralelismo entre operaciones**. Se puede acelerar el procesamiento de la consulta ejecutando en paralelo las diferentes operaciones de las expresiones de las consultas. Esta forma de paralelismo se considera en el Apartado 21.6.

Las dos formas de paralelismo son complementarias y pueden utilizarse simultáneamente en una misma consulta. Dado que el número de operaciones de una consulta típica es pequeño comparado con el número de tuplas procesado por cada operación, la primera modalidad puede adaptarse mejor a un aumento del paralelismo. Sin embargo, con el número relativamente pequeño de procesadores de los sistemas paralelos típicos de hoy en día, ambas formas de paralelismo son importantes.

En la siguiente discusión sobre la paralelización de las consultas se da por supuesto que éstas son **sólo de lectura**. La elección de los algoritmos para la evaluación de las consultas en paralelo depende de la arquitectura de la máquina. En lugar de presentar por separado los algoritmos para cada arquitectura se utilizará en la descripción un modelo de arquitectura sin compartimiento. Por tanto, se describirá explícitamente el momento en que se deben transferir los datos de un procesador a otro. Este modelo se puede simular con facilidad utilizando las otras arquitecturas, dado que la transferencia de los datos puede realizarse mediante la memoria compartida en las arquitecturas de memoria compartida y mediante los discos compartidos en las arquitecturas de discos compartidos. Por tanto, los algoritmos para las arquitecturas sin compartimiento también pueden utilizarse en las demás arquitecturas. Ocasionalmente se menciona la manera en que se pueden optimizar aún más los algoritmos para los sistemas de memoria o de discos compartidos.

Para simplificar la exposición de los algoritmos se supone que existen n procesadores, $P_0, P_1, \ldots, P_{n-1}$ y n discos, $D_0, D_1, \ldots, D_{n-1}$, donde el disco D_i está asociado con el procesador P_i. Los sistemas reales pueden tener varios discos por cada procesador. No es difícil extender los algoritmos para que permitan varios discos por procesador: basta con permitir que D_i sea un conjunto de discos. Sin embargo, en aras de la sencillez de la exposición, aquí se supondrá que D_i es un solo disco.

21.5 Paralelismo en operaciones

Dado que las operaciones relacionales trabajan con relaciones que contienen grandes conjuntos de tuplas, las operaciones se pueden paralelizar ejecutándolas sobre subconjuntos diferentes de las relaciones en paralelo. Dado que el número de tuplas de cada relación puede ser grande, el grado de paralelismo

es potencialmente enorme. Por tanto, el paralelismo en operaciones resulta natural en los sistemas de bases de datos. Las versiones paralelas de algunas operaciones relacionales frecuentes se estudiarán en los Apartados 21.5.1 a 21.5.3.

21.5.1 Ordenación paralela

Supóngase que se desea ordenar una relación que reside en n discos, $D_0, D_1, \ldots, D_{n-1}$. Si la relación se ha dividido por rangos basándose en los atributos por los que se va a ordenar, entonces, como se indicó en el Apartado 21.2.2, se puede ordenar cada partición por separado y concatenar los resultados para obtener la relación completa ordenada. Dado que las tuplas se hallan divididas en n discos, el tiempo necesario para leer la relación completa se reduce gracias al acceso en paralelo.

Si la relación se ha dividido siguiendo algún otro método, se puede ordenar de una de estas dos maneras:

1. Se puede dividir en rangos de acuerdo con los atributos de ordenación y luego ordenar cada partición por separado.
2. Se puede utilizar una versión paralela del algoritmo externo de ordenación–mezcla.

21.5.1.1 Ordenación con división por rangos

La **ordenación con división por rangos** tiene dos etapas: primero, se divide por rangos la relación y, después, se ordena cada una de las particiones. Cuando la relación se ordena mediante división por rangos, no hace falta realizar la división en los mismos procesadores o discos en los que se almacena la relación. Supóngase que se escogen los procesadores $P_0, P_1, \ldots, P_m$, donde $m < n$, para ordenar la relación. Esta operación se divide en dos fases:

1. Redistribuir las tuplas de la relación utilizando una estrategia de división por rangos, de manera que todas las tuplas que se hallen dentro del rango i-ésimo se envíen al procesador P_i, que almacena temporalmente la relación en el disco D_i.

   Para implementar en paralelo la división por rangos cada procesador lee las tuplas de su disco y las envía al procesador de destino. Cada procesador $P_0, P_1, \ldots, P_m$ también recibe las tuplas correspondientes a su partición y las almacena localmente. Esta fase necesita una sobrecarga en la E/S de disco y en las comunicaciones.

2. Cada uno de los procesadores ordena localmente su partición de la relación sin interactuar con los demás procesadores. Cada procesador ejecuta la misma operación—por ejemplo, ordenar—sobre un conjunto de datos diferente (la ejecución de la misma acción en paralelo sobre conjuntos diferentes de datos se denomina **paralelismo de datos**).

   La operación final de mezcla es trivial, ya que la división por rangos de la primera etapa asegura que, para $1 \leq i < j \leq m$, los valores de la clave del procesador P_i son todos menores que los de P_j.

La división por rangos se debe llevar a cabo empleando un buen vector de división por rangos, de manera que cada partición tenga aproximadamente el mismo número de tuplas. Los procesadores virtuales también se pueden utilizar para reducir el sesgo.

21.5.1.2 Ordenación y mezcla externas paralelas

La **ordenación y mezcla externas paralelas** son una alternativa a la división por rangos. Supóngase que la relación ya se ha dividido entre los discos $D_0, D_1, \ldots, D_{n-1}$ (no importa la manera en que se haya dividido la relación). La ordenación y mezcla externas paralelas funcionan de la siguiente manera:

1. Cada procesador P_i ordena localmente los datos del disco D_i.
2. El sistema mezcla las partes ordenadas por cada procesador para obtener el resultado ordenado final.

La mezcla de las partes ordenadas del paso 2 puede paralelizarse mediante esta secuencia de acciones:

1. El sistema divide en rangos las particiones ordenadas en cada procesador P_i (utilizando el mismo vector de división) entre los procesadores $P_0, P_1, \ldots, P_{m-1}$. Envía las tuplas de acuerdo con el orden establecido, por lo que cada procesador recibe las tuplas en corrientes ordenadas.
2. Cada procesador P_i realiza una mezcla de las corrientes según las recibe para obtener una sola parte ordenada.
3. Las partes ordenadas de los procesadores $P_0, P_1, \ldots, P_{m-1}$ se concatenan para obtener el resultado final.

Como ya se ha descrito, esta secuencia de acciones da lugar a una variedad interesante del **sesgo de ejecución**, ya que, al principio cada procesador envía todos los bloques de la división 0 a P_0, después cada procesador envía todos los bloques de la partición 1 a P_1, etc. Así, mientras que el envío se produce en paralelo, la recepción de las tuplas es secuencial: primero sólo P_0 recibe tuplas, luego sólo lo hace P_1, y así, sucesivamente. Para evitar este problema, cada procesador envía repetidamente un bloque de datos a cada partición. En otras palabras, cada procesador envía el primer bloque de cada partición, luego envía el segundo bloque de cada partición, etc. En consecuencia, todos los procesadores reciben los datos en paralelo.

Algunas máquinas, como las de la serie DBC de Teradata, utilizan hardware especializado en realizar mezclas. La red de interconexión Y-net de las máquinas DBC de Teradata puede mezclar los resultados de varios procesadores para ofrecer un solo resultado ordenado.

21.5.2 Reunión paralela

La operación reunión exige que el sistema compare pares de tuplas para ver si satisfacen la condición de reunión; si la cumplen, añade el par al resultado de la reunión. Los algoritmos de reunión paralela intentan repartir entre varios procesadores los pares que hay que comparar. Cada procesador procesa luego localmente parte de la reunión. Después, el sistema reúne los resultados de cada procesador para producir el resultado final.

21.5.2.1 Reunión por división

Para ciertos tipos de reuniones, como las equirreuniones y las reuniones naturales, es posible *dividir* las dos relaciones de entrada entre los procesadores y procesar localmente la reunión en cada uno de ellos. Supóngase que se utilizan n procesadores y que las relaciones que hay que reunir son r y s. La **reunión por división** funciona de esta forma: el sistema divide las relaciones en n particiones, denominadas $r_0, r_1, \ldots, r_{n-1}$ y $s_0, s_1, \ldots, s_{n-1}$. Envía las particiones r_i y s_i al procesador P_i, donde la reunión se procesa localmente.

La técnica anterior sólo funciona correctamente si la reunión es una equirreunión (por ejemplo, $r \bowtie_{r.A=s.B} s$) y se dividen r y s utilizando la misma función de división para sus atributos de reunión. La idea de la división es exactamente la misma que subyace a la fase de división de la reunión por asociación. Sin embargo, en la reunión por división hay dos maneras diferentes de dividir r y s:

- División por rangos de los atributos de reunión.
- División por asociación de los atributos de reunión.

En ambos casos se debe utilizar la misma función de división para las dos relaciones. Para la división por rangos, el mismo vector de división. En el caso de la división por asociación, la misma función de asociación. La Figura 21.2 muestra la división utilizada en una reunión por división paralela.

Una vez divididas las relaciones, se puede utilizar localmente cualquier técnica de reunión en cada procesador P_i para calcular la reunión de r_i y s_i. Por ejemplo, se puede emplear la reunión por asociación, por mezcla o con bucles anidados. Por tanto, se puede utilizar la división para paralelizar cualquier técnica de reunión.

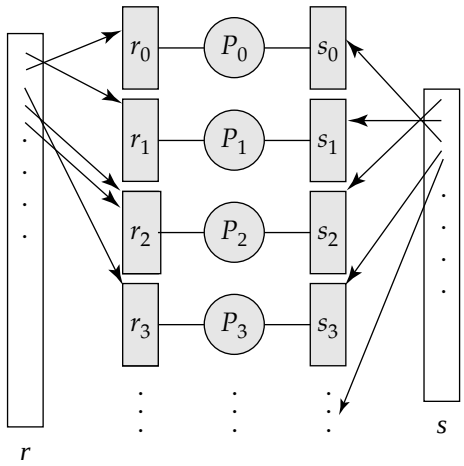


Figura 21.2 Reunión por división paralela.

Si alguna de las relaciones r y s, o las dos, ya están divididas basándose en los atributos de reunión (mediante división por asociación o por rangos) el trabajo necesario para la división se reduce mucho. Si las relaciones no están divididas, o lo están de acuerdo con atributos distintos de los de la reunión, hay que volver a dividir las tuplas. Cada procesador P_i lee las tuplas del disco *Di*, procesa para cada tupla t la partición j a la que pertenece *i* y la envía al procesador P_j. El procesador P_j almacena las tuplas en el disco D_j.

Se puede optimizar el algoritmo de reunión utilizado localmente en cada procesador para reducir E/S guardando en la memoria intermedia algunas de las tuplas, en lugar de escribirlas en el disco. Estas optimizaciones se describen en el Apartado 21.5.2.3.

El sesgo representa un problema especial cuando se utiliza la división por rangos, dado que un vector de división que divida una relación de la reunión en particiones de igual tamaño puede dividir las demás relaciones en particiones de tamaño muy variable. El vector de división debe ser tal que $|r_i| + |s_i|$ (es decir, la suma de los tamaños de r_i y s_i) sea aproximadamente igual para todo $i = 0, 1, \ldots, n-1$. Con una buena función de asociación, la división por asociación probablemente tenga menos sesgo, excepto cuando haya muchas tuplas con los mismos valores de los atributos de reunión.

21.5.2.2 Reunión con fragmentos y réplicas

La división no es aplicable a todos los tipos de reuniones. Por ejemplo, si la condición de reunión es una desigualdad, como $r \bowtie_{r.a<s.b} s$, es posible que todas las tuplas de r se reúnan con alguna tupla de s (y viceversa). Por tanto, puede que no haya un medio sencillo de dividir r y s de modo que las tuplas de la partición r_i sólo se reúnan con tuplas de la partición s_i.

Esas reuniones pueden paralelizarse utilizando una técnica denominada *fragmentos y réplicas*. En primer lugar se considerará un caso especial de fragmentos y réplicas—la **reunión con fragmentos y réplicas asimétricos**—que funciona de la manera siguiente:

1. El sistema divide una de las relaciones (por ejemplo, r). Se puede utilizar en r cualquier técnica de división, incluida la división por turno rotatorio.
2. El sistema replica la otra relación, s, en todos los procesadores.
3. El procesador P_i procesa localmente la reunión de r i con toda s, utilizando cualquier técnica de reunión.

El esquema asimétrico de fragmentos y réplicas se muestra en la Figura 21.3a. Si r ya está almacenada por particiones no es necesario dividirla más en la primera fase. Todo lo que hace falta es replicar s en todos los procesadores.

El caso general de **reunión con fragmentos y réplicas** se muestra en la Figura 21.3b; funciona de la manera siguiente. El sistema divide la relación r en n particiones, $r_0, r_1, \ldots, r_{n-1}$ y s en otras m, $s_0, s_1, \ldots, s_{m-1}$. Al igual que antes, se puede utilizar cualquier técnica de división para r y para s. No es necesario que los valores de m y de n sean iguales, pero deben escogerse de modo que haya al menos $m * n$ procesadores. El esquema asimétrico de fragmentos y réplicas sólo es un caso especial de fragmentos y réplicas, en el que $m = 1$. El esquema de fragmentos y réplicas reduce el tamaño de las relaciones en cada procesador en comparación con el caso asimétrico.

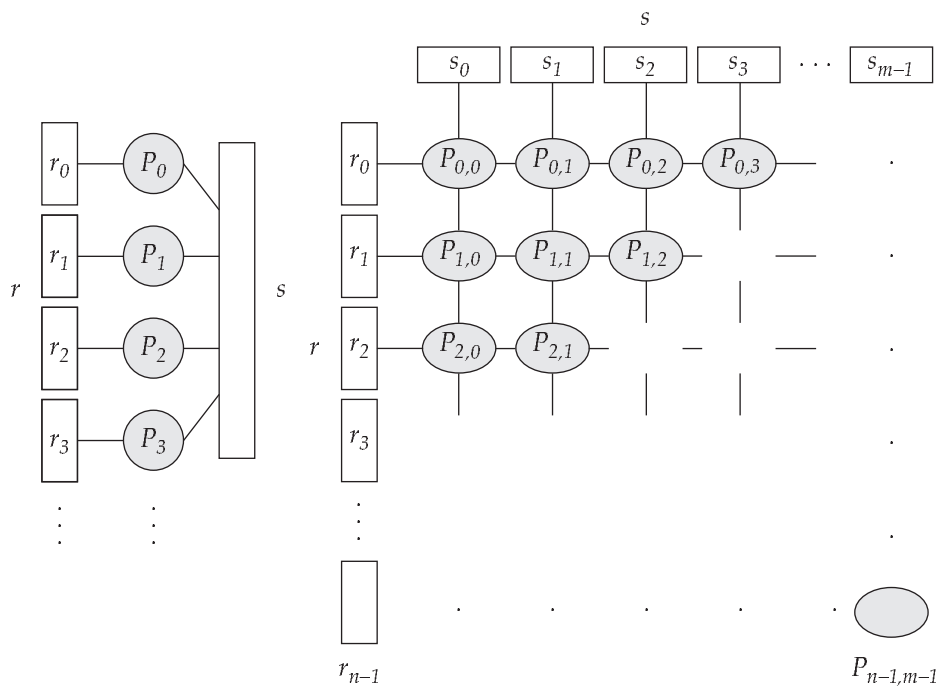
Sean los procesadores $P_{0,0}, P_{0,1}, \ldots, P_{0,m-1}, P_{1,0}, \ldots, P_{n-1,m-1}$. El procesador $P_{i,j}$ procesa la reunión de r_i con s_j. Cada procesador debe obtener las tuplas de las particiones sobre las que trabaja. Para ello, el sistema replica r_i en los procesadores $P_{i,0}, P_{i,1}, \ldots, P_{i,m-1}$ (que forman una fila en la Figura 21.3b) y s_i en los procesadores $P_{0,i}, P_{1,i}, \ldots, P_{n-1,i}$ (que forman una columna en la Figura 21.3b). Se puede utilizar la técnica de reunión que se prefiera en cada procesador $P_{i,j}$.

El esquema de fragmentos y réplicas funciona con cualquier condición de reunión, ya que cada tupla de r puede compararse con cada una de las de s. Por tanto, puede utilizarse cuando no pueda emplearse la división.

El esquema de fragmentos y réplicas suele tener un mayor coste que la división cuando ambas relaciones son aproximadamente del mismo tamaño, ya que hay que replicar, como mínimo, una de las relaciones. Sin embargo, si una de las relaciones (por ejemplo, s) es pequeña, puede resultar más barato replicar s en todos los procesadores que volver a dividir r y s basándose en los atributos de reunión. En tal caso, el esquema asimétrico de fragmentos y réplicas es preferible, aunque pueda utilizarse la división.

21.5.2.3 Reunión por asociación dividida en paralelo

La reunión por asociación dividida del Apartado 13.5.5 puede hacerse en paralelo. Supóngase que se tienen n procesadores, P_0, $P_1, \ldots, P_{n-1}$ y dos relaciones, r y s, que se encuentran divididas entre varios discos. Recuérdese del Apartado 12.6 que se debe escoger la relación de menor tamaño como relación



(a) Fragmentos y réplicas asimétricos

(b) Fragmentos y réplicas

Figura 21.3 Esquemas de fragmentos y réplicas.

de construcción. Si el tamaño de s es menor que el de r, el algoritmo de reunión por asociación paralela procede de la manera siguiente:

1. Se escoge una función de asociación (por ejemplo, h_1) que tome el valor del atributo de reunión de cada tupla de r y de s y asigne esa tupla a uno de los n procesadores. Sean r i las tuplas de la relación r asignadas al procesador P_i; análogamente, sean s_i las tuplas de la relación s asignadas al procesador P_i. Cada procesador P_i lee las tuplas de s que están en el disco D_i y envía cada tupla al procesador apropiado basándose en la función de asociación h_1.
2. A medida que el procesador de destino P_i recibe las tuplas de s_i, las vuelve a dividir de acuerdo con otra función de asociación, h_2, que utiliza para procesar localmente la reunión por asociación. La división en esta etapa es exactamente la misma que en la fase de división del algoritmo secuencial de reunión por asociación. Cada procesador P_i ejecuta esta fase independientemente de los demás procesadores.
3. Una vez que se han distribuido las tuplas de s, el sistema redistribuye la relación de mayor tamaño, r, entre los n procesadores de acuerdo con la función de asociación h_1 del mismo modo que anteriormente. A medida que recibe cada tupla, el procesador de destino la divide según la función h_2, igual que la relación de exploración se divide en el algoritmo secuencial de reunión por asociación.
4. Cada procesador P_i ejecuta las fases de construcción y exploración del algoritmo de reunión por asociación en las particiones locales r_i y s_i para generar una división del resultado final de la reunión por asociación.

La reunión por asociación realizada en cada procesador es independiente de las realizadas en los demás, y recibir las tuplas de r_i y de s_i es parecido a leerlas del disco. Por tanto, también se puede aplicar cualquiera de las optimizaciones de la reunión por asociación descritas en el Capítulo 13 al caso paralelo. En concreto, se puede utilizar el algoritmo híbrido de reunión por asociación para almacenar en caché algunas de las tuplas entrantes en la memoria, y evitar así el coste de escribirlas y volver a leerlas.

21.5.2.4 Reuniones con bucles anidados en paralelo

Para ilustrar el empleo de la paralelización basada en fragmentos y réplicas se considera el caso en que la relación s sea mucho menor que r. Supóngase que la relación r se almacena por división; el atributo de acuerdo con el cual se divide es irrelevante. Supóngase también que hay un índice basado en un atributo de reunión de la relación r en cada una de las particiones de la relación r.

Se utiliza el esquema asimétrico de fragmentos y réplicas mientras se replica la relación s y se emplea la división ya existente de la relación r. Cada procesador P_j en que se almacena una partición de la relación s lee las tuplas de la relación s almacenadas en D_j y las replica en el resto de procesadores P_i. Al final de esta fase, la relación s está replicada en todos los puntos de almacenamiento de las tuplas de la relación r.

A continuación cada procesador P_i realiza una reunión indexada con bucles anidados de la relación s con la partición i-ésima de la relación r. Se puede solapar la reunión indexada con bucles anidados con la distribución de las tuplas de la relación s para reducir el coste de escribir en el disco las tuplas de la relación s y volver a leerlas. Sin embargo, la réplica de la relación s debe sincronizarse con la reunión para que haya espacio suficiente en las memorias intermedias de la memoria principal de cada procesador P_i para albergar las tuplas de la relación s que se hayan recibido, pero que todavía no se hayan utilizado en la reunión.

21.5.3 Otras operaciones relacionales

También se puede realizar en paralelo la evaluación de otras operaciones relacionales:

- **Selección**. Sea la selección $\sigma_\theta(r)$. Considérese en primer lugar el caso en el que θ es de la forma $a_i = v$, donde a_i es un atributo y v es un valor. Si la relación r se divide de acuerdo con a_i, la

selección se lleva a cabo en un solo procesador. Si θ es de la forma $l \leq a_i \leq u$ (es decir, que θ es una selección de rango) y la relación se ha dividido por rangos según a_i, entonces la selección se lleva a cabo en cada procesador cuya partición se solape con el rango de valores especificado. En el resto de los casos, la selección se lleva a cabo en todos los procesadores en paralelo.

- **Eliminación de duplicados**. Los duplicados se pueden eliminar por ordenación; para ellos puede utilizarse cualquiera de las técnicas de ordenación en paralelo, optimizada para eliminar los duplicados durante la ordenación en cuanto aparezcan. También se puede paralelizar la eliminación de duplicados dividiendo las tuplas (por rangos o por asociación) y eliminado los duplicados localmente en cada procesador.
- **Proyección**. Se puede llevar a cabo la proyección sin eliminación de duplicados a medida que se leen en paralelo las tuplas del disco. Si se van a eliminar los duplicados, se puede utilizar cualquiera de las técnicas que se acaban de describir.
- **Agregación**. Considérese una operación de agregación. Se puede paralelizar dividiendo la relación de acuerdo con los atributos de agrupación y procesando localmente los valores de agregación en cada procesador. Se puede dividir por rangos o por asociación. Si la relación ya está dividida según los atributos de agrupación, se puede omitir la primera fase.

  Se puede reducir el coste de transferir las tuplas durante la división calculando parcialmente los valores de agregación antes de la división, al menos para las funciones de agregación utilizadas habitualmente. Considérese una operación de agregación sobre la relación r, que utiliza la función de agregación **sum** en el atributo B y la agrupación basada en el atributo A. El sistema puede llevar a cabo la operación en cada procesador P_i sobre las tuplas de r almacenadas en el disco D_i. Este cálculo da lugar en cada procesador a tuplas con sumas parciales; hay una tupla en P_i para cada valor del atributo A presente en las tuplas de r almacenadas en D_i. El sistema divide el resultado de la agregación local de acuerdo con el atributo de agrupación A y vuelve a llevar a cabo la agregación (sobre las tuplas con sumas parciales) en cada procesador P_i para obtener el resultado final.

  A consecuencia de esta optimización no es necesario enviar tantas tuplas a los demás procesadores durante la división. Esta idea puede extenderse fácilmente a las funciones de agregación **min** y **max**. Las extensiones para las funciones de agregación **count** y **avg** se proponen al lector en el Ejercicio 21.10.

La paralelización de otras operaciones se trata en varios ejercicios.

21.5.4 Coste de la evaluación paralela de las operaciones

El paralelismo se obtiene dividiendo la E/S entre varios discos y el trabajo de la CPU entre varios procesadores. Si se logra un reparto así sin sobrecarga y no hay sesgo en el reparto del trabajo, las operaciones en paralelo que utilicen n procesadores tardarán $1/n$ lo que tardarían en un solo procesador. Ya se sabe cómo estimar el coste de operaciones como la reunión o la selección. El coste en tiempo del procesamiento paralelo sería entonces $1/n$ el del procesamiento secuencial de esa operación.

También hay que tener en cuenta los costes siguientes:

- Los **costes iniciales** de comenzar la operación en varios procesadores.
- El **sesgo** en la distribución del trabajo entre los procesadores, con algunos procesadores con mayor número de tuplas que otros.
- La **competencia por los recursos**—como la memoria, los discos y la red de comunicaciones—que dan lugar a retrasos.
- El **coste de construir** el resultado final mediante la transmisión de los resultados parciales desde cada procesador.

El tiempo empleado por una operación en paralelo puede estimarse como

$$T_{\text{part}} + T_{\text{asm}} + \max(T_0, T_1, \ldots, T_{n-1})$$

donde T_{part} es el tiempo necesario para dividir las relaciones, T_{asm} es el tiempo empleado en construir los resultados y T_i el tiempo utilizado por la operación en el procesador P_i. Suponiendo que las tuplas se distribuyen sin sesgo, el número de tuplas enviadas a cada procesador puede estimarse como $1/n$ del número total de tuplas. Ignorando la competencia, el coste T_i de las operaciones en cada procesador P_i, puede estimarse mediante las técnicas descritas en el Capítulo 13.

La estimación precedente es optimista, dado que el sesgo es habitual. Aunque dividir una sola consulta en varias fases paralelas reduce el tamaño de la fase promedio, es el tiempo de procesamiento del paso más lento el que determina el tiempo empleado en procesar la consulta en su conjunto. Una evaluación en paralelo dividida, por ejemplo, no puede ser más rápida que la más lenta de sus ejecuciones en paralelo. Por tanto, cualquier sesgo en la distribución del trabajo entre los procesadores afecta mucho al rendimiento.

El problema del sesgo de la división está íntimamente relacionado con el del desbordamiento de las divisiones en las reuniones secuenciales por asociación (Capítulo 13). Se puede utilizar la resolución del desbordamiento y las técnicas de evitación desarrolladas para las reuniones por asociación para tratar el sesgo cuando se utilice la división por asociación. Se puede utilizar la división equilibrada por rangos y la división con procesadores virtuales para minimizar el sesgo debido a la división por rangos, como en el Apartado 21.2.3.

21.6 Paralelismo entre operaciones

Existen dos formas de paralelismo entre operaciones: el paralelismo de encauzamiento y el paralelismo independiente.

21.6.1 Paralelismo de encauzamiento

Como se estudió en el Capítulo 13, el encauzamiento supone una importante fuente de economía de cálculo para el procesamiento de las consultas de bases de datos. Hay que recordar que, en el encauzamiento, las tuplas resultado de una operación, A, las consume una segunda operación, B, incluso antes de que la primera operación haya producido todo el conjunto de tuplas de su resultado. La ventaja principal de la ejecución encauzada de las evaluaciones secuenciales es que se puede ejecutar una secuencia de operaciones de ese tipo sin escribir en el disco ninguno de los resultados intermedios.

Los sistemas paralelos utilizan el encauzamiento principalmente por la misma razón que los sistemas secuenciales. Sin embargo, el encauzamiento también es una fuente de paralelismo, del mismo modo que el encauzamiento de instrucciones se utiliza como fuente de paralelismo en el diseño de hardware. Es posible ejecutar simultáneamente A y B en procesadores diferentes de modo que B consuma las tuplas en paralelo con su producción por A. Esta forma de paralelismo se denomina **paralelismo de encauzamiento**.

Considérese una reunión de cuatro relaciones:

$$r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$$

Se puede configurar un cauce que permita que las tres reuniones se calculen en paralelo. Supóngase que se asigna al procesador P_1 el cálculo de$temp_1 \leftarrow r_1 \bowtie r_2$ y al procesador P_2 el cálculo de $r_3 \bowtie temp_1$. A medida que P_1 procesa las tuplas de $r_1 \bowtie r_2$, las pone a disposición del procesador P_2. Por tanto, P_2 tiene a su disposición algunas de las tuplas de $r_1 \bowtie r_2$ antes de que P_1 haya finalizado su cálculo. P_2 puede utilizar esas tuplas que están disponibles para comenzar el cálculo de $temp_1 \bowtie r_3$, incluso antes de que P_1 haya calculado completamente $r_1 \bowtie r_2$. Análogamente, a medida que P_2 procesa las tuplas de $(r_1 \bowtie r_2) \bowtie r_3$, las pone a disposición de P_3, que calcula su reunión con r_4.

El paralelismo encauzado resulta útil con un número pequeño de procesadores, pero no puede extenderse bien. En primer lugar, las cadenas del cauce no suelen lograr la longitud suficiente para proporcionar un alto grado de paralelismo. En segundo lugar, no es posible encauzar los operadores relacionales que no producen resultados hasta que han accedido a todos los datos, como la operación diferencia de conjuntos. En tercer lugar, sólo se obtiene una aceleración marginal en el caso frecuente de que el coste de ejecución de un operador sea mucho mayor que el de los demás.

Por consiguiente, cuando el grado de paralelismo es elevado, el encauzamiento es una fuente de paralelismo menos importante que la división. El verdadero motivo del empleo del encauzamiento es que las ejecuciones encauzadas pueden evitar escribir en el disco los resultados intermedios.

21.6.2 Paralelismo independiente

Las operaciones en las expresiones de las consultas que son independientes entre sí pueden ejecutarse en paralelo. Esta forma de paralelismo se denomina **paralelismo independiente**.

Considérese la reunión $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$. Evidentemente, se puede procesar $temp_1 \leftarrow r_1 \bowtie r_2$ en paralelo con $temp_2 \leftarrow r_3 \bowtie r_4$. Cuando se completen esos dos cálculos se calculará

$$temp_1 \bowtie temp_2$$

Para obtener más paralelismo se pueden encauzar las tuplas de $temp_1$ y $temp_2$ al cálculo de $temp_1 \bowtie temp_2$, que se ejecuta mediante una reunión encauzada (Apartado 13.7.2.2).

Como ocurre con el paralelismo encauzado, el paralelismo independiente no proporciona un alto grado de paralelismo y resulta menos útil en sistemas con un elevado nivel de paralelismo, aunque es útil con un grado menor de paralelismo.

21.6.3 Optimización de consultas

Los optimizadores de consultas son responsables de gran parte del éxito de la tecnología relacional. Recuérdese que los optimizadores de consultas toman una consulta y hallan el plan de ejecución más económico de entre todos los que proporcionan la misma respuesta.

Los optimizadores de consultas para la evaluación en paralelo de las consultas son más complicados que los correspondientes a la evaluación secuencial de consultas. En primer lugar, los modelos de costes son más complicados, ya que hay que tener en cuenta los costes de división y aspectos como el sesgo y la competencia por los recursos. Más importante aún es la manera de paralelizar las consultas. Supóngase que, de algún modo, se ha escogido una expresión (de entre las equivalentes a la consulta) para utilizarla para evaluar la consulta. La expresión puede representarse mediante un árbol de operadores, como en el Apartado 13.1.

Para evaluar un árbol de operadores en un sistema paralelo hay que tomar las decisiones siguientes:

- El modo de paralelizar cada operación y el número de procesadores que se emplearán para ello.
- Las operaciones que se encauzarán entre los diferentes procesadores, las operaciones que se ejecutarán independientemente en paralelo y las que lo harán secuencialmente, una tras otra.

Estas decisiones constituyen la tarea de **planificación** del árbol de ejecución.

Determinar los recursos de cada clase—como procesadores, discos y memoria—que se deben asignar a cada operación del árbol es otro aspecto del problema de la optimización. Por ejemplo, puede que parezca conveniente utilizar la máxima cantidad disponible de paralelismo, pero es buena idea no ejecutar ciertas operaciones en paralelo. Las operaciones cuyos requisitos de cálculo sean significativamente menores que la sobrecarga de comunicaciones deben agruparse con una de sus vecinas. En caso contrario, la ventaja del paralelismo se anula debido a la sobrecarga en las comunicaciones.

Un problema es que los cauces largos no se prestan a un buen empleo de los recursos. A menos que las operaciones tengan grano grueso, puede que la operación final del encauzamiento espere mucho tiempo para obtener sus datos, mientras retiene recursos preciosos, como la memoria. Por tanto, deben evitarse los cauces largos.

El número de planes de evaluación en paralelo entre los que se puede escoger es mucho mayor que el de los secuenciales. Optimizar las consultas en paralelo teniendo en cuenta todas las alternativas es, por tanto, mucho más costoso que optimizar las consultas secuenciales. Por tanto, se suelen adoptar enfoques heurísticos para reducir el número de planes de ejecución en paralelo que se deben tomar en consideración. A continuación se describen dos heurísticas muy conocidas.

La primera heurística es considerar únicamente los planes de evaluación que paralelizan todas las operaciones de todos los procesadores y que no utilizan encauzamiento. Este enfoque se utiliza en las máquinas de la serie DBC de Teradata. Buscar el mejor plan de ejecución de este tipo es parecido a

realizar la optimización de consultas en sistemas secuenciales. Las principales diferencias radican en la manera de llevar a cabo la división y en la fórmula de estimación de costes utilizada.

La segunda heurística es escoger el plan de evaluación secuencial más eficiente y luego paralelizar sus operaciones. El sistema paralelo de bases de datos Volcano ha popularizado el modelo de paralelización denominado **intercambio de operadores**. Este modelo utiliza implementaciones ya existentes de las operaciones, que actúan sobre copias locales de los datos, acopladas con una operación de intercambio que traslada los datos entre los diferentes procesadores. Se pueden introducir los operadores de intercambio en el plan de evaluación para transformarlo en un plan de evaluación en paralelo.

Otra dimensión más de la optimización es el diseño de la organización del almacenamiento físico para acelerar las consultas. La organización física óptima es diferente para las diferentes consultas. El administrador de la base de datos debe escoger la organización física que considere adecuada para la combinación esperada de consultas a la base de datos. Por tanto, el área de la optimización de consultas en paralelo es compleja y sigue siendo un campo de investigación activa.

21.7 Diseño de sistemas paralelos

Hasta ahora, este capítulo se ha concentrado en la paralelización del almacenamiento de los datos y del procesamiento de las consultas. Dado que los sistemas paralelos de bases de datos de gran escala se utilizan principalmente para almacenar grandes volúmenes de datos y para procesar consultas de ayuda a las decisiones basadas en esos datos, estos aspectos son los más importantes de los sistemas paralelos de bases de datos. La carga de los datos en paralelo desde fuentes externas es un requisito importante si se van a tratar grandes volúmenes de datos entrantes.

Los grandes sistemas paralelos de bases de datos deben abordar también los siguientes aspectos de disponibilidad:

- El poder de recuperación frente a fallos de algunos procesadores o discos.
- La reorganización interactiva de los datos y los cambios interactivos de los esquemas.

Estos temas se tratan a continuación.

Con un gran número de procesadores y de discos la probabilidad de que, al menos, un procesador o un disco funcionen mal es significativamente mayor que en sistemas con un único procesador y un solo disco. Un sistema paralelo mal diseñado dejará de funcionar si cualquier componente (procesador o disco) falla. Suponiendo que la probabilidad de fallo de cada procesador o disco sea pequeña, la probabilidad de fallo del sistema aumenta linealmente con el número de procesadores y de discos. Si un solo procesador o disco falla una vez cada cinco años, un sistema con cien procesadores tendrá un fallo cada dieciocho días.

Por tanto, los sistemas paralelos de bases de datos de gran escala, como las máquinas Himalaya de Compaq, las de Teradata y las XPS de Informix (ahora una división de IBM) se diseñan para operar aunque falle un procesador o un disco. Los datos se replican en, al menos, dos procesadores. Si falla un procesador, se puede seguir accediendo desde los demás procesadores a los datos que almacenaba. El sistema hace un seguimiento de los procesadores averiados y distribuye el trabajo entre los que funcionan. Las peticiones de datos que estaban almacenados en el emplazamiento estropeado se desvían automáticamente a los emplazamientos de respaldo, que almacenan una réplica. Si todos los datos del procesador A se replican en un solo procesador B, B tendrá que procesar todas las peticiones formuladas a A, así como las propias, y eso hará que B se transforme en un cuello de botella. Por tanto, las réplicas de los datos de cada procesador se dividen entre varios procesadores.

Cuando se manejan grandes volúmenes de datos (del orden de terabytes), las operaciones sencillas, como la creación de índices, y los cambios en los esquemas, como añadir una columna a una relación, pueden tardar mucho tiempo—quizás horas o, incluso, días. Por tanto, no resulta aceptable que los sistemas de bases de datos no estén disponibles mientras se llevan a cabo esas operaciones. Muchos sistemas paralelos de bases de datos, como los sistemas Himalaya de Compaq, permiten que tales operaciones se lleven a cabo **interactivamente**, es decir, mientras el sistema ejecuta otras transacciones.

Considérese, por ejemplo, la **generación interactiva de índices**. Los sistemas que tienen esta característica permiten que se realicen inserciones, borrados y actualizaciones en una relación aunque se esté generando un índice de la misma. La operación de generación de índices, por tanto, no puede bloquear toda la relación en modo compartido, como habría hecho en caso contrario. Por el contrario, el proceso hace un seguimiento de las actualizaciones que tienen lugar mientras está activo e incorpora los cambios en el índice que se está generando.

21.8 Resumen

- Las bases de datos en paralelo han logrado una aceptación comercial significativa en los últimos veinte años.
- En el paralelismo de E/S las relaciones se dividen entre los discos disponibles para poder recuperarlas más rápidamente. Tres técnicas de división utilizadas frecuentemente son la división por turno rotatorio, la división por asociación y la división por rangos.
- El sesgo es un problema importante, especialmente con grados elevados de paralelismo. Los vectores de división equilibrados, que utilizan histogramas, y la división con procesadores virtuales son algunas técnicas usadas para reducir el sesgo.
- En el paralelismo entre consultas se ejecutan concurrentemente diferentes consultas para aumentar la productividad.
- El paralelismo en las consultas intenta reducir el coste de ejecución de las consultas. Existen dos tipos de paralelismo en las consultas: el paralelismo en operaciones y el paralelismo entre operaciones.
- El paralelismo en operaciones se utiliza para ejecutar operaciones relacionales, como las ordenaciones y las reuniones, en paralelo. El paralelismo en operaciones es algo natural en las operaciones relacionales, ya que están orientadas a conjuntos.
- Existen dos enfoques básicos en la paralelización de las operaciones binarias como las reuniones.
  - En el paralelismo de divisiones las relaciones se dividen en varias partes y las tuplas de r_i sólo se reúnen con las tuplas de s_i. El paralelismo de divisiones sólo se puede usar para las reuniones naturales y para las equirreuniones.
  - En el esquema de fragmentos y réplicas las dos relaciones se dividen y cada división se duplica. En el esquema asimétrico de fragmentos y réplicas una de las relaciones se replica mientras la otra se divide. A diferencia del paralelismo de divisiones, el esquema de fragmentos y réplicas, tanto simétrico como asimétrico, puede utilizarse con cualquier condición de reunión.

  Ambas técnicas de paralelismo pueden utilizarse en combinación con cualquiera de las técnicas de reunión.
- En el paralelismo independiente, las diferentes operaciones que son independientes entre sí se ejecutan en paralelo.
- En el paralelismo encauzado, los procesadores envían los resultados de una operación a otra a medida que los van calculando, sin esperar a que concluya toda la operación.
- La optimización de consultas en las bases de datos paralelas es significativamente más compleja que su equivalente en las bases de datos secuenciales.

Términos de repaso

- Consultas de ayuda a la toma de decisiones.
- Paralelismo de E/S.
- División horizontal.
- Técnicas de división:
  - Turno rotatorio.
  - División por asociación.
  - División por rangos.
- Atributo de división.
- Vector de división.
- Consulta concreta.
- Consulta de rango.
- Sesgo:
  - De ejecución.
  - De los valores de los atributos.
  - De la división.
- Manejo del sesgo.
  - Vector de división por rangos equilibrado.
  - Histograma.
  - Procesadores virtuales.
- Paralelismo entre consultas.
- Coherencia de la caché.
- Paralelismo en consultas.
  - Paralelismo en operaciones.
  - Paralelismo entre operaciones.
- Ordenación paralela.
  - Ordenación con división por rangos
  - Ordenación y mezcla externas paralelas.
- Paralelismo de datos.
- Reunión paralela:
  - Por división.
  - Con fragmentos y réplicas.
  - Con esquema asimétrico de fragmentos y réplicas.
  - Por asociación dividida en paralelo.
  - Con bucles anidados en paralelo.
- Selección paralela.
- Eliminación de duplicados en paralelo.
- Proyección paralela.
- Agregación paralela.
- Coste de la evaluación paralela.
- Paralelismo entre operaciones:
  - De encauzamiento.
  - Independiente.
- Optimización de consultas.
- Planificación.
- Modelo del operador de intercambio.
- Diseño de sistemas paralelos.
- Creación interactiva de índices.

Ejercicios prácticos

21.1 En una selección de rango sobre un atributo dividido por rangos es posible que sólo haga falta acceder a un disco. Descríbanse las ventajas y los inconvenientes de esta propiedad.

21.2 Indíquese la forma de paralelismo (entre consultas, entre operaciones o en operaciones) que puede resultar más importante para cada una de las tareas siguientes.

a. Incrementar la productividad de un sistema con muchas consultas pequeñas.
b. Incrementar la productividad de un sistema con unas pocas consultas de gran tamaño cuando el número de discos y de procesadores es elevado.

21.3 Con el paralelismo de encauzamiento suele resultar conveniente llevar a cabo varias operaciones de un cauce en un mismo procesador, aunque haya disponibles varios procesadores.

a. Explíquese el motivo.
b. ¿Serían válidos los argumentos anteriores si la máquina tuviera una arquitectura de memoria compartida? Explíquese el motivo.
c. ¿Serían válidos los argumentos anteriores con paralelismo independiente? Es decir, ¿hay casos en que, incluso si las operaciones no se encauzan y hay muchos procesadores disponibles, sigue siendo conveniente llevar a cabo varias operaciones en el mismo procesador?

21.4 Considérese el procesamiento de reuniones utilizando el esquema simétrico de fragmentos y réplicas con división por rangos. ¿Cómo se puede optimizar la evaluación si la condición de reunión es de la forma $| r.A - s.B | \leq k$, donde k es una constante pequeña? Aquí, $| x |$ denota el valor absoluto de x. Las reuniones con una condición de reunión así se denominan **reuniones de banda**.

21.5 Recuérdese que los histogramas se utilizan para generar particiones de rangos con carga equilibrada.

a. Supóngase que se tiene un histograma en el que los valores varían de 1 a 100 y están divididos en 10 rangos, 1–10, 11–20, ..., 91–100, con las frecuencias 15, 5, 20, 10, 10, 5, 5, 20, 5 y 5, respectivamente. Propóngase una función de división por rangos con carga equilibrada para dividir los valores en cinco particiones.

b. Escríbase un algoritmo para calcular una división por rangos con carga equilibrada con p particiones, dado un histograma de las distribuciones de frecuencias que contenga n rangos.

21.6 Algunos sistemas paralelos de bases de datos almacenan una copia adicional de cada elemento de los datos en discos conectados a un procesador diferente, para evitar la pérdida de los datos si falla alguno de los procesadores.

a. ¿Por qué es conveniente dividir las copias de los elementos de los datos de cada procesador entre varios procesadores?

b. ¿Cuáles son las ventajas y los inconvenientes de utilizar almacenamiento RAID en lugar de almacenar otra copia de cada elemento de datos?

Ejercicios

21.7 Para cada una de las tres técnicas de división, es decir, por turno rotatorio, por asociación y por rangos, propóngase un ejemplo de consulta para la que esa técnica de división proporcione la respuesta más rápida.

21.8 Indíquense los factores que pueden dar lugar a sesgo cuando se divide una relación de acuerdo con uno de sus atributos utilizando:

a. División por asociación

b. División por rangos

En cada caso, indíquese lo que se puede hacer para reducir el sesgo.

21.9 Propóngase un ejemplo de reunión, que no sea una equirreunión simple, para la que pueda utilizarse paralelismo de divisiones. ¿Qué atributos deberían utilizarse para la división?

21.10 Descríbase una buena manera de paralelizar lo siguiente.

a. La operación diferencia.

b. La agregación utilizando la operación **count**.

c. La agregación utilizando la operación **count distinct**.

d. La agregación utilizando la operación **avg**.

e. La reunión externa por la izquierda, si la condición de reunión sólo implica igualdad.

f. La reunión externa por la izquierda, si la condición de reunión implica comparaciones distintas de la igualdad.

g. La reunión externa completa, si la condición de reunión implica comparaciones distintas de la igualdad.

21.11 Descríbanse las ventajas y los inconvenientes del paralelismo de encauzamiento.

21.12 Supóngase que se desea tratar una carga de trabajo consistente en gran cantidad de transacciones de pequeño tamaño mediante paralelismo independiente.

a. ¿Se necesita paralelismo en consultas en esta situación? En caso de no ser así, indicar el motivo y la forma de paralelismo que se considera adecuada.

b. ¿Qué forma de sesgo sería relevante con esta carga de trabajo?

c. Supóngase que la mayoría de las transacciones accediesen a un registro de *cuenta*, que incluye un atributo del tipo de cuenta, y a un registro asociado *maestro_tipo_cuenta*, que proporciona la información sobre el tipo de la cuenta. ¿Cómo se dividirían y/o duplicarían los datos para acelerar las transacciones? Se puede suponer que la relación *maestro_tipo_cuenta* se actualiza rara vez.

Notas bibliográficas

Los sistemas de bases de datos relacionales comenzaron a aparecer en el mercado en 1983; hoy en día, lo dominan. A finales de los 70 y comienzos de los 80, a medida que el modelo relacional lograba un fundamento razonablemente sólido, se fue reconociendo que los operadores relacionales se pueden paralelizar en gran medida y que tienen buenas propiedades de flujo de datos. Se lanzaron en rápida sucesión un sistema comercial, Teradata, y varios proyectos de investigación, como GRACE (Kitsuregawa et al. [1983], Fushimi et al. [1986]), GAMMA (DeWitt et al. [1986], DeWitt [1990]) y Bubba (Boral et al. [1990]). Los investigadores utilizaron estos sistemas paralelos de bases de datos para estudiar la viabilidad de la ejecución en paralelo de los operadores relacionales. Posteriormente, a finales de los 80 y en los 90, varias compañías más—como Tandem, Oracle, Sybase, Informix y Red-Brick (ahora parte de Informix, que a su vez es parte de IBM)—entraron en el mercado de las bases de datos paralelas. Los proyectos de investigación dentro del mundo académico incluyen XPRS (Stonebraker et al. [1989]) y Volcano (Graefe [1990]).

El bloqueo en las bases de datos paralelas se estudia en Joshi [1991], Mohan y Narang [1991] y Mohan y Narang [1992]. Los protocolos de coherencia de la caché para los sistemas paralelos de bases de datos se estudian en Dias et al. [1989], Mohan y Narang [1991], Mohan y Narang [1992] y Rahm [1993]. Carey et al. [1991] estudian los aspectos de la caché en los sistemas cliente-servidor. El paralelismo y la recuperación en sistemas de bases de datos se estudian en Bayer et al. [1980].

Graefe [1993] presenta un excelente resumen del procesamiento de consultas, incluido el procesamiento de consultas en paralelo. Graefe [1990] y Graefe [1993] defendieron el modelo del operador de intercambio.

La ordenación en paralelo se estudia en DeWitt et al. [1992]. Los algoritmos de reunión en paralelo se describen en Nakayama et al. [1984], Kitsuregawa et al. [1983], Richardson et al. [1987], Schneider y DeWitt [1989], Kitsuregawa y Ogawa [1990], Lin et al. [1994] y Wilschut et al. [1995], entre otros trabajos. Los algoritmos de la reunión en paralelo para arquitecturas de memoria compartida se describen en Tsukuda et al. [1992], Deshpande y Larson [1992], y Shatdal y Naughton [1993].

El tratamiento del sesgo en las reuniones en paralelo se describe en Walton et al. [1991], Wolf [1991] y DeWitt et al. [1992].

Las técnicas de optimización de consultas en paralelo se describen en H. Lu y Tan [1991], Hong y Stonebraker [1991], Ganguly et al. [1992], Lanzelotte et al. [1993] y Jhingran et al. [1997].

CAPÍTULO 22

Bases de datos distribuidas

A diferencia de los sistemas paralelos, en los que los procesadores se hallan estrechamente acoplados y constituyen un solo sistema de bases de datos, los sistemas de bases de datos distribuidos están formados por sitios débilmente acoplados que no comparten ningún componente físico. Además, puede que los sistemas de bases de datos que se ejecutan en cada sitio sean sustancialmente independientes entre sí. La estructura básica de los sistemas distribuidos se estudió en el Capítulo 20.

Cada sitio puede participar en la ejecución de transacciones que acceden a los datos de uno o de varios sitios diferentes. La diferencia principal entre los sistemas de bases de datos centralizados y los distribuidos es que, en los primeros, los datos residen en una única ubicación, mientras que en los segundos los datos se reparten entre varios lugares. Esta distribución de los datos provoca muchas dificultades en el procesamiento de las transacciones y de las consultas. En este capítulo se abordarán esas dificultades.

Se comenzará por clasificar las bases de datos distribuidas en homogéneas y heterogéneas, en el Apartado 22.1. A continuación se abordará el problema del almacenamiento de los datos en las bases de datos distribuidas en el Apartado 22.2. El Apartado 22.3 esboza un modelo de procesamiento de las transacciones en las bases de datos distribuidas. En el Apartado 22.4 se describe la manera de implementar transacciones atómicas en bases de datos distribuidas mediante protocolos de compromiso especiales. El Apartado 22.5 describe el control de concurrencia en las bases de datos distribuidas. En el Apartado 22.6 se esboza el modo de proporcionar una elevada disponibilidad en bases de datos distribuidas aprovechando las réplicas, de manera que el sistema pueda continuar procesando las transacciones aunque se produzca una avería. El procesamiento de las consultas en las bases de datos distribuidas se aborda en el Apartado 22.7. En el Apartado 22.8 se esbozan aspectos del manejo de bases de datos heterogéneas. El Apartado 22.9 describe los sistemas de directorio, que pueden considerarse una forma especializada de las bases de datos distribuidas.

22.1 Bases de datos homogéneas y heterogéneas

En los sistemas de **bases de datos distribuidas homogéneas** todos los sitios emplean idéntico software de gestión de bases de datos, son conscientes de la existencia de los demás sitios y acuerdan cooperar en el procesamiento de las solicitudes de los usuarios. En estos sistemas, los sitios locales renuncian a una parte de su autonomía en cuanto a su derecho a modificar los esquemas o el software de gestión de bases de datos. Ese software también debe cooperar con los demás sitios en el intercambio de la información sobre las transacciones para hacer posible su procesamiento entre varios sitios.

A diferencia de lo anterior, en las **bases de datos distribuidas heterogéneas** puede que los diferentes sitios utilicen esquemas y software de gestión de sistemas de bases de datos diferentes. Puede que algunos sitios no tengan información de la existencia del resto y que sólo proporcionen facilidades limitadas para la cooperación en el procesamiento de las transacciones. Las diferencias en los esquemas suelen

constituir un problema importante para el procesamiento de las consultas, mientras que la divergencia del software supone un inconveniente para el procesamiento de transacciones que acceden a varios sitios.

Este capítulo se centrará en las bases de datos distribuidas homogéneas. No obstante, en el Apartado 22.8 se estudiarán brevemente los aspectos del procesamiento de las consultas en los sistemas de bases de datos distribuidas heterogéneas. Los aspectos del procesamiento de las transacciones en esos sistemas se tratan más adelante, en el Apartado 25.7.

22.2 Almacenamiento distribuido de datos

Considérese una relación *r* que hay que almacenar en la base de datos. Existen dos enfoques del almacenamiento de esta relación en la base de datos distribuida:

- **Réplica**. El sistema conserva varias réplicas (copias) idénticas de la relación y guarda cada réplica en un sitio diferente. La alternativa a las réplicas es almacenar sólo una copia de la relación *r*.
- **Fragmentación**. El sistema divide la relación en varios fragmentos y guarda cada fragmento en un sitio diferente.

La fragmentación y la réplica pueden combinarse: las relaciones pueden dividirse en varios fragmentos y puede haber varias réplicas de cada fragmento. En los subapartados siguientes se profundizará en cada una de estas técnicas.

22.2.1 Réplica de datos

Si la relación *r* se replica, se guarda una copia de esa relación en dos o más sitios. En el caso más extremo se tiene una **réplica completa**, en la que se guarda una copia en cada sitio del sistema.

Las réplicas presentan varias ventajas e inconvenientes.

- **Disponibilidad**. Si alguno de los sitios que contiene la relación *r* falla, esa relación puede hallarse en otro sitio distinto. Por tanto, el sistema puede seguir procesando las consultas que impliquen a *r*, pese al fallo del sitio.
- **Paralelismo incrementado**. En el caso en el que la mayoría de los accesos a la relación *r* sólo resultasen en lecturas, diferentes sitios podrían procesar en paralelo las lecturas que impliquen a *r*. Cuantas más réplicas de *r* existan, mayor será la posibilidad de que los datos necesarios se encuentren en el sitio en que se ejecuta la transacción. Por tanto, la réplica de los datos minimiza su transmisión entre los diferentes sitios.
- **Sobrecarga incrementada durante la actualización**. El sistema debe asegurar que todas las réplicas de la relación *r* sean consistentes; en caso contrario pueden producirse cálculos erróneos. Por tanto, siempre que se actualiza *r*, hay que propagar la actualización a todos los sitios que contienen réplicas. El resultado es una sobrecarga incrementada. Por ejemplo, en un sistema bancario, en el que la información de las cuentas se replica en varios sitios, es necesario asegurarse de que el saldo de cada cuenta concuerde en todos ellos.

En general, la réplica mejora el rendimiento de las operaciones de lectura y aumenta la disponibilidad de los datos para las transacciones de lectura. Sin embargo, las transacciones de actualización suponen una mayor sobrecarga. El control de las actualizaciones concurrentes de los datos replicados realizadas por varias transacciones resulta más complejo que en los sistemas centralizados, los cuales se estudiaron en el Capítulo 16. Se puede simplificar la gestión de las réplicas de la relación *r* escogiendo una de ellas como **copia principal** de *r*. Por ejemplo, en un sistema bancario, las cuentas pueden asociarse con el sitio en que se abrieron. De manera parecida, en un sistema de reserva de billetes de avión, cada vuelo puede asociarse con el sitio en que se origina. El esquema de copias principales y otras opciones del control de concurrencia distribuida se examinarán en el Apartado 22.5.

22.2.2 Fragmentación de los datos

Si la relación *r* se fragmenta, *r* se divide en varios *fragmentos* $r_2, \ldots, r_n$. Estos fragmentos contienen suficiente información como para permitir la reconstrucción de la relación original *r*. Existen dos esquemas diferentes de fragmentación de las relaciones: la fragmentación *horizontal* y la *vertical*. La fragmentación horizontal divide la relación asignando cada tupla de *r* a uno o más fragmentos. La fragmentación vertical divide la relación descomponiendo el esquema *r* de la relación *r*.

Estos enfoques se ilustrarán fragmentando la relación *cuenta*, con el esquema

Esquema_cuenta = (*número_cuenta*, *nombre_sucursal*, *saldo*)

En la **fragmentación horizontal** la relación *r* se divide en varios subconjuntos, r_1, $r_2, \ldots, r_n$. Cada tupla de la relación *r* debe pertenecer, como mínimo, a uno de los fragmentos, de modo que se pueda reconstruir la relación original, si fuera necesario.

A modo de ejemplo, la relación *cuenta* puede dividirse en varios fragmentos, cada uno de los cuales consiste en tuplas de cuentas pertenecientes a una sucursal concreta. Si el sistema bancario sólo tiene dos sucursales (Guadarrama y Cercedilla), habrá dos fragmentos diferentes:

$$cuenta_1 = \sigma_{nombre_sucursal = \text{“Guadarrama”}}(cuenta)$$
$$cuenta_2 = \sigma_{nombre_sucursal = \text{“Cercedilla”}}(cuenta)$$

La fragmentación horizontal suele emplearse para conservar las tuplas en los sitios en que más se utilizan, para minimizar la transferencia de datos.

En general, los fragmentos horizontales pueden definirse como una *selección* de la relación global *r*. Es decir, se utiliza un predicado P_i para construir el fragmento r_i:

$$r_i = \sigma_{P_i}(r)$$

La relación *r* se reconstruye tomando la unión de todos los fragmentos; es decir,

$$r = r_1 \cup r_2 \cup \cdots \cup r_n$$

En el ejemplo, los fragmentos son disjuntos. Al cambiar los predicados de selección empleados para crear los fragmentos se puede hacer que una tupla de *r* dada aparezca en más de uno de los fragmentos r_i.

En su forma más sencilla la fragmentación vertical es igual que la descomposición (véase el Capítulo 7). La **fragmentación vertical** de *r*(*R*) implica la definición de varios subconjuntos de atributos $R_1, R_2, \ldots, R_n$ del esquema *R* de modo que

$$R = R_1 \cup R_2 \cup \cdots \cup R_n$$

Cada fragmento r_i de *r* se define mediante

$$r_i = \Pi_{R_i}(r)$$

La fragmentación debe hacerse de modo que se pueda reconstruir la relación *r* a partir de los fragmentos tomando la reunión natural

$$r = r_1 \bowtie r_2 \bowtie r_3 \bowtie \cdots \bowtie r_n$$

Una manera de asegurar que la relación r pueda reconstruirse es incluir los atributos de la clave principal de R en cada uno de los fragmentos R_i. De manera más general, se puede utilizar cualquier superclave. Suele resultar conveniente añadir un atributo especial, denominado *id_tupla*, al esquema *R*. El valor id_tupla de cada tupla es un valor único que distingue a esa tupla de todas las demás. El atributo id_tupla, por tanto, sirve como clave candidata para el esquema aumentado y se incluye en cada uno de los fragmentos R_i. La dirección física o lógica de la tupla puede utilizarse como id_tupla, ya que cada tupla tiene una dirección única.

Para ilustrar la fragmentación vertical considérese una base de datos universitaria con una relación *info_empleado* que almacena, para cada empleado, *id_empleado*, *nombre*, *puesto* y *salario*. Por motivos de preservación de la intimidad puede que esta relación se fragmente en una denominada *empleado_infoprivada*, que contenga *id_empleado* y *salario*, y en otra llamada *empleado_infopública*, que contenga los

atributos *id_empleado*, *nombre* y *puesto*. Puede que las dos relaciones se almacenen en sitios diferentes, nuevamente, por motivos de seguridad.

Se pueden aplicar los dos tipos de fragmentación a un mismo esquema; por ejemplo, los fragmentos obtenidos de la fragmentación horizontal de una relación pueden dividirse nuevamente de manera vertical. Los fragmentos también pueden replicarse. En general, los fragmentos pueden replicarse, las réplicas de los fragmentos pueden fragmentarse más, y así sucesivamente.

22.2.3 Transparencia

No se debe exigir a los usuarios de los sistemas distribuidos de bases de datos que conozcan la ubicación física de los datos ni el modo en que se puede acceder a ellos en cada sitio local concreto. Esta característica, denominada **transparencia de los datos**, puede adoptar varias formas:

- **Transparencia de la fragmentación**. No se exige a los usuarios que conozcan el modo en que se ha fragmentado la relación.
- **Transparencia de la réplica**. Los usuarios ven cada objeto de datos como lógicamente único. Puede que el sistema distribuido replique los objetos para incrementar el rendimiento del sistema o la disponibilidad de los datos. Los usuarios no deben preocuparse por los objetos que se hayan replicado ni por la ubicación de esas réplicas.
- **Transparencia de la ubicación**. No se exige a los usuarios que conozcan la ubicación física de los datos. El sistema distribuido de bases de datos debe poder hallar los datos siempre que la transacción del usuario facilite el identificador de esos datos.

Los elementos de datos (como las relaciones, los fragmentos y las réplicas) deben tener nombres únicos. Esta propiedad es fácil de asegurar en las bases de datos centralizadas. En las bases de datos distribuidas, sin embargo, hay que tener cuidado para asegurarse de que dos sitios no utilicen el mismo nombre para elementos de datos diferentes.

Una solución a este problema es exigir que todos los nombres se registren en un **servidor de nombres** central. El servidor de nombres ayuda a garantizar que el mismo nombre no se utilice para elementos de datos diferentes. También se puede utilizar el servidor de nombres para localizar los elementos de datos, dado su nombre. Este enfoque, sin embargo, presenta dos inconvenientes principales. En primer lugar, puede que el servidor de nombres se transforme en un cuello de botella para el rendimiento cuando los elementos de datos se buscan por el nombre, lo que da lugar a un bajo rendimiento. En segundo lugar, si el servidor de nombres queda fuera de servicio, puede que ningún otro sitio del sistema distribuido logre seguir en funcionamiento.

Un enfoque alternativo más utilizado exige que cada sitio anteponga su propio identificador de sitio a cualquier nombre que genere. Este enfoque garantiza que dos sitios diferentes no generen nunca el mismo nombre (dado que cada sitio tiene un identificador único). Además, no se necesita ningún control centralizado. Esta solución, no obstante, no logra conseguir transparencia en la ubicación, dado que a los nombres se les adjuntan los identificadores de los sitios. Así, se puede hacer referencia a la relación *cuenta* como *cuenta.sitio*17, o *cuenta@sitio*17, en lugar de meramente *cuenta*. Muchos sistemas de bases de datos utilizan la dirección de Internet de los sitios para identificarlos.

Para superar este problema, el sistema de bases de datos puede crear un conjunto de nombres alternativos, o **alias**, para los elementos de datos. En consecuencia, los usuarios se pueden referir a los elementos de datos mediante nombres sencillos que el sistema traduce a los nombres completos. La relación entre los alias y los nombres reales puede almacenarse en cada sitio. Cuando se emplean alias, el usuario puede ignorar la ubicación física de los elementos de datos. Además, no se ve afectado si el administrador de la base de datos decide trasladar un elemento de datos de un sitio a otro.

Los usuarios no deberían tener necesidad de hacer referencia a una réplica concreta de un elemento de datos. En vez de eso, el sistema debe determinar la réplica a la que hay que hacer referencia en las solicitudes leer, y actualizar todas las réplicas en las solicitudes escribir. Se puede asegurar que lo hace si se mantiene una tabla de catálogo, que el sistema utilizará para determinar todas las réplicas del elemento de datos.

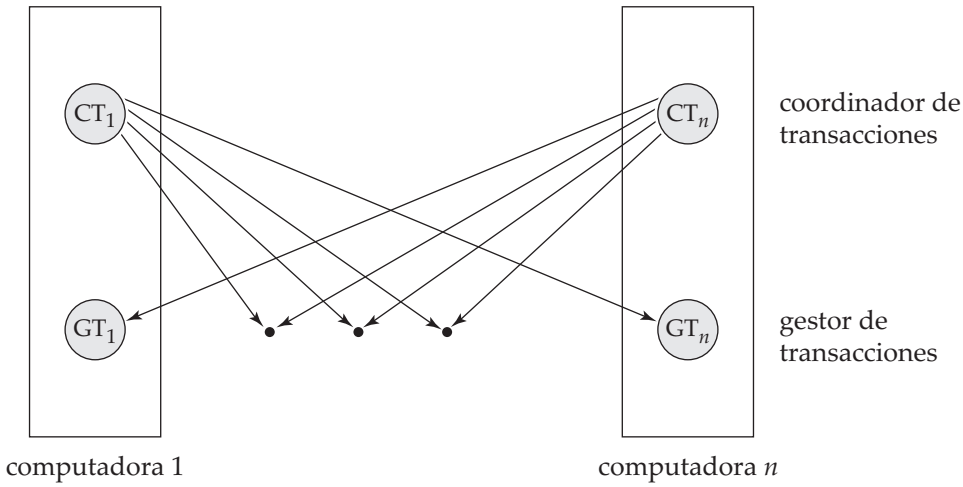


Figura 22.1 Arquitectura del sistema.

22.3 Transacciones distribuidas

El acceso a los diferentes elementos de datos en los sistemas distribuidos suele realizarse mediante transacciones, que deben preservar las propiedades ACID (Apartado 15.1). Se deben considerar dos tipos de transacciones. Las **transacciones locales** son las que acceden a los datos y los actualizan en una única base de datos local; las **transacciones globales** son las que acceden a los datos y los actualizan en varias bases de datos locales. Las propiedades ACID de las transacciones locales se pueden asegurar como se describe en los Capítulos 15, 16 y 17. No obstante, para las transacciones globales, esta tarea resulta mucho más complicada, dado que puede que participen en la ejecución varios sitios. El fallo de alguno de estos sitios, o el de alguno de los enlaces de comunicaciones que los conectan entre sí, puede dar lugar a cálculos erróneos.

En este apartado se estudia la estructura del sistema de una base de datos distribuida y sus posibles modos de fallo. Con base en el modelo presentado en este apartado, en el Apartado 22.4 se estudian los protocolos para garantizar el compromiso atómico de las transacciones globales, y en el Apartado 22.5 se estudian los protocolos para el control de concurrencia en las bases de datos distribuidas. En el Apartado 22.6 se estudia el modo en que pueden seguir funcionando las bases de datos distribuidas incluso en presencia de varios tipos de fallo.

22.3.1 Estructura del sistema

Cada sitio tiene su propio gestor *local* de transacciones, cuya función es garantizar las propiedades ACID de las transacciones que se ejecuten allí. Los diferentes gestores de transacciones colaboran para ejecutar las transacciones globales. Para comprender el modo en que se pueden implementar estos gestores, considérese un modelo abstracto de sistema de transacciones, en el que cada sitio contenga dos subsistemas:

- El **gestor de transacciones** gestiona la ejecución de las transacciones (o subtransacciones) que acceden a los datos almacenados en un sitio local. Téngase en cuenta que cada una de esas transacciones puede ser local (es decir, una transacción que se ejecuta sólo en ese sitio) o formar parte de una transacción global (es decir, una transacción que se ejecuta en varios sitios).
- El **coordinador de transacciones** coordina la ejecución de las diferentes transacciones (tanto locales como globales) iniciadas en ese sitio.

La arquitectura global del sistema aparece en la Figura 22.1.

La estructura de los gestores de transacciones es parecida en muchos aspectos a la de los sistemas centralizados. Cada gestor de transacciones es responsable de:

- Mantener un registro histórico con fines de recuperación.

- Participar en un esquema adecuado de control de concurrencia para coordinar la ejecución concurrente de las transacciones que se ejecuten en ese sitio.

Como se verá, es necesario modificar tanto el esquema de recuperación como el de concurrencia para adaptarlos a la distribución de las transacciones.

El subsistema del coordinador de transacciones no es necesario en los entornos centralizados, ya que las transacciones sólo acceden a los datos de un sitio. Los coordinadores de transacciones, como su propio nombre implica, son responsables de la coordinación de la ejecución de todas las transacciones iniciadas en ese sitio. En cada una de esas transacciones el coordinador es responsable de:

- Iniciar la ejecución de la transacción.
- Dividir la transacción en varias subtransacciones y distribuir esas subtransacciones a los sitios correspondientes para su ejecución.
- Coordinar la terminación de la transacción, lo que puede hacer que la transacción se comprometa o se aborte en todos los sitios.

22.3.2 Modos de fallo del sistema

Los sistemas distribuidos pueden sufrir los mismos tipos de fallos que los sistemas centralizados (por ejemplo, errores de software, errores de hardware y fallos de discos). No obstante, en los entornos distribuidos también hay que tratar con otros tipos de fallos. Los tipos básicos de fallos son:

- Fallos de sitios.
- Pérdidas de mensajes.
- Fallos de enlaces de comunicaciones.
- Divisiones de la red.

En los sistemas distribuidos siempre es posible la pérdida o el deterioro de los mensajes. El sistema utiliza protocolos de control de las transmisiones, como TCP/IP, para tratar esos errores. Se puede encontrar información sobre esos protocolos en los libros de texto estándar sobre redes (véanse las notas bibliográficas).

No obstante, si dos sitios A y B no se hallan conectados de manera directa, los mensajes de uno a otro deben *encaminarse* mediante una serie de enlaces de comunicaciones. Si falla uno de los enlaces, hay que volver a encaminar los mensajes que debería haber transmitido. En algunos casos se puede hallar otra ruta por la red, de modo que los mensajes puedan alcanzar su destino. En otros casos, el fallo puede hacer que no haya ninguna conexión entre los dos sitios. Un sistema está **dividido** si se ha partido en dos (o más) subsistemas, denominados **particiones**, que carecen de conexión entre ellas. Obsérvese que, con esta definición, cada subsistema puede consistir en un solo nodo.

22.4 Protocolos de compromiso

Si hay que asegurar la atomicidad, todos los sitios en los que se ejecute una transacción T deben coincidir en el resultado final de esa ejecución. T debe comprometerse o abortarse en todos los sitios. Para garantizar esta propiedad, el coordinador de transacciones de T debe ejecutar un *protocolo de compromiso*.

Entre los protocolos de compromiso más sencillos y más utilizados está el **protocolo de compromiso de dos fases** (C2F), que se describe en el Apartado 22.4.1. Una alternativa es el **protocolo de compromiso de tres fases** (C3F), que evita ciertos inconvenientes del protocolo C2F pero añade complejidad y sobrecarga. El Apartado 22.4.2 describe brevemente el protocolo C3F.

22.4.1 Compromiso de dos fases

En primer lugar se describe el modo en que opera el protocolo de compromiso de dos fases (C2F) durante el funcionamiento normal, luego se describe cómo maneja los fallos y, finalmente, la manera en que ejecuta la recuperación y el control de concurrencia.

Considérese una transacción T iniciada en el sitio S_i, en el que el coordinador de transacciones es C_i.

22.4.1.1 El protocolo de compromiso

Cuando se acaba de ejecutar T (es decir, cuando todos los sitios en los que se ha ejecutado informan a C_i de que T se ha completado) C_i inicia el protocolo C2F.

- **Fase 1**. C_i añade el registro <preparar T> al registro histórico y obliga a que se guarde en un lugar de almacenamiento estable. Luego envía el mensaje preparar T a todos los sitios en los que se ha ejecutado T. Al recibir este mensaje, el gestor de transacciones de cada sitio determina si desea comprometer su parte de T. Si la respuesta es negativa, añade el registro <no T> al registro histórico y responde enviando a C_i el mensaje abortar T. Si la respuesta es positiva, añade el registro <T preparada> al registro histórico y hace que se guarde (con todos los registros del registro histórico correspondientes a T) en un almacenamiento estable. El gestor de transacciones contesta entonces a C_i con el mensaje T preparada.
- **Fase 2**. Cuando C_i recibe de todos los sitios las respuestas al mensaje preparar T, o cuando ha transcurrido un intervalo de tiempo especificado con anterioridad desde que se envió el mensaje preparar T, C_i puede determinar si la transacción T puede comprometerse o abortarse. La transacción T se puede comprometer si C_i ha recibido el mensaje T preparada de todos los sitios participantes. En caso contrario, hay que abortar la transacción T. En función del resultado, se añade el registro <T comprometida> o <T abortada> al registro histórico, que se guarda en un almacenamiento estable. En ese momento, el destino de la transacción ya se ha sellado. A partir de este momento el coordinador envía a todos los sitios participantes el mensaje comprometer T o abortar T. Cuando un sitio recibe ese mensaje, lo guarda en el registro histórico.

Los sitios en los que se ejecutó T pueden abortarla de manera incondicional en cualquier momento antes de enviar al coordinador el mensaje T preparada. Una vez enviado el mensaje, se dice que la transacción está en **estado preparado** en el sitio. El mensaje T preparada constituye, en realidad, un compromiso del sitio de acatar la orden del coordinador de comprometer o de abortar T. Para establecer ese compromiso, primero hay que guardar en un almacenamiento estable la información necesaria. En caso contrario, si el sitio fallara tras enviar el mensaje T preparada, puede que no fuera capaz de cumplir su promesa. Además, los bloqueos adquiridos por la transacción deben mantenerse hasta que ésta se complete.

Dado que se exige la unanimidad para comprometer cada transacción, el destino de T queda sellado en cuanto un sitio responda abortar T. Dado que el sitio coordinador S_i es uno de los sitios en los que se ha ejecutado T, el coordinador puede decidir unilateralmente abortarla. El veredicto final sobre T se determina en el momento en que el coordinador lo escribe (comprometer o abortar) en el registro histórico y obliga a que ese veredicto se guarde en un almacenamiento estable. En algunas implementaciones del protocolo C2F, los sitios envían al coordinador el mensaje acuse-de-recibo T al final de la segunda fase del protocolo. Cuando el coordinador recibe el mensaje acuse-de-recibo T de todos los sitios, añade el registro <T completada> al registro histórico.

22.4.1.2 Tratamiento de los fallos

El protocolo C2F responde de modo diferente a los distintos tipos de fallos:

- **Fallo de un sitio participante**. Si el coordinador C_i detecta que un sitio ha fallado, emprende las acciones siguientes: si el sitio falla antes de responder a C_i con el mensaje T preparada, el coordinador da por supuesto que ha respondido con el mensaje abortar T. Si el sitio falla después

de que el coordinador haya recibido del sitio el mensaje T preparada, el coordinador ejecuta el resto del protocolo de compromiso de manera normal, ignorando el fallo del sitio.

Cuando el sitio participante S_k se recupera de un fallo, debe examinar su registro histórico para determinar el destino de las transacciones que se hallaban en trance de ejecución cuando se produjo ese fallo. Supóngase que T es una de esas transacciones. Se toman en consideración cada uno de los casos posibles:

- El registro histórico contiene el registro $<T$ comprometida$>$. En ese caso, el sitio ejecuta rehacer(T).
- El registro histórico contiene el registro $<T$ abortada$>$. En ese caso, el sitio ejecuta deshacer(T).
- El registro histórico contiene el registro $<T$ preparada$>$. En ese caso, el sitio debe consultar con C_i para determinar el destino de T. Si C_i está activo, notifica a S_k si T se comprometió o se abortó. En el primer caso, se ejecuta rehacer(T); en el segundo, se ejecuta deshacer(T). Si C_i no está activo, S_k debe intentar averiguar el destino de T consultando a otros sitios. Lo hace enviando el mensaje consulta-estado T a todos los sitios del sistema. Al recibir ese mensaje, cada sitio debe consultar su registro histórico para determinar si allí se ejecutó T y, en caso afirmativo, si se comprometió o se abortó. Luego notifica a S_k el resultado. Si ningún sitio tiene la información correspondiente (es decir, si T se comprometió o se abortó), S_k no puede abortar ni comprometer T. La decisión sobre T se pospone hasta que S_k pueda obtener la información necesaria. Por tanto, S_k debe volver a enviar de manera periódica el mensaje consulta-estado a los demás sitios. Seguirá haciéndolo hasta que se recupere algún sitio que contenga la información necesaria. Téngase en cuenta que el sitio en el que reside C_i siempre tiene la información necesaria.
- El registro histórico no contiene ningún registro de control (abortada, comprometida, preparada) relativo a T. Por tanto, se sabe que S_k falló antes de responder al mensaje preparar T enviado por C_i. Dado que el fallo de S_k evitó el envío de la respuesta, de acuerdo con el algoritmo, C_i debe abortar T. Por tanto, S_k debe ejecutar deshacer(T).

- **Fallo del coordinador**. Si el coordinador falla durante la ejecución del protocolo de compromiso para la transacción T, los sitios participantes deben decidir el destino de T. Se verá que, en ciertos casos, los sitios participantes no pueden decidir si comprometer o abortar T y, por tanto, deben esperar a la recuperación del coordinador que ha fallado.
  - Si algún sitio activo contiene el registro $<T$ comprometida$>$ en su registro histórico, se debe comprometer T.
  - Si algún sitio activo contiene el registro $<T$ abortada$>$ en su registro histórico, se debe abortar T.
  - Si algún sitio activo *no* contiene el registro $<T$ preparada$>$ en su registro histórico, el coordinador C_i que ha fallado no puede haber decidido comprometer T, ya que los sitios que no contienen el registro $<T$ preparada$>$ en su registro histórico no pueden haber enviado el mensaje T preparada a C_i. No obstante, puede que el coordinador haya decidido abortar T, pero no comprometer T. En vez de esperar a que se recupere C_i, resulta preferible abortar T.
  - Si no se da ninguno de los casos anteriores, todos los sitios activos deben tener el registro $<T$ preparada$>$ en sus registros históricos, pero ningún otro registro de control (como $<T$ abortada$>$ o $<T$ comprometida$>$). Dado que el coordinador ha fallado, resulta imposible determinar si se ha tomado alguna decisión y, en caso de haberse tomado, averiguar la que era, hasta que se recupere el coordinador. Por tanto, los sitios activos deben esperar a que se recupere C_i. Dado que el destino de T sigue siendo dudoso, puede que siga consumiendo recursos del sistema. Por ejemplo, si se emplean bloqueos, puede que T conserve en los sitios activos los bloqueos sobre los datos. Esa situación no es deseable, ya que pueden pasar horas o días antes de que C_i vuelva a estar activo. Durante ese tiempo puede que otras transacciones se vean obligadas a esperar a T. En consecuencia, puede que los elementos de datos no estén disponibles, no sólo en el sitio que ha fallado (C_i), sino también en los sitios activos. Esta situación se denomina problema del **bloqueo**, ya que T queda bloqueada a la espera de la recuperación del sitio C_i.

- **División de la red**. Cuando una red queda dividida, caben dos posibilidades:
  1. El coordinador y todos los sitios participantes siguen en una de las particiones. En este caso, el fallo no tiene ningún efecto sobre el protocolo de compromiso.
  2. El coordinador y los sitios participantes se distribuyen entre varias particiones. Desde el punto de vista de los sitios de cada partición, parece que los sitios de las demás particiones hubieran fallado. Los sitios que no se hallan en la partición que contiene al coordinador, sencillamente, ejecutan el protocolo para tratar el fallo del coordinador. El coordinador y los sitios que se hallan en su misma partición siguen el protocolo de compromiso habitual, dando por supuesto que los sitios de las demás particiones han fallado.

Por tanto, el mayor inconveniente del protocolo C2F es que el fallo del coordinador puede dar lugar a un bloqueo, en el que puede que haya que retrasar la decisión sobre comprometer o abortar T hasta que se recupere C_i.

22.4.1.3 Recuperación y control de concurrencia

Cuando se reinicia el sitio que ha fallado, se puede llevar a cabo la recuperación, por ejemplo, utilizando el algoritmo de recuperación descrito en el Apartado 17.8. Para tratar con los protocolos de compromiso distribuidos (como C2F y C3F), el procedimiento de recuperación debe tratar de manera especial las **transacciones dudosas**; las transacciones dudosas son transacciones para las que se encuentra en el registro histórico un registro $<T$ preparada$>$, pero no uno $<T$ comprometida$>$ ni uno $<T$ abortada$>$. El sitio que se recupera debe determinar la situación comprometer-abortar de esas transacciones, como se describe en el Apartado 22.4.1.2.

Sin embargo, si se realiza la recuperación como se acaba de describir, el procesamiento normal de las transacciones en el sitio no puede comenzar hasta que se hayan comprometido o deshecho todas las transacciones dudosas. Averiguar la situación de las transacciones dudosas puede ser un proceso lento, ya que puede que haya que contactar con varios sitios. Además, si ha fallado el coordinador, y ningún otro sitio tiene información sobre la situación comprometer-abortar de una transacción incompleta, se podría bloquear la recuperación si se utiliza C2F. En consecuencia, puede que el sitio que lleva a cabo la recuperación de reinicio quede inutilizable durante un largo periodo de tiempo.

Para evitar este problema los algoritmos de recuperación suelen ofrecer soporte para anotar en el registro histórico la información relativa a los bloqueos (se da por supuesto que se utilizan los bloqueos para el control de concurrencia). En lugar de escribir en el registro histórico el registro $<T$ preparada$>$, el algoritmo escribe $<T$ preparada, $L>$, donde L es una lista de todos los bloqueos de escritura que tiene la transacción T cuando se escribe el registro del registro histórico. En el momento de la recuperación, tras llevar a cabo las acciones de recuperación locales, se renuevan para cada transacción dudosa T todos los bloqueos de escritura anotados en el registro $<T$ preparada, $L>$ del registro histórico (tras leerlos en el registro histórico).

Después de que se haya completado la renovación de los bloqueos para todas las transacciones dudosas, puede comenzar en el sitio el procesamiento de las transacciones, incluso antes de que se determine el estado comprometer—abortar de las transacciones dudosas. Las transacciones dudosas se comprometen o se deshacen de manera concurrente con la ejecución de las nuevas transacciones. Así, la recuperación del sitio es más rápida y no se bloquea nunca. Obsérvese que las transacciones nuevas que tengan conflictos de bloqueo con algún bloqueo de escritura establecido por cualquier transacción dudosa no pueden progresar hasta que se hayan comprometido o deshecho las transacciones dudosas con las que estén en conflicto.

22.4.2 Compromiso de tres fases

El protocolo de compromiso de tres fases (C3F) es una extensión del protocolo de compromiso de dos fases que evita el problema del bloqueo con determinadas suposiciones. En concreto, se supone que no se produce ninguna fragmentación de la red y que no fallan más de k sitios, donde k es un número predeterminado. Con estas suposiciones, el protocolo evita el bloqueo introduciendo una tercera fase adicional en que varios sitios se implican en la decisión sobre el compromiso. En lugar de anotar directamente la decisión sobre el compromiso en su almacenamiento persistente, el coordinador se asegura

antes de que, al menos, otros k sitios sepan que pretende comprometer la transacción. Si el coordinador falla, los sitios restantes seleccionan primero un nuevo coordinador. Este nuevo coordinador comprueba el estado del protocolo a partir de los demás sitios; si el coordinador había decidido comprometer, al menos uno de los otros k sitios a los que informó estará funcionando y garantizará que se respete la decisión de comprometer. El nuevo coordinador vuelve a iniciar la tercera fase del protocolo si algún sitio sabía que el antiguo coordinador pretendía comprometer la transacción. En caso contrario, el nuevo coordinador la aborta.

Aunque el protocolo C3F tiene la propiedad deseable de no bloquearse a menos que fallen k sitios, tiene el inconveniente de que una división de la red puede parecer lo mismo que el fallo de más de k sitios, lo que produce un bloqueo. El protocolo también tiene que implementarse con mucho cuidado para garantizar que la división de la red (o el fallo de más de k sitios) no provoque inconsistencias, en las que una transacción se comprometa en una de las particiones y se aborte en otra. Debido a la sobrecarga que supone, el protocolo C3F no se utiliza mucho. Véanse las notas bibliográficas para hallar referencias que den más detalles del protocolo C3F.

22.4.3 Modelos alternativos del procesamiento de transacciones

Para muchas aplicaciones el problema del bloqueo del compromiso de dos fases no resulta aceptable. El problema en este caso es la idea de una sola transacción que trabaja en varios sitios. En este apartado se describe el modo de utilizar la *mensajería persistente* para evitar el problema del compromiso distribuido y luego se describe brevemente el problema, más importante, de los *flujos de trabajo*; los flujos de trabajo se consideran con mayor detalle en el Apartado 25.2.

Para comprender la mensajería persistente, considérese el modo en que se podrían transferir fondos entre dos bancos diferentes, cada uno con su propia computadora. Un enfoque es hacer que la transacción abarque los dos sitios y utilizar el compromiso de dos fases para asegurar la atomicidad. Sin embargo, puede que la transacción tenga que actualizar todo el saldo del banco, y el bloqueo podría tener afectar gravemente a las demás transacciones de cada banco, ya que casi todas las transacciones de los bancos actualizan el saldo total del banco.

Por el contrario, considérese el modo en que se produce la transferencia de fondos mediante cheque conformado. El banco deduce en primer lugar el importe del cheque del saldo disponible e imprime un cheque. El cheque se transfiere físicamente al otro banco, donde se ingresa. Tras comprobar el cheque, el banco incrementa el saldo local en el importe del cheque. El cheque constituye un mensaje enviado entre los dos bancos. Para que los fondos no se pierdan ni se incrementen de manera incorrecta, el cheque no se debe perder, duplicar ni ingresar más de una vez. Cuando las computadoras de los bancos se hallan conectadas en red, los mensajes persistentes ofrecen el mismo servicio que los cheques (pero, por supuesto, mucho más rápido).

Los **mensajes persistentes** son mensajes que tienen garantizada su entrega al destinatario exactamente una sola vez (ni más, ni menos), independientemente de los fallos, si la transacción que envía el mensaje se compromete, y tienen garantizado que no se entregan si la transacción se aborta. Se emplean técnicas de recuperación de las bases de datos para implementar la mensajería persistente por encima de los canales normales de la red, como se verá brevemente. Por el contrario, los mensajes normales se pueden perder o, incluso, se pueden entregar varias veces en determinadas circunstancias.

El manejo de los errores resulta más complicado con la mensajería persistente que con el compromiso de dos fases. Por ejemplo, si la cuenta en la que hay que ingresar el cheque se ha cerrado, hay que devolver el cheque a la cuenta que lo originó y volver a cargar su importe en ella. Por tanto, ambos sitios deben disponer de código para el manejo de errores y con código para manejar los mensajes persistentes. Por el contrario, con el compromiso de dos fases, el error lo detectaría la propia transacción, que, por tanto, no deduciría nunca el importe del cheque de la primera cuenta.

Los tipos de condiciones de excepción que pueden surgir dependen de la aplicación, por lo que no es posible que el sistema de bases de datos maneje las excepciones de manera automática. Los programas de aplicación que envían y reciben los mensajes persistentes deben incluir código para el manejo de las condiciones de excepción y para la devolución del sistema a un estado consistente. Por ejemplo, no resulta aceptable que se pierda el dinero que se iba a transferir porque la cuenta receptora se haya

cerrado; hay que devolver el dinero a la cuenta ordenante, y si ello no resulta posible por algún motivo, hay que advertir a los empleados del banco para que resuelvan manualmente el problema.

Existen muchas aplicaciones en que la ventaja de la eliminación del bloqueo compensa ampliamente el esfuerzo adicional de implementar sistemas que utilicen mensajes persistentes. De hecho, pocas organizaciones aceptarían soportar el compromiso de dos fases en transacciones que se originen en su exterior, ya que los fallos pueden provocar el bloqueo del acceso a los datos locales. La mensajería persistente, por tanto, desempeña un papel importante en la ejecución de las transacciones que cruzan las fronteras de las organizaciones.

Los *flujos de trabajo* proporcionan un modelo general de procesamiento de las transacciones que implican a varios sitios y, posiblemente, el procesamiento manual por los empleados de la organización de determinadas fases del proceso. Por ejemplo, cuando un banco recibe una solicitud de préstamo, debe dar muchos pasos, incluido el contacto con agencias externas de calificación de crédito, antes de aceptar o rechazar esa solicitud. Esos pasos, en su conjunto, forman un flujo de trabajo. Los flujos de trabajo se estudian con mayor detalle en el Apartado 25.2. También se observa que la mensajería consistente forma la base subyacente a los flujos de trabajo en los entornos distribuidos.

Se considerará ahora la **implementación** de la mensajería persistente. Los siguientes protocolos pueden implementar la mensajería persistente por encima de estructuras de mensajería que no sean dignas de confianza, las cuales pueden perder mensajes o entregarlos varias veces:

- **Protocolo de sitio remitente**. Cuando una transacción desea enviar un mensaje persistente, escribe el registro que contiene el mensaje en la relación especial *mensajes_por_enviar*, en lugar de enviar el mensaje directamente. El mensaje también recibe un identificador de mensaje único.

  Un *proceso de entrega de mensajes* controla la relación y, cuando detecta un mensaje nuevo, lo envía a su destino. Los mecanismos habituales de control de concurrencia de las bases de datos garantizan que el proceso del sistema sólo lea el mensaje una vez que la transacción que lo ha escrito se haya comprometido; si la transacción se aborta, el mecanismo habitual de recuperación borra el mensaje de la relación.

  El proceso de entrega de mensajes sólo elimina los mensajes de la relación una vez que ha recibido un acuse de recibo del sitio de destino. Si no lo recibe, pasado cierto tiempo vuelve a enviar el mensaje. Repite este proceso hasta que recibe un acuse de recibo. En caso de fallo permanente, el sistema decide, pasado algún tiempo, que el mensaje no puede entregarse. Entonces invoca al código proporcionado por la aplicación para el manejo de excepciones para que trate el fallo.

  La escritura del mensaje en una relación y su procesamiento tan sólo después de que se haya comprometido la transacción garantiza que el mensaje se entregue si y sólo si la transacción se compromete. Su envío repetido garantiza que se entregue aunque haya fallos (temporales) del sistema o de la red.

- **Protocolo de sitio receptor**. Cuando un sitio recibe un mensaje persistente, ejecuta una transacción que añade el mensaje a la relación especial *mensajes_recibidos*, siempre que no se halle ya presente en la relación (el identificador único de mensajes permite detectar los duplicados). Una vez comprometida la transacción, o si el mensaje ya se hallaba en la relación, el sitio receptor devuelve un acuse de recibo al sitio remitente.

  Hay que tener en cuenta que no resulta seguro enviar el acuse de recibo antes de que la transacción se comprometa, ya que un fallo del sistema podría dar lugar a la pérdida del mensaje. Comprobar si el mensaje se ha recibido previamente resulta fundamental para evitar que se entregue varias veces.

  En muchos sistemas de mensajería los mensajes se pueden retrasar de manera arbitraria, aunque esos retrasos sean muy improbables. Por tanto, para asegurarse, los mensajes no se deben eliminar nunca de la relación *mensajes_recibidos*. Su eliminación puede hacer que no se detecte una entrega duplicada. Pero, como consecuencia, la relación *mensajes_recibidos* puede crecer de manera indefinida. Para resolver este problema se asigna a cada mensaje una marca temporal y, si la marca temporal del mensaje recibido es más antigua que la de algún punto arbitrario de corte, ese mensaje se descarta. Todos los mensajes registrados en la relación *mensajes_recibidos* que sean más antiguos que el punto de corte se pueden eliminar.

22.5 Control de la concurrencia en las bases de datos distribuidas

En este apartado se muestra el modo en que se pueden modificar algunos de los esquemas de control de concurrencia que se estudian en el Capítulo 16 para utilizarlos en entornos distribuidos. Se da por supuesto que cada sitio participa en la ejecución de un protocolo de compromiso para garantizar la atomicidad global de las transacciones.

Los protocolos que se describen en este apartado necesitan que se hagan actualizaciones de todas las réplicas de los elementos de datos. Si ha fallado algún sitio que contenga una réplica de un elemento de datos, no se pueden procesar las actualizaciones de ese elemento de datos. En el Apartado 22.6 se describen los protocolos que pueden continuar el procesamiento de las transacciones aunque haya fallado algún sitio o algún enlace, lo que proporciona una gran disponibilidad.

22.5.1 Protocolos de bloqueo

Los diferentes protocolos de bloqueo descritos en el Capítulo 16 se pueden utilizar en entornos distribuidos. La única modificación que hay que incorporar es el modo en que el gestor de bloqueos trata los datos replicados. Se presentan varios esquemas posibles que son aplicables a entornos en que los datos se pueden replicar en varios sitios. Al igual que en el Capítulo 16, se dará por supuesto la existencia de los modos de bloqueo *compartido* y *exclusivo*.

22.5.1.1 Enfoque de gestor único de bloqueos

En el enfoque de **gestor único de bloqueos** el sistema mantiene un *único* gestor de bloqueos que reside en un sitio *único* escogido (por ejemplo, S_i). Todas las solicitudes de bloqueo y de desbloqueo se realizan en el sitio S_i. Cuando una transacción necesita bloquear un elemento de datos, envía una solicitud de bloqueo a S_i. El gestor de bloqueos determina si se puede conceder el bloqueo de manera inmediata. En caso afirmativo, envía un mensaje al respecto al sitio en el que se inició la solicitud de bloqueo. En caso contrario, la solicitud se retrasa hasta que se puede conceder, en cuyo momento se envía un mensaje al sitio en el que se inició la solicitud de bloqueo. La transacción puede leer el elemento de datos de *cualquiera* de los sitios en los que residan sus réplicas. En el caso de las operaciones de escritura, deben implicarse todos los sitios en los que residan réplicas del elemento de datos.

El esquema tiene las ventajas siguientes:

- **Implementación sencilla**. Este esquema necesita dos mensajes para tratar las solicitudes de bloqueo y sólo uno para las de desbloqueo.
- **Tratamiento sencillo de los interbloqueos**. Como todas las solicitudes de bloqueo y de desbloqueo se realizan en un solo sitio, se pueden aplicar directamente a este entorno los algoritmos de tratamiento de los interbloqueos estudiados en el Capítulo 16.

Los inconvenientes del esquema son:

- **Cuello de botella**. El sitio S_i se transforma en un cuello de botella, ya que todas las solicitudes deben procesarse allí.
- **Vulnerabilidad**. Si el sitio S_i falla, se pierde el controlador de la concurrencia. O bien hay que detener el procesamiento, o bien hay que utilizar un esquema de recuperación para que un sitio de respaldo pueda asumir la administración de los bloqueos, como se describe en el Apartado 22.6.5.

22.5.1.2 Gestor distribuido de bloqueos

Se puede lograr un compromiso entre las ventajas y los inconvenientes ya mencionados mediante el enfoque del **gestor distribuido de bloqueos**, en el que la función de gestor de bloqueos se halla distribuida entre varios sitios.

Cada sitio mantiene un gestor de bloqueos local, cuya función es administrar las solicitudes de bloqueo y de desbloqueo para los elementos de datos que se almacenan en ese sitio. Cuando una transac-

ción desea bloquear el elemento de datos Q, que no está replicado y reside en el sitio S_i, envía un mensaje al gestor de bloqueos del sitio S_i para solicitarle un bloqueo (en un modo de bloqueo determinado). Si el elemento de datos Q está bloqueado en un modo incompatible, la solicitud se retrasa hasta que se pueda conceder. Una vez se haya determinado que la solicitud de bloqueo se puede conceder, el gestor de bloqueos devuelve un mensaje al sitio que ha iniciado la solicitud para indicar que ha concedido la solicitud de bloqueo.

Existen varios modos alternativos de tratar con la réplica de los elementos de datos, que se estudian en los Apartados 22.5.1.3 a 22.5.1.6.

El esquema del gestor distribuido de bloqueos presenta la ventaja de su sencilla implementación y reduce el grado en el que el coordinador constituye un cuello de botella. Tiene una sobrecarga razonablemente baja, ya que sólo necesita dos transferencias de mensajes para tratar las solicitudes de bloqueo y una transferencia de mensaje para las de desbloqueo. Sin embargo, el tratamiento de los interbloqueos resulta más complejo, dado que las solicitudes de bloqueo y de desbloqueo ya no se realizan en un solo sitio. Puede haber interbloqueos entre sitios aunque no los haya dentro de ninguno de los sitios. Los algoritmos de tratamiento de los interbloqueos estudiados en el Capítulo 16 deben modificarse, como se estudiará en el Apartado 22.5.4 para que detecten los interbloqueos globales.

22.5.1.3 Copia principal

Cuando un sistema utiliza la réplica de datos se puede escoger una de las réplicas como **copia principal**. Así, para cada elemento de datos Q, la copia principal de Q debe residir exactamente en un sitio, que se denomina **sitio principal** de Q.

Cuando una transacción necesita bloquear el elemento de datos Q, solicita un bloqueo en el sitio principal de Q. Como ya se ha visto, la respuesta a la solicitud se retrasa hasta que pueda concederse.

Por tanto, la copia principal permite que el control de concurrencia de los datos replicados se trate como el de los datos no replicados. Esta semejanza permite una implementación sencilla. No obstante, si falla el sitio principal de Q, ese elemento de datos queda inaccesible, aunque otros sitios que contengan réplicas suyas estén accesibles.

22.5.1.4 Protocolo de mayoría

El **protocolo de mayoría** funciona de la manera siguiente: si el elemento de datos Q se replica en n sitios diferentes, hay que enviar un mensaje de solicitud de bloqueo a más del cincuenta por ciento de esos sitios. Cada gestor de bloqueos determina si se puede conceder el bloqueo de manera inmediata (en lo que a él se refiere). Como ya se ha visto, la respuesta se retrasa hasta que la solicitud se pueda conceder. La transacción no se lleva a cabo en Q hasta que logre obtener un bloqueo sobre la mayoría de las réplicas de Q.

Supóngase por ahora que las operaciones de escritura se llevan a cabo en todas las réplicas, lo que exige que todos los sitios que las contienen estén disponibles. No obstante, la principal ventaja del protocolo de mayoría es que puede extenderse para que trate los fallos de los sitios, como se verá en el Apartado 22.6.1. Este protocolo también trata los datos replicados de manera descentralizada, con lo que evita los inconvenientes del control centralizado. Sin embargo, presenta los siguientes inconvenientes:

- **Implementación**. El protocolo de mayoría es más complicado de implementar que los esquemas anteriores. Necesita, como mínimo, $2(n/2 + 1)$ mensajes para manejar las solicitudes de bloqueo y, al menos, $(n/2 + 1)$ mensajes para manejar las solicitudes de desbloqueo.
- **Tratamiento de los interbloqueos**. Además del problema de los interbloqueos globales debidos al empleo del enfoque del gestor distribuido de bloqueos, puede que se produzcan interbloqueos aunque sólo se esté bloqueando un elemento de datos. A modo de ejemplo, considérese un sistema con cuatro sitios y réplica completa. Supóngase que las transacciones T_1 y T_2 desean bloquear el elemento de datos Q en modo exclusivo. Puede que la transacción T_1 tenga éxito en el bloqueo de Q en los sitios S_1 y S_3 y la transacción T_2 consiga bloquear Q en los sitios S_2 y S_4. Cada una de ellas deberá esperar a adquirir el tercer bloqueo; por tanto, se ha producido un interbloqueo. Por

fortuna, estos interbloqueos se pueden evitar con relativa facilidad exigiendo que todos los sitios soliciten el bloqueo de las réplicas de cada elemento de datos en el mismo orden predeterminado.

22.5.1.5 Protocolo sesgado

El **protocolo sesgado** es otro enfoque del manejo de las réplicas. La diferencia con el protocolo de mayoría es que se concede un tratamiento más favorable a las solicitudes de bloqueo compartido que a las solicitudes de bloqueo exclusivo.

- **Bloqueos compartidos**. Cuando una transacción necesita bloquear el elemento de datos Q, simplemente solicita su bloqueo al gestor de bloqueos de un sitio que contenga una réplica de Q.
- **Bloqueos exclusivos**. Cuando una transacción necesita bloquear el elemento de datos Q, solicita su bloqueo al gestor de bloqueos de todos los sitios que contienen una réplica de Q.

Al igual que antes, la respuesta a la solicitud se retrasa hasta que pueda concederse.

El esquema sesgado tiene la ventaja de imponer menos sobrecarga a las operaciones leer que el protocolo de mayoría. Este ahorro resulta especialmente significativo en el frecuente caso en que la frecuencia de las operaciones leer es mucho mayor que la de las operaciones escribir. No obstante, la sobrecarga adicional sobre las operaciones de escritura supone un inconveniente. Además, el protocolo sesgado comparte con el protocolo de mayoría el inconveniente de la complejidad en el manejo de los interbloqueos.

22.5.1.6 Protocolo de consenso de quórum

El protocolo de **consenso de quórum** es una generalización del protocolo de mayoría. El protocolo de consenso de quórum asigna a cada sitio un peso no negativo. Asigna a las operaciones de lectura y de escritura sobre el elemento x dos enteros, denominados **quórum de lectura** Q_l y **quórum de escritura** Q_e, que deben cumplir la siguiente condición, donde S es el peso total de todos los sitios en los que x reside:

$$Q_l + Q_e > S \textbf{ y } 2 * Q_e > S$$

Para ejecutar una operación de lectura deben bloquearse suficientes réplicas como para que su peso total sea $\geq Q_l$. Para ejecutar una operación de escritura se deben bloquear suficientes réplicas como para que su peso total sea $\geq Q_e$.

Una ventaja del enfoque de consenso de quórum es que puede permitir la reducción selectiva del coste de los bloqueos de lectura o de escritura mediante la adecuada definición de los quórum de lectura y de escritura. Por ejemplo, con un quórum de lectura pequeño, las operaciones de lectura necesitan obtener menos bloqueos, pero el quórum de escritura será mayor, por lo que las operaciones de escritura necesitarán obtener más bloqueos. Además, si se asignan pesos más elevados a algunos sitios (por ejemplo, a los que tengan menos posibilidad de fallar), hace falta acceder a menos sitios para adquirir los bloqueos. De hecho, si se definen los pesos y los quórum de manera adecuada, el protocolo de consenso de quórum puede simular tanto el protocolo de mayoría como el sesgado.

Al igual que el protocolo de mayoría, el de consenso de quórum se puede extender para que trabaje incluso en caso de fallos de los sitios, como se verá en el Apartado 22.6.1.

22.5.2 Marcas temporales

La idea principal tras el esquema de marcas temporales del Apartado 16.2 es que se concede a cada transacción una marca temporal *única* que el sistema utiliza para decidir el orden de secuenciación. La primera tarea, por tanto, al generalizar el esquema centralizado a uno distribuido es desarrollar un esquema para la generación de marcas temporales únicas. Por tanto, los diferentes protocolos pueden operar directamente en el entorno no replicado.

Existen dos métodos principales para la generación de marcas temporales únicas, uno centralizado y otro distribuido. En el esquema centralizado un solo sitio distribuye las marcas temporales. El sitio puede utilizar para ello un contador lógico o su propio reloj local.

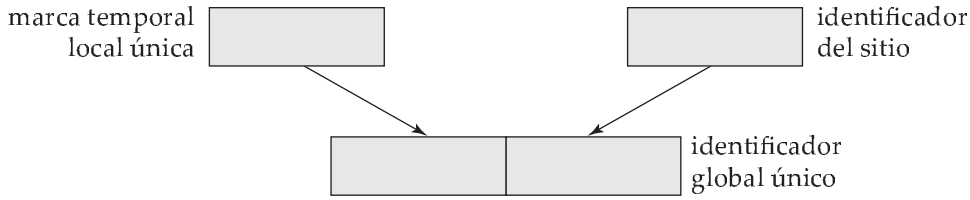


Figura 22.2 Generación de marcas temporales únicas.

En el esquema distribuido cada sitio genera una marca temporal local única mediante un contador lógico o su reloj local. La marca temporal global única se obtiene concatenando la marca temporal local única correspondiente con el identificador de ese sitio, que también debe ser único (Figura 22.2). El orden de concatenación es importante. El identificador del sitio se utiliza en la posición menos significativa para garantizar que las marcas temporales globales generadas en un sitio dado no sean siempre mayores que las generadas en otro. Compárese esta técnica para la generación de marcas temporales únicas con la presentada en el Apartado 22.2.3 para la generación de nombres únicos.

Puede que todavía surja algún problema si un sitio genera marcas temporales locales a una velocidad mayor que los demás sitios. En ese caso el contador lógico del sitio rápido será mayor que el de los demás sitios. Por tanto, todas las marcas temporales generadas por el sitio rápido serán mayores que las generadas por los demás sitios. Lo que se necesita es un mecanismo que garantice que las marcas temporales locales se generen de manera homogénea en todo el sistema. En cada sitio S_i se define un **reloj lógico** (RL_i), que genera la marca temporal local única. El reloj lógico puede implementarse como un contador que se incremente después de generar cada nueva marca temporal local. Para garantizar que los diferentes relojes lógicos estén sincronizados, se exige que el sitio S_i adelante su reloj lógico siempre que una transacción *Ti* con la marca temporal <*x*,*y*> visite ese sitio y x sea mayor que el valor actual de RL_i. En ese caso, el sitio S_i adelantará su reloj lógico hasta el valor $x + 1$.

Si se utiliza el reloj del sistema para generar las marcas temporales, éstas se asignarán de manera homogénea, siempre que ningún sitio tenga un reloj del sistema que adelante o atrase. Dado que es posible que los relojes no sean totalmente exactos, hay que utilizar una técnica parecida a la de los relojes lógicos para garantizar que ningún reloj se adelante o se atrase mucho respecto de los demás.

22.5.3 Réplica con grado de consistencia bajo

Muchas bases de datos comerciales actuales soportan las réplicas, que pueden adoptar varias formas. Con la **réplica maestro–esclavo**, la base de datos permite las actualizaciones en el sitio principal y las propaga de manera automática a las réplicas de los demás sitios. Las transacciones pueden leer las réplicas en los demás sitios, pero no se les permite actualizarlas.

Una característica importante de esta réplica es que las transacciones no consiguen bloqueos en los sitios remotos. Para garantizar que las transacciones que se ejecutan en los sitios de réplica vean una vista consistente (aunque quizás desactualizada) de la base de datos la réplica debe reflejar una **instantánea consistente para las transacciones** de los datos del sitio principal, es decir, la réplica debe reflejar todas las actualizaciones de las transacciones hasta una transacción dada según el orden de secuenciación, y no deben reflejar ninguna actualización de transacciones posteriores según el orden de secuenciación.

Se puede configurar la base de datos para que propague las actualizaciones de manera inmediata, una vez producidas en el sitio principal, o para que las propague sólo de manera periódica.

La réplica maestro–esclavo resulta especialmente útil para distribuir información, por ejemplo, desde una oficina central a las sucursales de una organización. Otra aplicación de esta forma de réplica es la creación de copias de la base de datos para la ejecución de consultas de gran tamaño, de modo que las consultas no interfieran con las transacciones. Las actualizaciones deben propagarse de manera periódica (cada noche, por ejemplo), de modo que la propagación no interfiera con el procesamiento de las consultas.

El sistema de bases de datos Oracle posee la sentencia **create snapshot** (crear instantánea), que puede crear en un sitio remoto una copia instantánea de una relación, o de un conjunto de relaciones, consistente para las transacciones. También soporta la actualización de las instantáneas, que puede hacerse

volviendo a calcular cada instantánea o actualizándola de manera incremental. Oracle soporta la actualización automática, tanto continua como a intervalos periódicos.

Con la **réplica multimaestro** (también denominada **réplica de actualización distribuida**) las actualizaciones se permiten en cualquier réplica de cada elemento de datos y se propagan de manera automática a todas las réplicas. Este modelo es el modelo básico utilizado para administrar las réplicas en las bases de datos distribuidas. Las transacciones actualizan la copia local y el sistema actualiza las demás réplicas de manera transparente.

Un modo de actualizar las réplicas es aplicar la actualización inmediata con compromiso de dos fases, utilizando una de las técnicas de control de concurrencia distribuida que se han visto. Muchos sistemas de bases de datos utilizan el protocolo sesgado, en el que las operaciones de escritura tienen que bloquear y actualizar todas las réplicas y las operaciones de lectura bloquean y leen cualquier réplica, como técnica de control de concurrencia.

Muchos sistemas de bases de datos ofrecen una forma alternativa de actualización: actualizan en un sitio dado, con **propagación perezosa** de las actualizaciones a los demás sitios, en lugar de aplicar de manera inmediata las actualizaciones a todas las réplicas como parte de la transacción que lleva a cabo la actualización. Los esquemas basados en la propagación perezosa permiten que continúe el procesamiento de las transacciones (incluidas las actualizaciones) aunque algún sitio quede desconectado de la red, lo que mejora la disponibilidad, pero, por desgracia, lo hacen a costa de la consistencia. Se suele seguir uno de estos dos enfoques cuando se emplea la propagación perezosa:

- Las actualizaciones de las réplicas se traducen en actualizaciones del sitio principal, que luego se propagan de manera perezosa a todas las réplicas.

  Este enfoque garantiza que las actualizaciones de cada elemento se ordenen de manera secuencial, aunque puedan producirse problemas de secuenciabilidad, ya que puede que las transacciones lean un valor antiguo de algún otro elemento de datos y lo utilicen para llevar a cabo una actualización.

- Las actualizaciones se llevan a cabo en cualquier réplica y se propagan a todas las demás.

  Este enfoque puede provocar todavía más problemas, ya que el mismo elemento de datos puede ser actualizado de manera concurrente en varios sitios.

Algunos conflictos debidos a la falta de control de concurrencia distribuida pueden detectarse cuando las actualizaciones se propagan a otros sitios (se verá el modo de hacerlo en el Apartado 24.5.4), pero la resolución del conflicto implica hacer retroceder transacciones comprometidas y, por tanto, no se garantiza la durabilidad de las transacciones comprometidas. Además, puede que se necesite la intervención del personal encargado para que resuelva el conflicto. Por tanto, los esquemas mencionados deben evitarse o utilizarse con precaución.

22.5.4 Tratamiento de los interbloqueos

Los algoritmos de prevención y de detección de interbloqueos del Capítulo 16 pueden utilizarse en los sistemas distribuidos, siempre que se realicen modificaciones. Por ejemplo, se puede utilizar el protocolo de árbol si se define un árbol *global* entre los elementos de datos del sistema. De manera parecida, el enfoque de ordenación por marcas temporales puede aplicarse de manera directa en entornos distribuidos, como se vio en el Apartado 22.5.2.

La prevención de interbloqueos puede dar lugar a esperas y retrocesos innecesarios. Además, puede que algunas técnicas de prevención de interbloqueos necesiten que se impliquen en la ejecución de cada transacción más sitios de los que serían necesarios de otro modo.

Si se permite que los interbloqueos se produzcan y se confía en su detección, el problema principal en los sistemas distribuidos es decidir el modo en que se mantiene el grafo de espera. Las técnicas habituales para tratar este problema exigen que cada sitio guarde un **grafo local de espera**. Los nodos del grafo en un momento dado se corresponden con todas las transacciones (locales y no locales) que en ese momento tienen o solicitan alguno de los elementos locales de ese sitio. Por ejemplo, la Figura 22.3 muestra un sistema que consta de dos sitios, cada uno de los cuales mantiene su propio grafo

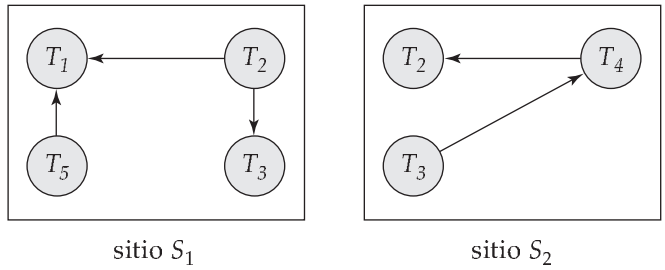


Figura 22.3 Grafos locales de espera.

local de espera. Obsérvese que las transacciones T_2 y T_3 aparecen en los dos grafos, lo que indica que han solicitado elementos en los dos sitios.

Estos grafos locales de espera se crean de la manera habitual para las transacciones y los elementos de datos locales. Cuando la transacción T_i del sitio S_1 necesita un recurso del sitio S_2, envía un mensaje de solicitud al sitio S_2. Si el recurso lo tiene la transacción T_j, el sistema introduce el arco $T_i \rightarrow T_j$ en el grafo de espera local del sitio S_2.

Evidentemente, si algún grafo de espera local tiene un ciclo, se ha producido un interbloqueo. Por otro lado, el hecho de que no haya ciclos en ninguno de los grafos locales de espera no significa que no haya interbloqueos. Para ilustrar este problema, considérense los grafos locales de espera de la Figura 22.3. Cada grafo de espera es acíclico y, sin embargo, hay un interbloqueo en el sistema debido a que la *unión* de los grafos locales de espera contiene un ciclo. Este grafo aparece en la Figura 22.4.

En el enfoque de **detección centralizada de interbloqueos** el sistema crea y mantiene un **grafo global de espera** (la unión de todos los grafos locales) en un *solo* sitio: el coordinador de detección de interbloqueos. Dado que hay un retraso en las comunicaciones en el sistema, hay que distinguir entre dos tipos de grafos de espera. Los grafos *reales* describen el estado real pero desconocido del sistema en un momento dado, como lo vería un observador omnisciente. Los grafos *creados* son una aproximación generada por el controlador durante la ejecución de su algoritmo. Evidentemente, el controlador debe generar el grafo creado de modo que, siempre que se invoque al algoritmo de detección, los resultados obtenidos sean correctos. *Correcto* significa en este caso que, si hay algún interbloqueo, se comunique con prontitud y, si el sistema comunica algún interbloqueo, realmente se halle en estado de interbloqueo.

El grafo global de espera debe poder volver a crearse o a actualizarse bajo las siguientes condiciones:

- Siempre que se introduzca o se elimine un nuevo arco en alguno de los grafos locales de espera.
- De manera periódica, cuando se hayan producido varias modificaciones en los grafos locales de espera.
- Siempre que el coordinador necesite invocar el algoritmo de detección de ciclos.

Cuando el coordinador invoca el algoritmo de detección de interbloqueos, busca en su grafo global. Si halla un ciclo, selecciona una víctima para hacer que retroceda. El coordinador debe comunicar a todos los sitios que se ha seleccionado como víctima a una transacción concreta. Los sitios, a su vez, hacen retroceder la transacción víctima.

Este esquema puede producir retrocesos innecesarios si:

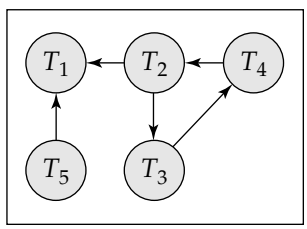


Figura 22.4 Grafo global de espera de la Figura 22.3.

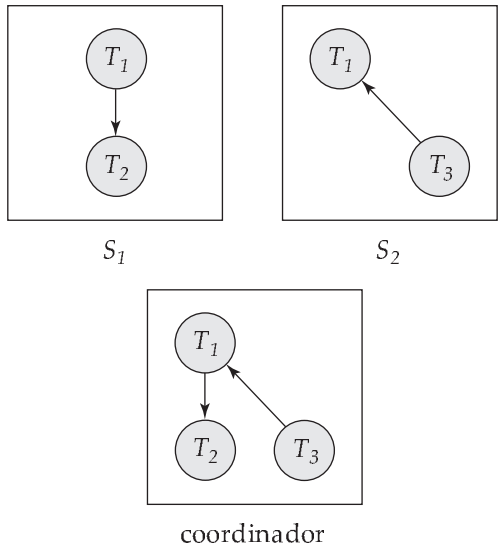


Figura 22.5 Ciclos falsos en el grafo global de espera.

- Existen **ciclos falsos** en el grafo global de espera. A modo de ejemplo, considérese una instantánea del sistema representado por los grafos locales de espera de la Figura 22.5. Supóngase que T_2 libera el recurso que tiene en el sitio S_1, lo que provoca la eliminación del arco $T_1 \rightarrow T_2$ de S_1. La transacción T_2 solicita entonces un recurso que tiene T_3 en el sitio S_2, lo que da lugar a la agregación del arco $T_2 \rightarrow T_3$ en S_2. Si el mensaje insertar $T_2 \rightarrow T_3$ llega antes que el mensaje eliminar $T_1 \rightarrow T_2$ de S_1, puede que el coordinador descubra el ciclo falso $T_1 \rightarrow T_2 \rightarrow T_3$ después del mensaje insertar (pero antes del mensaje eliminar). Puede que se inicie la recuperación de interbloqueos, aunque no se haya producido ninguno.

  Obsérvese que la situación con ciclos falsos no se puede producir con bloqueos de dos fases. La probabilidad de aparición de ciclos falsos suele ser lo bastante baja como para que no generen un problema serio de rendimiento.

- Se produce realmente un *interbloqueo* y se escoge una víctima cuando resulta que se ha abortado alguna de las transacciones por motivos no relacionados con el interbloqueo. Por ejemplo, supóngase que el sitio S_1 de la Figura 22.3 decide abortar T_2. Al mismo tiempo, el coordinador ha descubierto un ciclo y ha escogido como víctima a T_3. Se hace que retrocedan tanto T_2 como T_3, aunque sólo sería necesario hacer que retrocediera T_2.

La detección de interbloqueos puede hacerse de manera distribuida, con varios sitios que asuman partes de la tarea, en lugar de hacerla en un solo sitio. No obstante, los algoritmos correspondientes resultan más complicados y costosos. Véanse las notas bibliográficas para hallar referencias a esos algoritmos

22.6 Disponibilidad

Uno de los objetivos del empleo de bases de datos distribuidas es disfrutar de una **disponibilidad elevada**; es decir, la base de datos debe funcionar casi todo el tiempo. En concreto, dado que los fallos son más probables en los sistemas distribuidos de gran tamaño, las bases de datos distribuidas deben seguir funcionando aunque sufran diferentes tipos de fallos. La posibilidad de continuar funcionando incluso durante los fallos se denomina **robustez**.

Para que un sistema distribuido sea robusto debe *detectar* los fallos, *reconfigurar* el sistema de modo que el cálculo pueda continuar y *recuperarse* cuando se repare el procesador o el enlace.

Los diferentes tipos de fallos se tratan de manera diferente. Por ejemplo, la pérdida de mensajes se trata mediante su retransmisión. La retransmisión repetida de un mensaje por un enlace, sin la recepción de un acuse de recibo, suele ser síntoma de un fallo de ese enlace. La red intentará hallar una ruta

alternativa para el mensaje. La imposibilidad de hallar esa ruta suele ser síntoma de una división de la red.

No obstante, no suele ser posible diferenciar claramente entre los fallos de los sitios y las divisiones de la red. El sistema, generalmente, puede detectar que se ha producido un fallo, pero puede que no logre identificar su tipo. Por ejemplo, supóngase que el sitio S_1 no puede comunicar con S_2. Puede ser que S_2 haya fallado. No obstante, otra posibilidad es que el enlace entre S_1 y S_2 haya fallado, lo que habrá provocado la división de la red. El problema se aborda en parte empleando varios enlaces entre los diferentes sitios, de modo que, aunque falle algún enlace, sigan conectados. Sin embargo, todavía pueden fallar varios enlaces simultáneamente, por lo que hay situaciones en las que no se puede estar seguro de si se ha producido un fallo del sitio o una división de la red.

Supóngase que el sitio S_1 ha descubierto que se ha producido un fallo. Debe iniciar un procedimiento que permita que el sistema se reconfigure y continúe con el modo normal de operación.

- Si en el momento del fallo había transacciones activas en un sitio que haya fallado o que haya quedado inaccesible, hay que abortar esas transacciones. Resulta conveniente abortarlas cuanto antes, ya que puede que tengan bloqueos sobre datos de sitios que sigan activos; puede que esperar a que el sito que ha fallado o que ha quedado inaccesible vuelva a estar accesible impida otras transacciones en sitios que están operativos.

  No obstante, en algunos casos, cuando los objetos de datos están replicados, puede que sea posible seguir adelante con las operaciones de lectura y de actualización aunque algunas réplicas estén inaccesibles. En ese caso, cuando se recupera el sitio que ha fallado, si tenía réplicas de algún objeto de datos, debe obtener los valores actualizados de esos objetos de datos y asegurarse de que recibe todas las actualizaciones posteriores. Este problema se aborda en el Apartado 22.6.1.

- Si los datos replicados se guardan en un sitio que ha fallado o que está inaccesible, hay que actualizar el catálogo para que las consultas no hagan referencia a la copia ubicada en ese sitio. Cuando el sitio vuelva a estar activo, hay que asegurarse de que los datos que alberga sean consistentes, como se verá en el Apartado 22.6.3.

- Si el sitio que ha fallado es un servidor central de algún subsistema, hay que celebrar una *elección* para determinar el nuevo servidor (véase el Apartado 22.6.5). Entre los servidores centrales están los servidores de nombres, los coordinadores de concurrencia y los detectores globales de interbloqueos.

Dado que, en general, no es posible distinguir entre los fallos de los enlaces de red y los de los sitios, hay que diseñar todos los esquemas de reconfiguración para que funcionen de manera correcta en caso de división de la red. En concreto, deben evitarse las situaciones siguientes:

- Que se elijan dos o más servidores centrales en particiones distintas.

- Que más de una partición actualice un mismo elemento de datos replicado.

22.6.1 Enfoque basado en la mayoría

Se puede modificar el enfoque basado en la mayoría del control distribuido de la concurrencia del Apartado 22.5.1.4 para que funcione a pesar de los fallos. En este enfoque, cada objeto de datos guarda con él un número de versión para detectar el momento en que se escribió en él por última vez. Siempre que una transacción escribe un objeto, actualiza también su número de versión de la manera siguiente:

- Si el objeto de datos *a* se replica en *n* sitios diferentes, se debe enviar un mensaje de solicitud de bloqueo a más de la mitad de los *n* sitios en los que se guarda *a*. La transacción no opera sobre *a* hasta que ha conseguido obtener un bloqueo en la mayoría de las réplicas de *a*.

- Las operaciones de lectura examinan todas las réplicas sobre las que se ha obtenido el bloqueo y leen el valor de la réplica que tenga el número de versión más elevado (de manera opcional, también pueden escribir ese valor en las réplicas con números de versión más bajos). Las operaciones de escritura leen todas las réplicas, igual que hacen las operaciones de lectura, para hallar el número de versión más elevado (normalmente, una operación de lectura habrá llevado a cabo antes este paso de la transacción, y el resultado obtenido se puede volver a utilizar). El nuevo

número de versión es una unidad mayor que el número de versión más elevado encontrado. La operación de escritura escribe en todas las réplicas sobre las que ha obtenido bloqueos y da al número de versión de todas las réplicas el valor del nuevo número de versión.

Los fallos durante las transacciones (tanto las divisiones de la red como los fallos de los sitios) se pueden tolerar siempre que (1) los sitios disponibles en el momento del compromiso contengan la mayoría de las réplicas de todos los objetos en los que se ha escrito y (2) durante las operaciones de lectura se lea la mayoría de las réplicas para averiguar su número de versión. Si se violan estos requisitos, hay que abortar la transacción. Siempre que se satisfagan estos requisitos, se puede utilizar el protocolo de compromiso de dos fases, como siempre, en los sitios que estén disponibles.

En este esquema la reintegración resulta trivial; no hay que hacer nada. Esto se debe a que las operaciones de escritura habrían actualizado la mayoría de las réplicas, mientras las operaciones de lectura leerán la mayoría de las réplicas y hallarán, como mínimo, una que tenga la última versión.

La técnica de numeración de las versiones utilizada con el protocolo de mayoría se puede utilizar también para hacer que funcione el protocolo de consenso de quórum en presencia de fallos. Los detalles (evidentes) se dejan al lector. No obstante, el riesgo de que los fallos eviten que el sistema procese las transacciones aumenta si se asignan pesos superiores a algunos sitios.

22.6.2 Enfoque leer uno, escribir todos los disponibles

Como caso especial de consenso de quórum se puede emplear el protocolo sesgado, si se asignan pesos unitarios a todos los sitios, se define el quórum de lectura como 1 y se define el quórum de escritura como n (todos los sitios). En este caso especial no hace falta utilizar números de versión; sin embargo, con que falle un solo sitio que contenga un elemento de datos, no podrá llevarse a cabo ninguna operación de escritura en ese elemento, ya que no se dispondrá del quórum de escritura. Este protocolo se denomina protocolo **leer uno, escribir todos**, ya que hay que escribir todas las réplicas.

Para permitir que el trabajo continúe en caso de fallos, sería deseable poder utilizar el protocolo **leer uno, escribir todos los disponibles**. En este enfoque las operaciones de lectura se llevan a cabo como en el esquema **leer uno, escribir todos**; se puede leer cualquier réplica disponible, y sobre ella se obtiene un bloqueo de lectura. Se envía una operación de escritura a todas las réplicas, y se adquieren bloqueos de escritura sobre todas ellas. Si algún sitio no está disponible, el gestor de transacciones continúa su labor sin esperar a que se recupere.

Aunque este enfoque parezca muy atractivo, presenta varias complicaciones. En concreto, los fallos de comunicación temporales pueden hacer que un sitio parezca no disponible, lo que hace que la operación de escritura no se lleve a cabo pero, cuando el enlace se restaura, el sitio no sabe que tiene que llevar a cabo acciones de reintegración para ponerse al día con las operaciones de escritura que se ha perdido. Además, si la red se divide, puede que cada partición actualice el mismo elemento de datos, creyendo que los sitios de las demás particiones no funcionan.

El esquema leer uno, escribir todos los disponibles puede utilizarse si nunca se producen divisiones de la red, pero puede dar lugar a inconsistencias en caso de que se produzcan.

22.6.3 Reintegración de los sitios

La reintegración al sistema de los sitios o enlaces reparados exige la adopción de precauciones. Cuando un sitio que ha fallado se recupera, debe iniciar un procedimiento para actualizar sus tablas de sistema y que reflejen las modificaciones acaecidas mientras estaba fuera de servicio. Si el sitio tiene réplicas de elementos de datos, debe obtener sus valores actualizados y asegurarse de que recibe todas las actualizaciones que se produzcan a partir de ese momento. La reintegración de los sitios es más complicada de lo que parece a primera vista, ya que puede que haya actualizaciones de los elementos de datos que se procesen durante el tiempo en el que el sitio se está recuperando.

Una solución sencilla es detener temporalmente todo el sistema hasta que el sitio que ha fallado vuelva a estar activo. En la mayor parte de las aplicaciones, sin embargo, esa detención temporal plantea problemas inaceptables. Se han desarrollado técnicas para permitir que los sitios que han fallado se reintegren mientras se ejecutan de manera concurrente las actualizaciones de los elementos de datos. Antes de que se conceda ningún bloqueo de lectura o escritura sobre algún elemento de datos, el sitio

debe asegurarse de que se ha puesto al día con todas las actualizaciones de ese elemento de datos. Si se recupera un enlace que había fallado, se pueden volver a unir dos o más divisiones de la red. Dado que la división de la red limita las operaciones admisibles para algunos de los sitios, o para todos ellos, hay que informar con prontitud a todos los sitios de la recuperación de ese enlace. Véanse las notas bibliográficas para obtener más información sobre la recuperación en los sistemas distribuidos.

22.6.4 Comparación con la copia de seguridad remota

Los sistemas remotos de copia de seguridad, que se estudiaron en el Apartado 17.9, y la réplica en las bases de datos distribuidas son dos enfoques alternativos para la provisión de una disponibilidad elevada. La diferencia principal entre los dos esquemas es que, con los sistemas remotos de copia de seguridad, las acciones como el control de concurrencia y la recuperación se llevan a cabo en un único sitio, y sólo se replican en el otro sitio los datos y los registros del registro histórico. En concreto, los sistemas remotos de copia de seguridad ayudan a evitar el compromiso de dos fases, y las sobrecargas resultantes. Además, las transacciones sólo tienen que entrar en contacto con un sitio (el sitio principal) y, así, se evita la sobrecarga de la ejecución del código de las transacciones en varios sitios. Por tanto, los sistemas remotos de copia de seguridad ofrecen un enfoque de la elevada disponibilidad de menor coste que las réplicas.

Por otro lado, la réplica puede ofrecer mayor disponibilidad al tener disponibles varias réplicas y utilizar el protocolo de mayoría.

22.6.5 Selección del coordinador

Varios de los algoritmos que se han presentado exigen el empleo de un coordinador. Si el coordinador falla por problemas en el sitio en el que reside el sistema, el sistema sólo puede continuar la ejecución reiniciando un nuevo coordinador en otro sitio. Un modo de continuar la ejecución es mantener un coordinador suplente, que esté preparado para asumir la responsabilidad si el coordinador falla.

El **coordinador suplente** es un sitio que, además de otras tareas, mantiene de manera local suficiente información como para poder asumir el papel de coordinador con un perjuicio mínimo al sistema distribuido. Tanto el coordinador como su suplente reciben todos los mensajes dirigidos al coordinador. El coordinador suplente ejecuta los mismos algoritmos y mantiene la misma información interna de estado (como, por ejemplo, para el coordinador de concurrencia, la tabla de bloqueos) que el coordinador auténtico. La única diferencia en funcionamiento entre el coordinador y su suplente es que el suplente no emprende ninguna acción que afecte a otros sitios. Esas acciones se dejan al coordinador auténtico.

En caso de que el coordinador suplente detecte el fallo del coordinador auténtico, asume el papel de coordinador. Dado que el suplente dispone de toda la información que tenía el coordinador que ha fallado, el procesamiento puede continuar sin interrupción.

La ventaja principal del enfoque del suplente es la posibilidad de continuar el procesamiento de manera inmediata. Si no hubiera un suplente dispuesto a asumir la responsabilidad del coordinador, el coordinador que se designara ex novo tendría que buscar la información en todos los sitios del sistema para poder ejecutar las tareas de coordinación. Con frecuencia, la única fuente de parte de la información necesaria es el coordinador que ha fallado. En ese caso, puede que sea necesario abortar parte de las transacciones activas (o todas ellas) y reiniciarlas bajo el control del nuevo coordinador.

Por tanto, el enfoque del coordinador suplente evita retrasos sustanciales mientras el sistema distribuido se recupera del fallo del coordinador. El inconveniente es la sobrecarga de la ejecución duplicada de las tareas del coordinador. Además, el coordinador y su suplente necesitan comunicarse de manera regular para asegurarse de que sus actividades están sincronizadas.

En resumen, el enfoque del coordinador suplente supone una sobrecarga durante el procesamiento normal para permitir una recuperación rápida de los fallos del coordinador.

A falta de un coordinador suplente designado, o con objeto de tratar varios fallos, los sitios que siguen funcionando pueden escoger de manera dinámica un nuevo coordinador. Los **algoritmos de elección** permiten que los sitios escojan el sitio del nuevo coordinador de manera descentralizada. Los algoritmos de selección necesitan que se asocie un número de identificación único con cada sitio activo del sistema.

El **algoritmo de acoso** para la elección funciona de la manera siguiente. Para no complicar la notación ni la discusión, supóngase que el número de identificación del sitio S_i es i y que el coordinador elegido

siempre será el sitio activo con el número de identificación más elevado. Por tanto, cuando un coordinador falla, el algoritmo debe elegir el sitio activo que tenga el número de identificación más elevado. El algoritmo debe enviar ese número a cada sitio activo del sistema. Además, el algoritmo debe proporcionar un mecanismo por el que los sitios que se recuperen de un fallo puedan identificar al coordinador activo. Supóngase que el sitio S_i envía una solicitud que el coordinador no responde dentro del intervalo de tiempo predeterminado T. En esa situación se supone que el coordinador ha fallado y S_i intenta elegirse a sí mismo como sitio del nuevo coordinador.

El sitio S_i envía un mensaje de elección a cada sitio que tenga un número de identificación más elevado. Luego espera, un intervalo de tiempo T, la respuesta de cualquiera de esos sitios. Si no recibe respuesta dentro del tiempo T, da por supuesto que todos los sitios con números mayores que i han fallado, se elige a sí mismo sitio del nuevo coordinador y envía un mensaje para informar a todos los sitios activos con números de identificación menores de que i de que es el sitio en el que reside el nuevo coordinador.

Si S_i recibe respuesta, comienza un intervalo de tiempo T' para recibir un mensaje que lo informe de que se ha elegido un sitio con un número de identificación más elevado (algún otro sitio se está eligiendo coordinador, y debe comunicar los resultados dentro del tiempo T'). Si S_i no recibe ningún mensaje antes de T', da por supuesto que el sitio con el número más elevado ha fallado y vuelve a iniciar el algoritmo.

Después de que un sitio que ha fallado se haya recuperado, comienza de inmediato la ejecución de ese mismo algoritmo. Si no hay ningún sitio activo con un número más elevado, el sitio que se ha recuperado obliga a todos los sitios con números más bajos a permitirle transformarse en el sitio coordinador, aunque ya haya un coordinador activo con un número más bajo. Por este motivo, al algoritmo se le denomina algoritmo de *acoso*. Si la red se divide, el algoritmo de acoso elige un coordinador diferente en cada partición; para garantizar que, a lo sumo, se elige un coordinador, los sitios que ganan deben comprobar también que la mayoría de los sitios se halla en su partición.

22.7 Procesamiento distribuido de consultas

En el Capítulo 14 se estudió que existen una gran variedad de métodos para el cálculo de la respuesta a una consulta. Se examinaron varias técnicas para escoger una estrategia de procesamiento de consultas que minimice el tiempo que se tarda en calcular la respuesta. Para los sistemas centralizados el criterio principal para medir el coste de una estrategia dada es el número de accesos a disco. En los sistemas distribuidos hay que tener en cuenta varios asuntos más, entre los que se incluyen:

- El coste de la transmisión de los datos por la red.
- La posible ganancia de rendimiento al hacer que varios sitios procesen en paralelo diferentes partes de la consulta.

El coste relativo de la transferencia de datos por la red y del intercambio de datos con el disco varía ampliamente en función del tipo de red y de la velocidad de los discos. Por tanto, en general, uno no se puede centrar exclusivamente en los costes de disco ni en los de la red. Más bien hay que hallar un buen equilibrio entre los dos.

22.7.1 Transformación de consultas

Considérese una consulta extremadamente sencilla: "Hallar todas las tuplas de la relación *cuenta*". Aunque la consulta es sencilla (en realidad, trivial), su procesamiento no es trivial, ya que puede que la relación *cuenta* esté fragmentada, replicada o ambas cosas, como se vio en el Apartado 22.2. Si la relación *cuenta* está replicada, hay que elegir la réplica. Si las réplicas no se han dividido, se escoge aquélla para la que el coste de transmisión sea mínimo. Sin embargo, si alguna réplica se ha dividido, la elección no resulta tan sencilla, ya que hay que calcular varias reuniones o uniones para reconstruir la relación *cuenta*. En ese caso, el número de estrategias para el sencillo ejemplo escogido puede ser elevado. Puede que la optimización de las consultas mediante la enumeración exhaustiva de todas las estrategias alternativas no resulte práctica en esas situaciones.

La transparencia de la fragmentación implica que los usuarios pueden escribir una consulta como

$$\sigma_{nombre_sucursal = \text{“Guadarrama”}}(cuenta)$$

Ya que *cuenta* está definida como

$$cuenta_1 \cup cuenta_2$$

la expresión que resulta del esquema de traducción de nombres es

$$\sigma_{nombre_sucursal = \text{“Guadarrama”}}(cuenta_1 \cup cuenta_2)$$

Esta expresión se puede simplificar de manera automática mediante las técnicas de optimización de consultas del Capítulo 14. El resultado es la expresión

$$\sigma_{nombre_sucursal = \text{“Guadarrama”}}(cuenta_1) \cup \sigma_{nombre_sucursal = \text{“Guadarrama”}}(cuenta_2)$$

la cual incluye dos subexpresiones. La primera sólo implica a $cuenta_1$ y, por tanto, puede evaluarse en el sitio Guadarrama. La segunda sólo implica a $cuenta_2$ y, por tanto, puede evaluarse en el sitio Cercedilla.

Hay otra optimización más que puede hacerse al evaluar

$$\sigma_{nombre_sucursal = \text{“Guadarrama”}}(cuenta_1)$$

Dado que $cuenta_1$ sólo contiene tuplas correspondientes a la sucursal de Guadarrama, se puede eliminar la operación de selección. Al evaluar

$$\sigma_{nombre_sucursal = \text{“Guadarrama”}}(cuenta_2)$$

se puede aplicar la definición del fragmento de $cuenta_2$ para obtener

$$\sigma_{nombre_sucursal = \text{“Guadarrama”}}(\sigma_{nombre_sucursal = \text{“Cercedilla”}}(cuenta))$$

Esta expresión es el conjunto vacío, independientemente del contenido de la relación *cuenta*.

Por tanto, la estrategia final es que el sitio Guadarrama devuelva $cuenta_1$ como resultado de la consulta.

22.7.2 Procesamiento de reuniones sencillas

Como se vio en el Capítulo 14, una decisión importante en la selección de una estrategia de procesamiento de consultas es la elección de la estrategia de reunión. Considérese la siguiente expresión del álgebra relacional:

$$cuenta \bowtie impositor \bowtie sucursal$$

Supóngase que ninguna de las tres relaciones está replicada ni fragmentada, y que *cuenta* está almacenada en el sitio S_1, *impositor* en S_2 y *sucursal* en S_3. Supóngase que S_F denota el sitio en el que se ha formulado la consulta. El sistema necesita obtener el resultado en el sitio S_F. Entre las posibles estrategias para el procesamiento de esta consulta figuran las siguientes:

- Enviar copias de las tres relaciones al sitio S_F. Empleando las técnicas del Capítulo 14, hay que escoger una estrategia para el procesamiento local de toda la consulta en el sitio S_F.
- Enviar una copia de la relación *cuenta* al sitio S_2 y calcular $temp_1$ = *cuenta* $\bowtie$ *impositor* en S_2. Enviar $temp_1$ de S_2 a S_3 y calcular $temp_2$ = $temp_1$ $\bowtie$ *sucursal* en S_3. Enviar el resultado $temp_2$ a S_F.
- Diseñar estrategias parecidas a la anterior con los roles de S_1, S_2 y S_3 intercambiados.

Ninguna de las estrategias es la mejor en todos los casos. Entre los factores que deben tenerse en cuenta están el volumen de los datos que se envían, el coste de la transmisión de los bloques de datos entre cada par de sitios y la velocidad relativa de procesamiento en cada sitio. Considérense las dos primeras estrategias mencionadas. Supóngase que los índices existentes en S_2 y en S_3 resultan útiles para calcular la reunión. Si se envían las tres relaciones a S_F, habrá que volver a crear esos índices en S_F o emplear una estrategia de reunión diferente, posiblemente más costosa. Esta recreación de los índices

supone una sobrecarga adicional de procesamiento y más accesos a disco. Con la segunda estrategia hay que enviar una relación potencialmente grande (*cuenta* $\bowtie$ *impositor*) de S_2 a S_3. Esta relación repite el nombre de cada cliente una vez por cada cuenta que tenga abierta. Por tanto, puede que la segunda estrategia, en comparación con la primera, dé lugar a un mayor volumen de transmisión de datos por la red.

22.7.3 Estrategia de semirreunión

Supóngase que se desea evaluar la expresión $r_1 \bowtie r_2$, donde r_1 y r_2 se almacenan en los sitios S_1 y S_2, respectivamente. Sean r_1 y r_2 los esquemas de r_1 y de r_2. Supóngase que se desea obtener el resultado en S_1. Si hay muchas tuplas de r_2 que no se reúnen con ninguna tupla de r_1, el envío de r_2 a S_1 supone el envío de tuplas que no contribuyen al resultado. Se desea eliminar esas tuplas antes de enviar los datos a S_1, especialmente si los costes de la red son elevados.

Una posible estrategia para lograr todo esto es la siguiente:

1. Calcular $temp_1 \leftarrow \Pi_{R_1 \cap R_2}(r_1)$ en S_1.
2. Enviar $temp_1$ de S_1 a S_2.
3. Calcular $temp_2 \leftarrow r_2 \bowtie temp_1$ en S_2.
4. Enviar $temp_2$ de S_2 aS_1.
5. Calcular $r_1 \bowtie temp_2$ en S_1. La relación resultante es la misma que $r_1 \bowtie r_2$.

Antes de considerar la eficiencia de esta estrategia hay que comprobar que la estrategia calcula la respuesta correcta. En el paso 3 $temp_2$ tiene el resultado de $r_2 \bowtie \Pi_{R_1 \cap R_2}(r_1)$. En el paso 5 se calcula

$$r_1 \bowtie r_2 \bowtie \Pi_{R_1 \cap R_2}(r_1)$$

Dado que la reunión es asociativa y conmutativa, se puede volver a escribir esta expresión como

$$(r_1 \bowtie \Pi_{R_1 \cap R_2}(r_1)) \bowtie r_2$$

Dado que $r_1 \bowtie \Pi_{(R_1 \cap R_2)}(r_1) = r_1$, la expresión es, realmente, igual a $r_1 \bowtie r_2$, la expresión que se pretendía evaluar.

Esta estrategia resulta especialmente ventajosa cuando contribuyen a la reunión relativamente pocas tuplas de r_2. Es probable que se produzca esta situación si r_1 es resultado de una expresión de álgebra relacional que implica una selección. En esos casos, puede que $temp_2$ tenga significativamente menos tuplas que r_2. El ahorro de costes de la estrategia procede de no tener que enviar a S_1 más que $temp_2$, en vez de toda r_2. El envío de $temp_1$ a S_2 supone un coste adicional. Si sólo contribuye a la reunión una fracción de tuplas de r_2 lo bastante pequeña, la sobrecarga del envío de $temp_1$ queda dominada por el ahorro de no tener que enviar más que una parte de las tuplas de r_2.

Esta estrategia se denomina **estrategia de semirreunión** (debido al operador de semirreunión del álgebra relacional, denotado por $\ltimes$). La semirreunión de r_1 con r_2, denotada por $r_1 \ltimes r_2$, es

$$\Pi_{R_1}(r_1 \bowtie r_2)$$

Por tanto, $r_1 \ltimes r_2$ selecciona las tuplas de r_1 que han contribuido a $r_1 \bowtie r_2$. En el paso 3 $temp_2 = r_2 \ltimes r_1$.

Para las reuniones de varias relaciones esta estrategia puede ampliarse a una serie de pasos de semirreunión. Se ha desarrollado un importante corpus teórico en relación con el empleo de la semirreunión para la optimización de consultas. Parte de esta teoría se menciona en las notas bibliográficas.

22.7.4 Estrategias de reunión que aprovechan el paralelismo

Considérese una reunión de cuatro relaciones:

$$r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$$

donde la relación r_i se guarda en el sitio S_i. Supóngase que el resultado debe presentarse en el sitio S_1. Hay muchas estrategias posibles para la evaluación en paralelo (el problema del procesamiento de las consultas en paralelo se estudió con detalle en el Capítulo 21). En una de estas estrategias, r_1 se envía

a S_2, donde se calcula $r_1 \bowtie r_2$. Al mismo tiempo, r_3 se envía a S_4, donde se calcula $r_3 \bowtie r_4$. El sitio S_2 puede enviar las tuplas de $(r_1 \bowtie r_2)$ a S_1 a medida que se generan, en lugar de esperar a que se calcule toda la reunión. De manera parecida, S_4 puede enviar las tuplas de $(r_3 \bowtie r_4)$ a S_1. Una vez que las tuplas de $(r_1 \bowtie r_2)$ y de $(r_3 \bowtie r_4)$ hayan llegado a S_1, puede comenzar el cálculo de $(r_1 \bowtie r_2) \bowtie (r_3 \bowtie r_4)$, con la técnica de reunión canalizada del Apartado 13.7.2.2. Por tanto, el cálculo del resultado de la reunión final en S_1 puede hacerse en paralelo con el cálculo de $(r_1 \bowtie r_2)$ en S_2, y con el de $(r_3 \bowtie r_4)$ en S_4.

22.8 Bases de datos distribuidas heterogéneas

Muchas de las últimas aplicaciones de bases de datos necesitan datos de gran variedad de bases de datos ya existentes y ubicadas en un conjunto heterogéneo de entornos de hardware y de software. El tratamiento de la información ubicada en bases de datos distribuidas heterogéneas exige una capa de software adicional por encima de los sistemas de bases de datos ya existentes. Esta capa de software se denomina **sistema de bases de datos múltiples**. Puede que los sistemas locales de bases de datos empleen modelos lógicos y lenguajes de definición y de tratamiento de datos diferentes, y que difieran en sus mecanismos de control de concurrencia y de administración de las transacciones. Los sistemas de bases de datos múltiples crean la ilusión de la integración lógica de las bases de datos sin necesidad de su integración física.

La integración completa de sistemas heterogéneos en una misma base de datos distribuida homogénea suele resultar difícil o imposible:

- **Dificultades técnicas.** La inversión en los programas de aplicaciones basados en los sistemas de bases de datos ya existentes puede ser enorme, y el coste de transformar esas aplicaciones puede resultar prohibitivo.
- **Dificultades organizativas.** Aunque la integración resulte *técnicamente* posible, puede que no lo sea *políticamente*, porque los sistemas de bases de datos ya existentes pertenezcan a diferentes empresas u organizaciones. En ese caso es importante que el sistema de bases de datos múltiples permita que los sistemas de bases de datos locales conserven un elevado grado de **autonomía** para la base de datos local y para las transacciones que se ejecuten con esos datos.

Por estos motivos los sistemas de bases de datos múltiples ofrecen ventajas significativas que compensan la sobrecarga que suponen. En este apartado se proporciona una visión general de los retos que se afrontan al construir entornos con bases de datos múltiples desde el punto de vista de la definición de los datos y del procesamiento de las consultas. El Apartado 25.7 ofrece una visión general de los problemas de la administración de las transacciones en bases de datos múltiples.

22.8.1 Vista unificada de los datos

Cada sistema local de administración de bases de datos puede utilizar un modelo de datos diferente. Por ejemplo, puede que algunos empleen el modelo relacional, mientras que otros pueden emplear modelos de datos más antiguos, como el de red (véase el Apéndice A) o el jerárquico (véase el Apéndice B).

Dado que se supone que los sistemas con bases de datos múltiples ofrecen la ilusión de un solo sistema de bases de datos integrado, hay que utilizar un modelo de datos común. Una opción adoptada con frecuencia es el modelo relacional, con SQL como lenguaje común de consulta. En realidad, hoy en día hay varios sistemas disponibles que permiten realizar consultas SQL en sistemas de administración de bases de datos no relacionales.

Otra dificultad es proporcionar un esquema conceptual común. Cada sistema local ofrece su propio esquema conceptual. El sistema de bases de datos múltiples debe integrar esos esquemas independientes en uno común. La integración de los esquemas es una tarea complicada, sobre todo por la heterogeneidad semántica.

La integración de los esquemas no es la mera traducción directa de unos lenguajes de definición de datos a otros. Puede que los mismos nombres de atributos aparezcan en diferentes bases de datos locales pero con significados distintos. Puede que los tipos de datos utilizados en un sistema no estén soportados por los demás y que la traducción de unos tipos a otros no resulte sencilla. Incluso en el

caso de tipos de datos idénticos pueden surgir problemas debidos a la representación física de los datos: puede que un sistema utilice ASCII y otro EBCDIC; las representaciones en coma flotante pueden ser diferentes, los enteros pueden representarse como *orden más significativo* (*big–endian*) o como *orden menos significativo* (*little–endian*). En el nivel semántico, el valor entero de una longitud puede ser pulgadas en un sistema y milímetros en otro, lo que crea una situación incómoda en la que la igualdad entre los enteros sea sólo un concepto aproximado (como ocurre siempre con los números con coma flotante). Puede que el mismo nombre aparezca en idiomas distintos en diferentes sistemas. Por ejemplo, puede que un sistema basado en Estados Unidos se refiera a la ciudad de Saragossa, mientras que uno con base en España lo haga como Zaragoza.

Todas estas diferencias aparentemente menores deben registrarse de manera adecuada en el esquema conceptual global común. Hay que proporcionar funciones de traducción. Hay que anotar los índices para el comportamiento dependiente del sistema (por ejemplo, el orden de clasificación de los caracteres no alfanuméricos no es igual en ASCII que en EBCDIC). Como ya se ha comentado, puede que la alternativa de convertir cada base de datos a un formato común no resulte factible sin dejar obsoletos los programas de aplicación ya existentes.

22.8.2 Procesamiento de las consultas

El procesamiento de las consultas en las bases de datos heterogéneas puede resultar complicado. Algunos de los problemas son:

- Dada una consulta en un esquema global, puede que haya que traducir la consulta a consultas en los esquemas locales de cada uno de los sitios en que hay que ejecutar la consulta. Hay que volver a traducir los resultados de las consultas al esquema global.

  La tarea se simplifica escribiendo **envolturas** para cada origen de datos, que ofrezcan una vista de los datos locales en el esquema global. Las envolturas también traducen las consultas del esquema global a consultas del esquema local y vuelven a traducir los resultados al esquema global. Las envolturas puede ofrecerlas cada sitio o escribirse de manera independiente como parte del sistema de bases de datos múltiples.

  Las envolturas pueden, incluso, utilizarse para proporcionar una vista relacional de orígenes de datos no relacionales, como las páginas Web (posiblemente con interfaces de formularios), archivos planos, bases de datos jerárquicas y de red y sistemas de directorio.

- Puede que algunos orígenes de datos sólo ofrezcan capacidades de consulta limitadas; por ejemplo, puede que soporten selecciones pero no reuniones. Puede incluso que restrinjan la forma de las selecciones, permitiéndolas sólo para determinados campos; los orígenes de datos Web con interfaces de formulario son un ejemplo de este tipo de orígenes de datos. Por tanto, puede que haya que dividir las consultas para que se lleven a cabo en parte en el origen de datos y en parte en el sitio que formula la consulta.

- En general, puede que haya que acceder a más de un sitio para responder a una consulta dada. Es posible que haya que procesar las respuestas obtenidas de los diferentes sitios para eliminar los valores duplicados. Supóngase que un sitio contiene las tuplas de *cuenta* que satisfacen la selección $saldo < 100$, mientras que otro contiene las que satisfacen $saldo > 50$. Una consulta sobre toda la relación cuenta exigiría acceder a los dos sitios y eliminar las respuestas duplicadas consecuencia de las tuplas con saldo entre 50 y 100, que están replicadas en los dos sitios.

- La optimización global de consultas en bases de datos heterogéneas resulta difícil, ya que puede que el sistema de ejecución de consultas no conozca los costes de los planes de consulta alternativos en los diferentes sitios. La solución habitual es confiar sólo en la optimización a nivel local y utilizar únicamente la heurística a nivel global.

Los sistemas **mediadores** son sistemas que integran varios orígenes de datos heterogéneos, lo que proporciona una vista global integrada de los datos y facilidades de consulta sobre la misma. A diferencia de los sistemas de bases de datos múltiples completos, los sistemas mediadores no se ocupan del procesamiento de las transacciones (los términos mediador y bases de datos múltiples suelen utilizarse

de manera indistinta, y puede que los sistemas denominados mediadores soporten formas limitadas de transacción). El término **base de datos virtual** se utiliza para hacer referencia a los sistemas de bases de datos múltiples o a los sistemas mediadores, ya que ofrecen la apariencia de una sola base de datos con un esquema global, aunque los datos estén en varios sitios en esquemas locales.

22.9 Sistemas de directorio

Considérese una organización que desea poner los datos de sus empleados a disposición de diferentes miembros de la organización; entre esos datos estarían el nombre, el cargo, el ID de empleado, la dirección, la dirección de correo electrónico, el número de teléfono, el número de fax, etc. En los días anteriores a la informática las organizaciones creaban directorios físicos de los empleados y los distribuían por toda la organización. Incluso en nuestros días, las compañías telefónicas crean directorios físicos de sus clientes.

En general, un directorio es un listado de la información sobre alguna clase de objetos como las personas. Los directorios pueden utilizarse para hallar información sobre un objeto concreto o, en sentido contrario, hallar objetos que cumplen un determinado requisito. En el mundo de los directorios telefónicos físicos, los directorios que permiten las búsquedas en sentido directo se denominan **páginas blancas**, mientras que los directorios que permiten las búsquedas en sentido inverso se denominan **páginas amarillas**.

En el mundo interconectado de hoy en día la necesidad de los directorios sigue vigente y, si acaso, es aún más importante. No obstante, hoy en día los directorios deben estar disponibles en las redes informáticas en lugar de en forma física (papel).

22.9.1 Protocolos de acceso a directorios

La información de directorio puede ponerse a disposición de los usuarios mediante interfaces Web, como hacen muchas organizaciones y, en especial, las compañías telefónicas. Esas interfaces son buenas para las personas que las utilizan. No obstante, también los programas necesitan tener acceso a la información de los directorios. Éstos pueden utilizarse para almacenar otros tipos de información, de manera parecida a como hacen los directorios de sistemas de archivos. Por ejemplo, los exploradores Web pueden almacenar marcas personales de sitios favoritos y otros parámetros del explorador en un sistema de directorio. Por tanto, los usuarios pueden acceder a los mismos parámetros desde varias ubicaciones, por ejemplo, desde casa y desde el trabajo, sin tener que compartir el sistema de archivos.

Se han desarrollado varios **protocolos!de acceso a directorios** para ofrecer una manera normalizada de acceso a los datos contenidos en ellos. Entre todos, el más utilizado hoy en día es el **protocolo ligero de acceso a directorios** (Lightweight Directory Access Protocol, LDAP).

Evidentemente, todos los tipos de datos de los ejemplos de este capítulo pueden almacenarse sin demasiados problemas en sistemas de bases de datos, y se puede acceder a ellos mediante protocolos como JDBC u ODBC. La pregunta, entonces, es si hace falta crear un protocolo especializado para el acceso a la información de directorio. Al menos hay dos respuestas a esa pregunta.

- En primer lugar, los protocolos de acceso a directorios son protocolos simplificados que atienden a un tipo limitado de acceso a los datos. Han evolucionado en paralelo con los protocolos de acceso a las bases de datos.
- En segundo lugar, y lo que es más importante, los sistemas de directorio ofrecen un mecanismo sencillo para nombrar a los objetos de manera jerárquica, parecida a los nombres de los directorios de los sistemas de archivos, que pueden utilizarse en un sistema distribuido de directorio para especificar la información que se almacena en cada servidor de directorio. Por ejemplo, puede que un servidor de directorio concreto almacene la información de los empleados de Laboratorios Bell en Cáceres y que otro almacene la de los empleados de Zarzalejo, lo que da a ambos sitios autonomía para controlar sus datos locales. Se puede utilizar el protocolo de acceso al directorio para obtener datos de los dos directorios a través de la red. Lo que es más importante, el sistema de directorios puede configurarse para que envíe de manera automática a un sitio las consultas formuladas en el otro, sin intervención del usuario.

Por estos motivos, varias organizaciones tienen sistemas de directorios para hacer que la información de la organización esté disponible en conexión mediante un protocolo de acceso al directorio. La información del directorio de la organización se puede utilizar con varios fines, como encontrar la dirección, el teléfono o las direcciones de correo electrónico de la gente, encontrar el departamento al que pertenece cada persona y explorar la jerarquía de los departamentos. Los directorios también se utilizan para autenticar a los usuarios: las aplicaciones pueden recoger la información de autentificación, como las contraseñas de los usuarios, y autenticarlos mediante el directorio.

Como cabe esperar, varias implementaciones de los directorios consideran conveniente utilizar las bases de datos relacionales para almacenar los datos, en lugar de crear sistemas de almacenamiento específicos.

22.9.2 El protocolo de acceso ligero a directorios LDAP (Lightweight Directory Access Protocol)

En general, los sistemas de directorios se implementan como uno o varios servidores que atienden a varios clientes. Los clientes utilizan la interfaz de programación de aplicaciones definida por el sistema de directorios para comunicarse con los servidores de directorios. Los protocolos de acceso a los directorios también definen un modelo de datos y el control de los accesos.

El **protocolo de acceso a directorios X.500**, definido por la organización internacional para la normalización (International Organization for Standardization, ISO), es una norma para el acceso a información de los directorios. No obstante, el protocolo es bastante complejo y no se utiliza mucho. El **protocolo ligero de acceso a directorios** (Lightweight Directory Access Protocol, LDAP) ofrece muchas de las características de X.500, pero con menos complejidad y se utiliza mucho. En el resto de este apartado se esbozarán detalles del modelo de datos y del protocolo de acceso de LDAP.

22.9.2.1 El modelo de datos LDAP

En LDAP los directorios almacenan **entradas**, que son parecidas a los objetos. Cada entrada debe tener un **nombre distinguido** (ND), que la identifica de manera única. Cada ND, a su vez, está formado por una secuencia de **nombres distinguidos relativos** (NDR). Por ejemplo, una entrada puede tener el siguiente nombre distinguido:

```
cn=Silberschatz, ou=Laboratorios Bell, o=Lucent, c=EEUU
```

Como puede verse, el nombre distinguido de este ejemplo es una combinación de nombre y dirección (dentro de la organización), que comienza por el nombre de la persona y luego da la unidad organizativa (*organizational unit*, ou), la organización (*organization*, o) y el país (*country*, c). El orden de los componentes del nombre distinguido refleja el orden normal de las direcciones postales, en lugar del orden inverso que se utiliza al especificar nombres de camino para los archivos. El conjunto de NDRs de cada ND viene definido por el esquema del sistema de directorio.

Las entradas también pueden tener atributos. LDAP ofrece los tipos binary (binario), string (cadena de caracteres) y time (hora) y, de manera adicional, los tipos tel (telefónico) para los números de teléfono y PostalAddress (dirección postal) para las direcciones (las líneas se separan con un carácter "$"). A diferencia de los del modelo relacional, de manera predeterminada los atributos pueden tener varios valores, por lo que es posible almacenar varios números de teléfono o direcciones para cada entrada.

LDAP permite la definición de **clases de objetos** con nombres y tipos de atributos. Se puede utilizar la herencia para definir las clases de objetos. Además, se puede especificar que las entradas sean de una clase de objeto o de varias. No es necesario que haya una única clase de objeto más específica a la que pertenezca una entrada dada.

Las entradas se organizan en un **árbol de información del directorio (AID)**, de acuerdo con sus nombres distinguidos. Las entradas en el nivel de las hojas del árbol suelen representar objetos concretos. Las entradas que son nodos internos representan objetos como las unidades organizativas, las organizaciones o los países. Los hijos de cada nodo tienen un ND que contiene todos los NDR del padre, y uno o

varios NDR adicionales. Por ejemplo, puede que un nodo interno tenga como c=España del ND, y todas las entradas por debajo de él tengan el valor EEUU para el campo c del NDR.

No hace falta almacenar el nombre distinguido completo en las entradas. El sistema puede generar el nombre distinguido de cada una de ellas recorriendo el AID en sentido ascendente desde la entrada, reuniendo los componentes NDR=valor para crear el nombre distinguido completo.

Puede que las entradas tengan más de un nombre distinguido (por ejemplo, la entrada de una persona en más de una organización). Para tratar estos casos el nivel de las hojas del AID puede ser un **alias**, que apunte a una entrada en otra rama del árbol.

22.9.2.2 Tratamiento de los datos

A diferencia de SQL, LDAP no define ni un lenguaje de definición de datos ni uno de tratamiento de datos. Sin embargo, LDAP define un protocolo de red para llevar a cabo la definición y el tratamiento de los datos. Los usuarios de LDAP pueden utilizar tanto una interfaz de programación de aplicaciones como las herramientas ofrecidas por diferentes fabricantes para llevar a cabo la definición y el tratamiento de los datos. LDAP también define un formato de archivos denominado **formato de intercambio de datos LDAP** (LDAP Data Interchange Format, LDIF) que puede utilizarse para almacenar e intercambiar información.

El mecanismo de consulta en LDAP es muy sencillo, consiste simplemente en selecciones y proyecciones, sin ninguna reunión. Cada consulta debe especificar lo siguiente:

- Una base (es decir, un nodo del AID), dando su nombre distinguido (el camino desde la raíz hasta el nodo).
- Una condición de búsqueda, que puede ser una combinación booleana de condiciones para diferentes atributos. Se soportan la igualdad, la coincidencia con caracteres comodín y la igualdad aproximada (la definición exacta de la igualdad aproximada depende de cada sistema).
- Un ámbito, que puede ser sencillamente la base, la base y sus hijos o todo el subárbol por debajo de la base.
- Los atributos que hay que devolver.
- Los límites impuestos al número de resultados y al consumo de recursos.

La consulta también puede especificar si hay que eliminar de manera automática las referencias de los alias; si se desactiva la eliminación de las referencias de los alias se pueden devolver las entradas de los alias como respuestas.

Una manera de consultar orígenes de datos LDAP es emplear URL LDAP. Ejemplos de URL LDAP son:

```
ldap:://aura.research.bell-labs.com/o=Lucent,c=EEUU
ldap:://aura.research.bell-labs.com/o=Lucent,c=EEUU??sub?cn=Korth
```

El primer URL devuelve todos los atributos de todas las entradas del servidor en que la organización es Lucent y el país es EEUU. El segundo URL ejecuta una consulta de búsqueda (selección) cn=Korth en el subárbol del nodo con nombre distinguido o=Lucent, c=EEUU. Los signos de interrogación del URL separan campos diferentes. El primer campo es el nombre distinguido, en este caso o=Lucent,c=EEUU. El segundo campo, la lista de atributos que hay que devolver, se ha dejado vacío, lo que significa que hay que devolver todos los atributos. El tercer atributo, sub, indica que hay que buscar en todo el subárbol. El último parámetro es la condición de búsqueda.

Una segunda manera de consultar los directorios LDAP es utilizar una interfaz de programación de aplicaciones. La Figura 22.6 muestra un fragmento de código C que se utiliza para conectar con un servidor LDAP y ejecutar en él una consulta. El código, en primer lugar, abre una conexión con un servidor LDAP mediante ldap_open y ldap_bind. Luego ejecuta una consulta mediante ldap_search_s. Los argumentos para ldap_search_s son el controlador de conexión LDAP, el ND de la base desde la que se debe realizar la búsqueda, el ámbito de la búsqueda, la condición de búsqueda, la lista de atributos que hay que devolver y un atributo denominado attrsonly que, si se le asigna el valor de 1, hace que sólo

se devuelva el esquema del resultado, sin ninguna tupla real. El último argumento es un argumento de resultados que devuelve el resultado de la búsqueda en forma de estructura LDAPMessage.

El primer bucle **for** itera sobre cada entrada del resultado y la imprime. Obsérvese que cada entrada puede tener varios atributos, y el segundo bucle **for** imprime cada uno de ellos. Dado que los atributos en LDAP pueden tener varios valores, el tercer bucle **for** imprime cada uno de los valores de cada atributo. Las llamadas ldap_msgfree y ldap_value_free liberan la memoria que asignan las bibliotecas LDAP. La Figura 22.6 no muestra el código para el tratamiento de las condiciones de error.

La API de LDAP también contiene funciones para crear, actualizar y eliminar entradas, así como para otras operaciones con el AID. Cada llamada a una función se comporta como una transacción independiente; LDAP no soporta la atomicidad de las actualizaciones múltiples.

22.9.2.3 Árboles de directorio distribuidos

La información sobre las organizaciones se puede dividir entre varios AIDs, cada uno de los cuales almacena información sobre algunas entradas. El **sufijo** de cada AID es una secuencia de pares RDN=valor (RDN, Relative Distinguished Name, nombre relativo distinguido) que identifica la información que almacena ese AID; los pares están concatenados con el resto del nombre distinguido generado al recorrer el árbol desde la entrada hasta la raíz. Por ejemplo, el sufijo de un AID puede ser o=Lucent, c=EEUU,

```
#include <stdio.h>
#include <ldap.h>
main() {
      LDAP *ld;
      LDAPMessage *res, *entry;
      char *dn, *attr, *attrList[] = {"telephoneNumber", NULL};
      BerElement *ptr;
      int vals, i;
      ld = ldap_open("aura.research.bell-labs.com", LDAP_PORT);
      ldap_simple_bind(ld, "avi", "avi-passwd") ;
      ldap_search_s(ld, "o=Lucent, c=EEUU", LDAP_SCOPE_SUBTREE, "cn=Korth",
                  attrList, /*attrsonly*/ 0, &res);
      printf("found %d entries", ldap_count_entries(ld, res));
      for (entry=ldap_first_entry(ld, res); entry != NULL;
                        entry = ldap_next_entry(ld, entry))
      {
            dn = ldap_get_dn(ld, entry);
            printf("dn: %s", dn);
            ldap_memfree(dn);
            for (attr = ldap_first_attribute(ld, entry, &ptr);
                        attr ! NULL;
                        attr = ldap_next_attribute(ld, entry, ptr))
            {
                  printf(" %s: ", attr);
                  vals = ldap_get_values(ld, entry, attr);
                  for (i=0; vals[i] != NULL; i++)
                        printf(" %s, ", vals[i]);
                  ldap_value_free(vals);
            }
      }
      ldap_msgfree(res);
      ldap_unbind(ld);
}
```

Figura 22.6 Ejemplo de código LDAP en C.

mientras que otro puede tener el sufijo o=Lucent, c=India. Los AIDs pueden estar divididos organizativa y geográficamente.

Los nodos de un AID pueden contener una **referencia** a un nodo de otro AID; por ejemplo, la unidad organizativa Laboratorios Bellbajo o=Lucent, c=EEUU puede tener su propio AID, en cuyo caso el AID de o=Lucent, c=EEUU tendría el nodo ou=Laboratorios Bell, que representará una referencia al AID de Laboratorios Bell.

Las referencias son el componente clave que ayuda a organizar un conjunto distribuido de directorios en un sistema integrado. Puede que, cuando un servidor recibe una consulta sobre un AID, devuelva una referencia al cliente, el cual, a su vez, emite una consulta sobre el AID referenciado. El acceso al AID referenciado es transparente, y se lleva a cabo sin el conocimiento del usuario. Alternativamente, el propio servidor puede formular la consulta al AID referenciado y devolver el resultado junto con el resultado calculado localmente.

El mecanismo jerárquico de denominación utilizado por LDAP ayuda a repartir el control de la información entre diferentes partes de la organización. La facilidad de las referencias ayuda a integrar todos los directorios de la organización en un solo directorio virtual.

Aunque no sea un requisito LDAP, las organizaciones suelen decidir repartir la información por criterios geográficos (por ejemplo, puede que la organización mantenga un directorio por cada sitio en que tenga una presencia importante) o según su estructura organizativa (por ejemplo, cada unidad organizativa, como puede ser un departamento, mantiene su propio directorio).

Muchas implementaciones de LDAP soportan la réplica maestro−esclavo y la réplica multimaestro de los AIDs, aunque la réplica no forme parte de la versión actual de la norma LDAP, la 3. El trabajo de normalización de la réplica en LDAP se halla en curso.

22.10 Resumen

- Los sistemas distribuidos de bases de datos consisten en un conjunto de sitios, cada uno de los cuales mantiene un sistema local de bases de datos. Cada sitio puede procesar las transacciones locales: las transacciones que sólo acceden a datos de ese mismo sitio. Además, cada sitio puede participar en la ejecución de transacciones globales: las que acceden a los datos de varios sitios. La ejecución de las transacciones globales necesita que haya comunicación entre los diferentes sitios.

- Las bases de datos distribuidas pueden ser homogéneas, en las que todos los sitios tienen un esquema y un código de sistemas de bases de datos comunes, o heterogéneas, en las que el esquema y el código de los sistemas pueden ser diferentes.

- Surgen varios problemas relacionados con el almacenamiento de relaciones en bases de datos distribuidas, incluidas la réplica y la fragmentación. Es fundamental que el sistema minimice el grado de conocimiento por los usuarios del modo en que se almacenan las relaciones.

- Los sistemas distribuidos pueden sufrir los mismos tipos de fallos que los sistemas centralizados. No obstante, hay otros fallos con los que hay que tratar en los entornos distribuidos; entre ellos, los fallos de los sitios, los de los enlaces, las pérdidas de mensajes y las divisiones de la red. Es necesario tener en consideración cada uno de esos problemas en el diseño del esquema distribuido de recuperación.

- Para asegurar la atomicidad, todos los sitios en los que se ejecuta la transacción T deben estar de acuerdo en el resultado final de su ejecución. O bien T se compromete en todos los sitios o se aborta en todos. Para asegurar esta propiedad el coordinador de la transacción de T debe ejecutar un protocolo de compromiso. El protocolo de compromiso más empleado es el protocolo de compromiso de dos fases.

- El protocolo de compromiso de dos fases puede provocar bloqueos, la situación en que el destino de una transacción no se puede determinar hasta que se recupere un sitio que haya fallado (el coordinador). Se puede utilizar el protocolo de compromiso de tres fases para reducir la probabilidad de bloqueo.

- La mensajería persistente ofrece un modelo alternativo para el tratamiento de las transacciones distribuidas. El modelo divide cada transacción en varias partes que se ejecutan en diferentes bases de datos. Se envían mensajes persistentes (que está garantizado que se entregan exactamente una vez, independientemente de los fallos) a los sitios remotos para solicitar que se emprendan acciones en ellos. Aunque la mensajería persistente evita el problema de los bloqueos, los desarrolladores de aplicaciones tienen que escribir código para tratar diferentes tipos de fallos.
- Los diferentes esquemas de control de concurrencia empleados en los sistemas centralizados se pueden modificar para su empleo en entornos distribuidos.
  - En el caso de los protocolos de bloqueo, la única modificación que hay que hacer es el modo en que se implementa el gestor de bloqueos. Existen varios enfoques posibles. Se pueden utilizar uno o varios coordinadores centrales. Si, en vez de eso, se adopta un enfoque con un gestor distribuido de bloqueos, hay que tratar de manera especial los datos replicados.
  - Entre los protocolos para el tratamiento de los datos replicados se hallan el protocolo de copia principal, el de mayoría, el sesgado y el de consenso de quórum. Cada uno de ellos representa diferentes equilibrios en términos de coste y de posibilidad de trabajo en presencia de fallos.
  - En el caso de los esquemas de marcas temporales y de validación, el único cambio necesario es el desarrollo de un mecanismo para la generación de marcas temporales globales únicas.
  - Muchos sistemas de bases de datos soportan la réplica perezosa, en la que las actualizaciones se propagan a las réplicas ubicadas fuera del ámbito de la transacción que ha llevado a cabo la actualización. Estas facilidades deben utilizarse con grandes precauciones, ya que pueden dar lugar a ejecuciones no secuenciables.
- La detección de interbloqueos en entornos con gestor distribuido de bloqueos exige la colaboración entre varios sitios, dado que puede haber interbloqueos globales aunque no haya ningún interbloqueo local.
- Para ofrecer una elevada disponibilidad, las bases de datos distribuidas deben detectar los fallos, reconfigurarse de modo que el cálculo pueda continuar y recuperarse cuando se repare el procesador o el enlace correspondiente. La tarea se complica enormemente por el hecho de que resulta difícil distinguir entre las divisiones de la red y los fallos de los sitios.

  Se puede extender el protocolo de mayoría mediante el empleo de números de versión para permitir que continúe el procesamiento de las transacciones incluso en presencia de fallos. Aunque el protocolo supone una sobrecarga significativa, funciona independientemente del tipo de fallo. Se dispone de protocolos menos costosos para tratar los fallos de los sitios, pero dan por supuesto que no se producen divisiones de la red.
- Algunos algoritmos distribuidos exigen el empleo de coordinadores. Para ofrecer una elevada disponibilidad el sistema debe mantener una copia de seguridad que esté preparada para asumir la responsabilidad si falla el coordinador. Otro enfoque es escoger el nuevo coordinador después de que haya fallado el anterior. Los algoritmos que determinan el sitio que deberá actuar como coordinador se denominan **algoritmos de elección**.
- Puede que las consultas a las bases de datos distribuidas necesiten tener acceso a varios sitios. Se dispone de varias técnicas de optimización para escoger los sitios a los que hay que tener acceso. Basadas en la fragmentación y en la réplica, las técnicas pueden utilizar técnicas de semirreunión para reducir las transferencias de datos.
- Las bases de datos distribuidas heterogéneas permiten que cada sitio tenga su propio esquema y su propio código de sistema de bases de datos. Los sistemas de bases de datos múltiples ofrecen un entorno en el que las nuevas aplicaciones de bases de datos pueden acceder a los datos de gran variedad de bases de datos ya existentes ubicadas en diferentes entornos heterogéneos de hardware y de software. Puede que los sistemas locales de bases de datos empleen modelos lógicos y lenguajes de definición o de manipulación de datos diferentes y que se diferencien en los mecanismos de control de concurrencia o de administración de las transacciones. Los sistemas de bases de datos múltiples crean la ilusión de la integración lógica de las bases de datos, sin exigir su integración física.

- Los sistemas de directorio pueden considerarse una modalidad especializada de base de datos en la que la información se organiza de manera jerárquica, parecida al modo en que los archivos se organizan en los sistemas de archivos. Se accede a los directorios mediante protocolos normalizados de acceso a directorios, como LDAP.

  Los directorios pueden estar distribuidos entre varios sitios para proporcionar autonomía a cada sitio. Los directorios pueden contener referencias a otros directorios, lo que ayuda a crear vistas integradas en las que cada consulta sólo se envía a un directorio y se ejecuta de manera transparente en los directorios correspondientes

Términos de repaso

- Base de datos distribuida homogénea.
- Base de datos distribuida heterogénea.
- Réplica de datos.
- Copia principal.
- Fragmentación de los datos:
  - Horizontal.
  - Vertical.
- Transparencia de los datos:
  - De la fragmentación.
  - De la réplica.
  - De la ubicación.
- Servidor de nombres.
- Alias.
- Transacciones distribuidas:
  - Locales.
  - Globales.
- Gestor de transacciones.
- Coordinador de transacciones.
- Modalidades de fallo del sistema.
- División de la red.
- Protocolos de compromiso.
- Protocolo de compromiso de dos fases (C2F).
  - Estado de preparación.
  - Transacciones dudosas.
  - Problema del bloqueo.
- Protocolo de compromiso de tres fases (C3F).
- Mensajería persistente.
- Control de la concurrencia.
- Gestor único de bloqueos.
- Gestor distribuido de bloqueos.
- Protocolos para las réplicas.
  - Copia principal.
  - Protocolo de mayoría.
  - Protocolo sesgado.
  - Protocolo de consenso de quórum.
- Marcas temporales.
- Réplica maestro–esclavo.
- Réplica con varios maestros (actualización distribuida).
- Instantánea consistente con las transacciones.
- Propagación perezosa.
- Tratamiento de los interbloqueos.
  - Grafo local de espera.
  - Grafo global de espera.
  - Ciclos falsos.
- Disponibilidad.
- Robustez.
  - Enfoque basado en la mayoría.
  - Leer uno, escribir todo.
  - Leer uno, escribir todos los disponibles.
  - Reintegración de sitios.
- Selección del coordinador.
- Coordinador suplente.
- Algoritmos de selección.
- Algoritmo de acoso.
- Procesamiento distribuido de consultas.
- Estrategia de semirreunión.
- Sistema de bases de datos múltiples.
- Autonomía.
- Mediadores.
- Base de datos virtual.
- Sistemas de directorio.
- Protocolo ligero de acceso a directorios LDAP (Lightweight directory access protocol).
  - Nombre distinguido (ND).
  - Nombres distinguidos relativos (NDR).
  - Árbol de información del directorio (AID).
- Árboles distribuidos de directorio.
- Sufijo AID.
- Referencia.

Ejercicios prácticos

22.1 Indíquese lo que diferencia a una base de datos distribuida diseñada para una red de área local de otra diseñada para una red de área amplia.

22.2 Para crear un sistema distribuido con elevada disponibilidad hay que conocer los tipos de fallos que pueden producirse.

a. Indíquense los tipos de fallos posibles en los sistemas distribuidos.
b. Indíquense los elementos de la lista de la pregunta a que también sean aplicables a los sistemas centralizados.

22.3 Considérese un fallo que se produce durante la ejecución de C2F para una transacción. Para cada fallo posible de los indicados en el Ejercicio práctico 22.2.a, explíquese el modo en que C2F garantiza la atomicidad de la transacción a pesar del fallo.

22.4 Considérese un sistema distribuido con dos sitios, *A* y *B*. Indíquese si el sitio *A* puede distinguir entre:

- *B* deja de funcionar.
- El enlace entre *A* y *B* deja de funcionar.
- *B* está extremadamente sobrecargado y su tiempo de respuesta es cien veces el habitual.

Indíquense las implicaciones de la respuesta para la recuperación de los sistemas distribuidos.

22.5 El esquema de mensajería persistente descrito en este capítulo depende de las marcas temporales combinadas con el descarte de los mensajes recibidos si son demasiado antiguos. Propóngase un esquema alternativo basado en los números de secuencia en lugar de en las marcas temporales.

22.6 Dese un ejemplo en que el enfoque de leer uno, escribir todos los disponibles conduzca a un estado erróneo.

22.7 Explíquese la diferencia entre la réplica de datos en los sistemas distribuidos y el mantenimiento de sitios remotos de respaldo.

22.8 Dese un ejemplo en el que la réplica perezosa pueda conducir a un estado inconsistente de la base de datos aunque las actualizaciones obtengan un bloqueo exclusivo sobre la copia principal (maestra).

22.9 Considérese el siguiente algoritmo de detección de interbloqueos. Cuando la transacción T_i, en el sitio S_1, solicita un recurso a T_j, en el sitio S_3, se envía un mensaje de solicitud con la marca temporal *n*. Se inserta el arco (T_i, T_j, n) en el grafo local de espera de S_1. El arco (T_i, T_j, n) sólo se inserta en el grafo local de espera de S_3 si T_j ha recibido el mensaje de solicitud y no puede conceder de manera inmediata el recurso solicitado. La solicitud de T_i a T_j en el mismo sitio se trata de la manera habitual; no se asocia ninguna marca temporal con el arco (T_i, T_j). El coordinador central invoca el algoritmo de detección enviando el mensaje de inicio a cada sitio del sistema.

Al recibir ese mensaje, cada sitio envía al coordinador su grafo local de espera. Obsérvese que ese grafo contiene toda la información local que tiene cada sitio sobre el estado del grafo real. El grafo de espera refleja un estado instantáneo del sitio, pero no está sincronizado con respecto a ningún otro sitio.

Cuando el controlador ha recibido respuesta de cada sitio, crea un grafo de la manera siguiente:

- El grafo contiene un vértice para cada transacción del sistema.
- El grafo tiene un arco (T_i, T_j) si y sólo si
  - ☐ Existe un arco (T_i, T_j) en uno de los grafos de espera.
  - ☐ Aparece un arco (T_i, T_j, n) (para algún *n*) en más de un grafo de espera.

Pruébese que, si hay un ciclo en el grafo creado, el sistema se halla en estado de interbloqueo y que, si no hay ningún ciclo en el grafo creado, el sistema no se hallaba en estado de interbloqueo cuando comenzó la ejecución del algoritmo.

22.10 Considérese una relación que está fragmentada horizontalmente por *número_planta*:

A	B	C
1	2	3
4	5	6
1	2	4
5	3	2
8	9	7

r

C	D	E
3	4	5
3	6	8
2	3	2
1	4	1
1	2	3

s

Figura 22.7 Relaciones para el Ejercicio práctico 22.11

empleado (*nombre*, *dirección*, *sueldo*, *número_planta*)

Supóngase que cada fragmento tiene dos réplicas: una almacenada en el sitio de Madrid y otra almacenada localmente en el sitio de la planta. Descríbase una buena estrategia de procesamiento de las consultas siguientes, formuladas en el sitio de Lima.

a. Determinar todos los empleados de la planta de Managua.
b. Determinar el sueldo promedio de todos los empleados.
c. Determinar el empleado mejor pagado de cada uno de los sitios siguientes: Buenos Aires, Rosario, Córdoba, Bahía Blanca.
d. Determinar el empleado peor pagado de la compañía.

22.11 Calcúlese $r \ltimes s$ para las relaciones de la Figura 22.7.

22.12 Dado que la funcionalidad LDAP puede implementarse sobre un sistema de bases de datos, indíquese el motivo por el que la norma LDAP es necesaria.

Ejercicios

22.13 Explíquense las ventajas relativas de las bases de datos centralizadas y de las distribuidas.

22.14 Explíquense las diferencias entre transparencia de la fragmentación, transparencia de las réplicas y transparencia de la ubicación.

22.15 Indíquese en qué momento resulta útil tener réplicas de los datos o tenerlos fragmentados. Justifíquese la respuesta.

22.16 Explíquense los conceptos de transparencia y de autonomía. Indíquese el motivo de que estos conceptos sean deseables desde el punto de vista de los factores humanos.

22.17 Si se aplica una versión distribuida del protocolo de granularidad múltiple del Capítulo 16 a una base de datos distribuida, el sitio responsable de la raíz del GAD puede convertirse en un cuello de botella. Supóngase que se modifica ese protocolo de la manera siguiente:

- Sólo se permiten en la raíz bloqueos en modo tentativo.
- A todas las transacciones se les conceden de manera automática todos los bloqueos en modo tentativo posibles.

Pruébese que estas modificaciones alivian el problema sin permitir planificaciones no secuenciables.

22.18 Estúdiense y resúmanse las facilidades que ofrece el sistema de bases de datos que se esté utilizando para tratar los estados inconsistentes a los que se puede llegar con la propagación perezosa de las actualizaciones.

22.19 Explíquense las ventajas e inconvenientes de los dos métodos presentados en el Apartado 22.5.2 para la generación de marcas temporales globalmente únicas.

22.20 Considérense las relaciones

empleado (*nombre*, *dirección*, *sueldo*, *número_planta*)
máquina (*número_máquina*, *tipo*, *número_planta*)

Supóngase que la relación *empleado* está fragmentada horizontalmente por *número_planta* y que cada fragmento se almacena localmente en el sitio de la planta correspondiente. Supóngase que toda la relación *máquina* se almacena en el sitio de Sucre. Descríbase una buena estrategia para el procesamiento de cada una de las consultas siguientes.

a. Determinar todos los empleados de la planta que contiene el número de máquina 1130.
b. Determinar todos los empleados de las plantas que contienen máquinas cuyo tipo sea "trituradora".
c. Determinar todas las máquinas de la planta de Almadén.
d. Determinar *empleado* $\bowtie$ *máquina*.

22.21 Para cada una de las estrategias del Ejercicio 22.20 indíquese el modo en que la elección de estrategia depende:

a. Del sitio en el que se formuló la consulta.
b. Del sitio en el que se desea obtener el resultado.

22.22 ¿Es necesariamente $r_i \ltimes r_j$ igual a $r_j \ltimes r_i$? ¿En qué circunstancias se cumple $r_i \ltimes r_j = r_j \ltimes r_i$?

22.23 Descríbase el modo en que se puede utilizar LDAP para ofrecer varias vistas jerárquicas de los datos sin necesidad de replicar los datos del nivel básico.

Notas bibliográficas

Las bases de datos distribuidas se explican en los libros de texto Ozsu y Valduriez [1999] y Ceri y Pelagatti [1984]. Las redes de computadoras se estudian en Tanenbaum [2002] y Halsall [1996]. Breitbart et al. [1999b] presentan una visión general de las bases de datos distribuidas.

La implementación del concepto de transacción en bases de datos distribuidas se presenta en Gray [1981], Traiger et al. [1982], Spector y Schwarz [1983], y Eppinger et al. [1991]. El protocolo C2F lo desarrollaron Lampson y Sturgis [1976] y Gray [1978]. El protocolo de compromiso de tres fases proviene de Skeen [1981]. Mohan y Lindsay [1983] estudian dos versiones modificadas de C2F, denominadas *presumir compromiso* y *presumir abortar*, que reducen la sobrecarga de C2F mediante la definición de suposiciones predeterminadas relativas al destino de las transacciones.

El algoritmo de acoso del Apartado 22.6.5 proviene de Garcia-Molina [1982]. La sincronización distribuida de los relojes se estudia en Lamport [1978]. El control distribuido de la concurrencia se estudia en Menasce et al. [1980], Bernstein y Goodman [1980], Bernstein y Goodman [1981] y Bernstein y Goodman [1982].

El gestor de transacciones de R* se describe en Mohan et al. [1986]. El control de concurrencia de los datos replicados que se basa en el concepto de votación se presenta en Gifford [1979] y Thomas [1979]. Las técnicas de validación para los esquemas distribuidos de control de concurrencia se describen en Schlageter [1981], Ceri y Owicki [1983] y Bassiouni [1988].

El problema de las actualizaciones concurrentes de los datos replicados se revisó en el contexto de los almacenes de datos en Gray et al. [1996] Anderson et al. [1998] estudian problemas relativos a la réplica perezosa y a la consistencia. Breitbart et al. [1999a] describen los protocolos de actualización perezosa para el tratamiento de réplicas.

Los manuales de usuario de varios sistemas de bases de datos ofrecen detalles del modo en que tratan la réplica y la consistencia. Huang y Garcia-Molina [2001] abordan la semántica de sólo-una-vez en los sistemas de mensajería con réplica.

Knapp [1987] estudia la literatura sobre detección distribuida de interbloqueos, el Ejercicio práctico 22.9 procede de Stuart et al. [1984].

El procesamiento distribuido de las consultas se estudia en Wong [1977], Epstein et al. [1978], Hevner y Yao [1979] y Epstein y Stonebraker [1980].

Selinger y Adiba [1980] y Daniels et al. [1982] describen el enfoque del procesamiento distribuido de las consultas adoptado por R* (una versión distribuida del Sistema R). Mackert y Lohman [1986] ofrecen una evaluación del rendimiento de los algoritmos de procesamiento de consultas en R*.

La optimización dinámica de las consultas en bases de datos múltiples se aborda en Ozcan et al. [1997]. Adali et al. [1996] y Papakonstantinou et al. [1996] describen los problemas de optimización de las consultas en los sistemas mediadores.

Los libros de texto Weltman y Dahbura [2000] y Howes et al. [1999] tratan LDAP. Kapitskaia et al. [2000] describen los problemas del almacenamiento en la caché de los datos de directorio de LDAP.

PARTE 8

Otros temas

El Capítulo 23 trata varios aspectos sobre el ajuste de rendimiento de sistemas de bases de datos para aumentar la velocidad de las aplicaciones. También describe las pruebas que se usan para medir el rendimiento de los sistemas de bases de datos comerciales. Después se describe el proceso de normalización y las normas de lenguajes de bases de datos existentes. Concluye con un análisis del papel de los sistemas de bases de datos en el comercio electrónico y los desafíos en el mantenimiento del acceso a datos almacenados en "sistemas heredados", y sobre la migración de las aplicaciones al nuevo sistema.

El Capítulo 24 describe los tipos de datos, tales como los datos temporales, los espaciales y los multimedia, y los aspectos de su almacenamiento en bases de datos. Las aplicaciones como la informática móvil y sus conexiones con las bases de datos también se describen en este capítulo.

Finalmente, en el Capítulo 25, se describen varias técnicas avanzadas de procesamiento de transacciones, incluidos los monitores de procesamiento de transacciones, los flujos de trabajo transaccionales, las transacciones de larga duración y las transacciones entre varias bases de datos.

CAPÍTULO 23

Desarrollo avanzado de aplicaciones

Se pueden distinguir varias tareas en el desarrollo de aplicaciones. En los Capítulos del 6 al 8 se estudió la forma de diseñar y desarrollar una aplicación. Uno de los aspectos del diseño de aplicaciones es el rendimiento que se espera de ellas. De hecho es común encontrar que al finalizar el desarrollo de una aplicación, funciona más lentamente de lo que se hubiera deseado, o gestiona menos transacciones por segundo de las que se requieren. Las aplicaciones que necesitan una cantidad de tiempo excesiva para realizar las acciones requeridas pueden causar el descontento de los usuario en el mejor de los casos y ser totalmente inutilizable en el peor.

Se puede hacer que las aplicaciones se ejecuten significativamente más rápido mediante el ajuste del rendimiento, que consiste en hallar y eliminar los cuellos de botella y en añadir el hardware adecuado, como puede ser memoria o discos. Los desarrolladores de aplicaciones pueden ajustar las aplicaciones de diversas formas, así como los administradores pueden acelerar el procesamiento de las aplicaciones.

Las normas son muy importantes para el desarrollo de las aplicaciones, especialmente en la época de Internet, dado que éstas necesitan comunicarse entre sí para llevar a cabo tareas útiles. Se han propuesto varias normas que afectan al desarrollo de las aplicaciones de bases de datos.

Los sistemas heredados son sistemas de aplicaciones que están desfasados y basados en tecnologías de generaciones anteriores. Sin embargo, suelen hallarse en el núcleo de las organizaciones y ejecutan aplicaciones con misiones críticas. Se describen aspectos de la definición de interfaces y de la migración con los sistemas heredados y el modo en que pueden sustituirse por otros sistemas.

23.1 Ajuste del rendimiento

El ajuste del rendimiento de un sistema implica ajustar varios parámetros y opciones de diseño para mejorar su rendimiento en una aplicación concreta. Existen varios aspectos del diseño de los sistemas de bases de datos que afectan al rendimiento de las aplicaciones (aspectos de alto nivel como el esquema y el diseño de las transacciones, parámetros de las bases de datos como los tamaños de la memoria intermedia y aspectos del hardware como el número de discos). Cada uno de estos aspectos puede ajustarse de modo que se mejore el rendimiento.

23.1.1 Localización de los cuellos de botella

El rendimiento de la mayor parte de los sistemas (al menos, antes de ajustarlos) suele quedar limitado principalmente por el que presenta un componente o unos pocos, denominados **cuellos de botella**. Por ejemplo, puede que un programa pase el ochenta por ciento del tiempo en un pequeño bucle ubicado en las profundidades del código y el veinte por ciento restante del tiempo en el resto del código; ese pequeño bucle es, pues, un cuello de botella. La mejora del rendimiento de un componente que no sea un cuello de botella hace poco para mejorar la velocidad global del sistema; en este ejemplo, la mejora

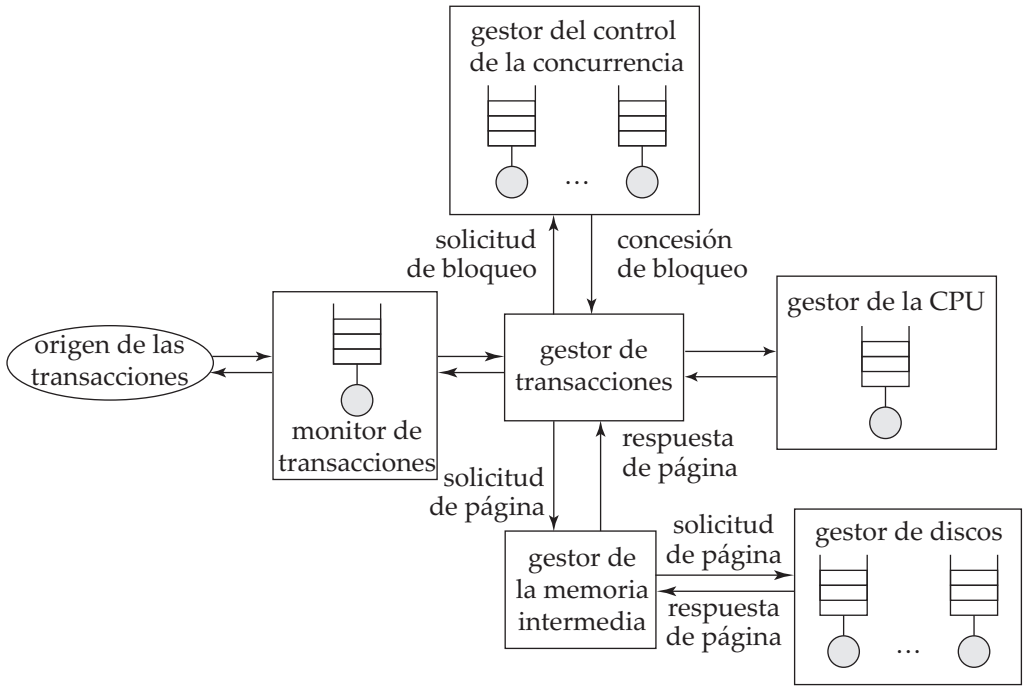


Figura 23.1 Colas de un sistema de bases de datos.

de la velocidad del resto del código no puede conducir a más de un veinte por ciento de mejora global, mientras que la mejora de la velocidad del bucle cuello de botella puede lograr una mejora de casi el ochenta por ciento global, en el mejor de los casos.

Por tanto, al ajustar un sistema, primero hay que intentar descubrir los cuellos de botella y luego eliminarlos mejorando el rendimiento de los componentes que los generan. Cuando se elimina un cuello de botella puede ocurrir que otro componente se transforme en cuello de botella. En los sistemas bien equilibrados ningún componente aislado constituye un cuello de botella. Si el sistema contiene cuellos de botella se infrautilizan los componentes que no forman parte de los cuellos de botella y quizás pudieran haberse sustituido por componentes más económicos de menores prestaciones.

En los programas sencillos el tiempo pasado en cada zona del código determina el tiempo global de ejecución. No obstante, los sistemas de bases de datos son mucho más complejos y pueden modelarse como **sistemas de colas**. Cada transacción necesita varios servicios del sistema de bases de datos, comenzando por la entrada en los procesos del servidor, las lecturas de disco durante la ejecución, los ciclos de la CPU y los bloqueos para el control de concurrencia. Cada uno de estos servicios tiene asociada una cola, y puede que las transacciones pequeñas pasen la mayor parte del tiempo esperando en las colas—especialmente en las colas de E/S de los discos—en lugar de ejecutando código. La Figura 23.1 muestra algunas de las colas de los sistemas de bases de datos.

Como consecuencia de las numerosas colas de la base de datos, los cuellos de botella de los sistemas de bases de datos suelen manifestarse en forma de largas colas para un servicio determinado o, de modo equivalente, en elevados índices de uso de un servicio concreto. Si las solicitudes se espacian de manera exactamente uniforme, y el tiempo para atender una solicitud es menor o igual que el tiempo antes de que llegue la siguiente solicitud, cada solicitud hallará el recurso sin usar y podrá iniciar la ejecución de manera inmediata, sin esperar. Por desgracia, la llegada de las solicitudes en los sistemas de bases de datos nunca es tan uniforme, más bien aleatoria.

Si un recurso (como puede ser un disco) tiene un índice de uso bajo, es probable que este recurso no se esté utilizando al realizar una solicitud, en cuyo caso el tiempo de espera de la solicitud será cero. Suponiendo llegadas distribuidas de forma aleatoria uniforme, la longitud de la cola (y, en consecuencia, el tiempo de espera) aumentará de manera exponencial con el uso; a medida que el uso se aproxime al cien por cien, la longitud de la cola aumentará abruptamente, lo que dará lugar a tiempos de espera excesivamente elevados. El uso de los recursos debe mantenerse lo suficientemente baja como para que

la longitud de la cola sea corta. Como indicativo, las usos cercanos al setenta por ciento se consideran buenas, y los superiores al noventa por ciento se consideran excesivas, dado que generan retrasos significativos. Para aprender más sobre la teoría de los sistemas de colas, generalmente conocida como **teoría de colas**, se pueden consultar las referencias citadas en las notas bibliográficas.

23.1.2 Parámetros ajustables

Los administradores de bases de datos pueden ajustar los sistemas de bases de datos en tres niveles. El nivel inferior es el nivel de hardware. Las opciones para el ajuste de los sistemas en este nivel incluyen añadir discos o usar sistemas RAID (si la E/S de disco constituye un cuello de botella), añadir más memoria si el tamaño de la memoria intermedia de disco constituye un cuello de botella o aumentar la velocidad del procesador si el uso de la CPU constituye un cuello de botella.

El segundo nivel consiste en los parámetros de los sistemas de bases de datos, como el tamaño de la memoria intermedia y los intervalos de puntos de revisión. El conjunto exacto de los parámetros de los sistemas de bases de datos que pueden ajustarse depende de cada sistema concreto de bases de datos. La mayor parte de los manuales de los sistemas de bases de datos proporcionan información sobre los parámetros del sistema de bases de datos que pueden ajustarse y sobre el modo en que deben escogerse los valores de esos parámetros. Los sistemas de bases de datos bien diseñados llevan a cabo automáticamente todos los ajustes posibles, lo que libera al usuario o al administrador de la base de datos de esa carga. Por ejemplo, en muchos sistemas de bases de datos el tamaño de la memoria intermedia es fijo pero ajustable. Si el sistema ajusta de manera automática el tamaño de la memoria intermedia observando los indicadores como las tasas de fallo de las páginas, el usuario no tendrá que preocuparse por el ajuste del tamaño de la memoria intermedia.

El tercer nivel es el nivel superior. Incluye el esquema y las transacciones. El administrador puede ajustar el diseño del esquema, los índices que se crean y las transacciones que se ejecutan para mejorar el rendimiento. El ajuste en este nivel es, comparativamente, independiente del sistema.

Los tres niveles de ajuste interactúan entre sí; hay que considerarlos en conjunto al ajustar los sistemas. Por ejemplo, el ajuste en un nivel superior puede hacer que el cuello de botella pase del sistema de discos a la CPU o viceversa.

23.1.3 Ajuste del hardware

Incluso en un sistema de procesamiento de transacciones bien diseñado, cada transacción suele tener que realizar al menos unas cuantas operaciones de E/S, si los datos necesarios para la transacción se hallan en el disco. Un factor importante en el ajuste de un sistema de procesamiento de transacciones, es asegurarse de que el subsistema de disco puede admitir la velocidad en que se solicitan las operaciones de E/S. Por ejemplo, considérese un disco con un tiempo de acceso de unos diez milisegundos y una velocidad media de transferencia de 25 megabytes por segundo (un disco habitual en la actualidad). Este disco soportaría un poco menos de cien operaciones E/S de acceso aleatorio de cuatro kilobytes cada segundo. Si cada transacción necesita exactamente dos operaciones E/S, cada disco soportará como mucho cincuenta transacciones por segundo. La única manera de soportar más transacciones por segundo es aumentar el número de discos. Si el sistema tiene que soportar n transacciones por segundo y cada una de ella lleva a cabo dos operaciones de E/S, hay que dividir (o fragmentar de otra manera) los datos al menos entre $n/50$ discos (ignorando el sesgo).

Hay que tener en cuenta que el factor limitador no es la capacidad del disco, sino la velocidad a la que se puede tener acceso a los datos aleatorios (limitada, a su vez, por la velocidad a la que se puede desplazar el brazo del disco). El número de operaciones de E/S por transacción puede reducirse almacenando más datos en la memoria. Si todos los datos se hallan en la memoria no habrá operaciones de E/S en el disco salvo por las operaciones de escritura. Guardar los datos usados con frecuencia en la memoria reduce el número de operaciones de E/S y compensa el coste extra de la memoria. Guardar los datos usados con muy poca frecuencia en la memoria sería un despilfarro, dado que la memoria es mucho más cara que los discos.

El problema es, para una cantidad dada de dinero disponible para gastarlo en discos o en memoria, hallar la mejor manera de gastar el dinero para obtener el número máximo de transacciones por

segundo. Una reducción de una operación de E/S por segundo ahorra

$$(\textit{precio por unidad de disco})/(\textit{accesos por segundo por disco})$$

Por tanto, si se tiene acceso a una página concreta n veces por segundo, el ahorro debido a guardarla en la memoria es n veces el valor calculado anteriormente. Guardar una página en la memoria cuesta

$$(\textit{precio por megabyte de memoria})/(\textit{páginas por megabyte de memoria})$$

Por tanto, el punto de equilibrio es

$$n * \frac{\textit{precio por unidad de disco}}{\textit{accesos por segundo por disco}} = \frac{\textit{precio por megabyte de memoria}}{\textit{páginas por megabyte de memoria}}$$

Se puede reordenar la ecuación y sustituir los valores actuales por cada uno de los parámetros citados más arriba para obtener un valor de n; si se tiene acceso a una página con una frecuencia mayor que ésta merece la pena comprar suficiente memoria como para almacenarla. La tecnología de discos y los precios actuales de los discos dan un valor de n de alrededor de $1/300$ veces por segundo (o, de manera equivalente, una vez cada cinco minutos) para las páginas a las que se tiene acceso de manera aleatoria.

Este razonamiento se refleja en la recomendación denominada **regla de los cinco minutos**: si una página se usa más de una vez cada cinco minutos se debe guardar en la caché de memoria. En otras palabras, merece la pena comprar suficiente memoria para guardar en la caché todas las páginas a las que se tiene acceso al menos una vez cada cinco minutos de promedio. Para los datos a los que se tiene acceso con menos frecuencia hay que comprar discos suficientes para soportar la tasa de E/S exigida por los datos.

La fórmula para hallar el punto de equilibrio depende de factores, como los costes de los discos y de la memoria, que han cambiado en factores de cien o de mil en la última década. No obstante, resulta interesante observar que los índices de los cambios han sido tales que el punto de equilibrio ha permanecido en unos cinco minutos; ¡la regla de los cinco minutos no se ha vuelto la regla de la hora o la regla del segundo!

Para los datos con acceso secuencial se puede leer un número de páginas por segundo significativamente mayor. Suponiendo que se lee cada vez un megabyte de datos, se obtiene la **regla del minuto**, que dice que los datos con acceso secuencial deben guardarse en la caché de memoria si se usan al menos una vez por minuto.

Las recomendaciones sólo tienen en cuenta el número de operaciones de E/S y no tienen en consideración factores como el tiempo de respuesta. Algunas aplicaciones necesitan guardar en la memoria incluso los datos que se usan con poca frecuencia para soportar tiempos de respuesta inferiores o similares al tiempo de acceso al disco.

Otro aspecto del ajuste es si se debe usar RAID 1 o RAID 5. La respuesta depende de la frecuencia con que se actualicen los datos, dado que RAID 5 es mucho más lento que RAID 1 en las operaciones aleatorias de escritura: RAID 5 necesita dos operaciones de lectura y dos operaciones de escritura para ejecutar una sola solicitud aleatoria de escritura. Si una aplicación realiza r operaciones aleatorias de lectura y e operaciones aleatorias de escritura por segundo para soportar un intercambio concreto, una implementación de RAID 5 necesitaría $r + 4e$ operaciones de E/S por segundo, mientras que una implementación de RAID 1 necesitaría $r + e$ operaciones de E/S por segundo. Se puede calcular el número de discos necesario para soportar las operaciones de E/S necesarias por segundo dividiendo el resultado del cálculo por cien operaciones de E/S por segundo (para los discos de la generación actual). Para muchas aplicaciones, r y e son lo bastante grandes como para que $(r + e)/100$ discos puedan guardar con facilidad dos copias de todos los datos. Para esas aplicaciones, si se usa RAID 1, ¡el número necesario de discos es realmente menor que el número necesario de discos si se usa RAID 5! Por tanto, RAID 5 sólo resulta útil cuando los requisitos de almacenamiento de datos son muy grandes, pero las velocidades de E/S y los requisitos de transferencia de datos son pequeños, es decir, para datos muy grandes y muy "fríos".

23.1.4 Ajuste del esquema

Dentro de las restricciones de la forma normal escogida es posible dividir las relaciones verticalmente. Por ejemplo, considérese la relación *cuenta*, con el esquema

cuenta (*número_cuenta*, *nombre_sucursal*, *saldo*)

para la que *número_cuenta* es una clave. Dentro de las restricciones de las formas normales (formas normales FNBC y tercera) se puede dividir la relación *cuenta* en dos relaciones:

sucursal_cuenta (*número_cuenta*, *nombre_sucursal*)
saldo_cuenta (*número_cuenta*, *saldo*)

Las dos representaciones son lógicamente equivalentes, dado que *número_cuenta* es una clave, pero tienen características de rendimiento diferentes.

Si la mayor parte de los accesos a la información de la cuenta sólo examinan *número_cuenta* y *saldo*, pueden ejecutarse sobre la relación *saldo_cuenta*, y es probable que el acceso resulte algo más rápido, dado que no se captura el atributo *nombre_sucursal*. Por el mismo motivo, cabrán en la memoria intermedia más tuplas de *saldo_cuenta* que las correspondientes tuplas de *cuenta*, lo que vuelve a generar un mayor rendimiento. Este efecto sería especialmente destacado si el atributo *nombre_sucursal* fuera de gran tamaño. Por tanto, un esquema que consistiera en *sucursal_cuenta* y *saldo_cuenta* sería preferible en este caso a otro que consistiera en la relación *cuenta*.

Por otro lado, si la mayor parte de los accesos a la información de la cuenta necesitan tanto *saldo* como *nombre_sucursal*, el uso de la relación *cuenta* será preferible, dado que se evitará el coste de la fusión de *saldo_cuenta* y *sucursal_cuenta*. Además, la sobrecarga de almacenamiento sería menor, dado que sólo habría una relación y no se replicaría el atributo *número_cuenta*.

Otro truco para mejorar el rendimiento es guardar una **relación desnormalizada**, como puede ser una fusión de *cuenta* y de *impositor*, donde la información sobre los nombres de las sucursales y sobre los saldos se repitiera para cada titular de una cuenta. Hay que realizar más esfuerzo para asegurarse de que la relación es consistente siempre que se realice una actualización. No obstante, una consulta que capture los nombres de los clientes y sus saldos asociados se aceleraría, dado que la fusión de *cuenta* con *impositor* se habría calculado previamente. Si se ejecuta con frecuencia una consulta de este tipo, y hay que llevarla a cabo con la máxima eficiencia posible, la relación desnormalizada puede resultar beneficiosa.

Las vistas materializadas pueden proporcionar las ventajas que ofrecen las relaciones desnormalizadas, al coste de algún almacenamiento extra; el ajuste del rendimiento de las vistas materializadas se describe en el Apartado 23.1.6. Una de las principales ventajas de las vistas materializadas respecto de las relaciones desnormalizadas es que el mantenimiento de la consistencia de los datos redundantes pasa a ser labor del sistema de bases de datos, no del programador. Por tanto, las vistas materializadas resultan preferibles, siempre que las soporte el sistema de bases de datos.

Otro enfoque de la aceleración del cálculo de la fusión sin materializarla es agrupar los registros que coincidirán en la fusión en la misma página del disco. Estas organizaciones agrupadas de archivos se vieron en el Apartado 11.7.2.

23.1.5 Ajuste de los índices

Se pueden ajustar los índices de un sistema de bases de datos para mejorar el rendimiento. Si las consultas constituyen el cuello de botella se las suele poder acelerar creando los índices adecuados en las relaciones. Si lo constituyen las actualizaciones, puede que haya demasiados índices, que hay que actualizar cuando se actualizan las relaciones. La eliminación de índices puede que acelere algunas actualizaciones.

La elección del tipo de índice también es importante. Algunos sistemas de bases de datos soportan diferentes tipos de índices, como los índices asociativos y los índices de árboles B. Si las consultas más frecuentes son de rango es preferible usar índices de árboles B a los índices asociativos. Otro parámetro ajustable es la posibilidad de hacer que un índice tenga agrupación. Sólo se puede hacer un índice con agrupación por relación, guardando la relación ordenada por los atributos del índice. Generalmente conviene hacer el índice con agrupación que beneficie al mayor número de consultas y de actualizaciones.

Para ayudar a identificar los índices que se deben crear y el índice (si es que hay alguno) de cada relación que se debe agrupar, la mayoría de sistemas de bases de datos comerciales proporcionan *asistentes para el ajuste*; se describen con más detalle en el Apartado 23.1.7. Estas herramientas usan el historial de consultas y de actualizaciones (denominado *carga de trabajo*) para estimar los efectos de varios índices en el tiempo de ejecución de las consultas y de las actualizaciones en la carga de trabajo. Las recomendaciones sobre los índices que se deben crear se basan en estas estimaciones.

23.1.6 Uso de vistas materializadas

El uso de las vistas materializadas puede acelerar enormemente ciertos tipos de consultas, en especial las consultas de agregación. Recuérdese el ejemplo del Apartado 14.5 en que el importe total de los créditos de cada sucursal (obtenido sumando los importes de los créditos de todos los créditos de la sucursal) se solicita con frecuencia. Como se vio en ese apartado, la creación de una vista materializada que guarde el importe total de los créditos de cada sucursal puede acelerar enormemente estas consultas.

Las vistas materializadas deben usarse con cuidado, no obstante, dado que no sólo supone una sobrecarga de espacio almacenarlas sino que, lo que es más importante, su mantenimiento también supone una sobrecarga de tiempo. En el caso del **mantenimiento inmediato de las vistas**, si las actualizaciones de una transacción afectan a la vista materializada, hay que actualizarla como parte de la misma transacción. Por tanto, puede que la transacción se ejecute más lentamente. En el caso del **mantenimiento diferido de las vistas**, la vista materializada se actualiza posteriormente; hasta que se actualice puede que la vista materializada sea inconsistente con las relaciones de la base de datos. Por ejemplo, puede que la vista materializada se actualice cuando la utilice una consulta o que se actualice de manera periódica. El uso del mantenimiento diferido reduce la carga de las transacciones de actualización.

Un problema importante es el modo en que se seleccionan las vistas materializadas que hay que mantener. El administrador del sistema puede realizar la selección de modo manual examinando los tipos de consultas de la carga de trabajo y averiguando las consultas que necesitan ejecutarse más rápidamente y las actualizaciones o consultas que pueden ejecutarse más lentamente. A partir del examen, el administrador del sistema puede escoger un conjunto adecuado de vistas materializadas. Por ejemplo, puede que el administrador descubra que se usa con frecuencia un agregado determinado y decida materializarlo, o puede que halle que una fusión concreta se calcula con frecuencia y decida materializarla.

Sin embargo, la selección manual resulta tediosa incluso para conjuntos de tipos de consultas moderadamente grandes y puede que resulte difícil realizar una buena selección, dado que exige comprender los costes de diferentes alternativas; sólo el optimizador de consultas puede estimar los costes con una precisión razonable, sin ejecutar realmente la consulta. Por tanto, puede que sólo se halle un buen conjunto de vistas mediante el procedimiento de prueba y error—es decir, materializando una o varias vistas, ejecutando la carga de trabajo y midiendo el tiempo usado para ejecutar las consultas de la carga de trabajo. El administrador repetirá el proceso hasta que se halle un conjunto de vistas que dé un rendimiento aceptable.

Una opción mejor es proporcionar soporte para la selección de las vistas materializadas desde el interior del propio sistema de bases de datos, integrado con el optimizador de consultas. Este enfoque se describe con más detalle en el Apartado 23.1.7.

23.1.7 Ajuste automático del diseño físico

La mayoría de sistemas comerciales de bases de datos actuales ofrecen herramientas para ayudar al administrador de la base de datos en la selección de los índices y de las vistas materializadas, y otras tareas relacionadas con el diseño de bases de datos tales como la división de datos en sistemas de bases de datos paralelos.

Estas herramientas examinan la carga de trabajo (el historial de consultas y de actualizaciones) y sugiere los índices y las vistas que hay que materializar. El usuario de la herramienta de ajuste tiene la posibilidad de especificar la importancia de la aceleración de las diferentes consultas, lo que el usuario tiene en cuenta al seleccionar las vistas que vaya a materializar. A menudo el ajuste se debe hacer antes de que se desarrolle completamente la aplicación, y el contenido real de la base de datos puede ser pequeño en la de desarrollo, pero se espera que sea mucho más grande en la de producción. Así, algunas

herramientas de ajuste también permiten que el usuario especifique la información sobre el tamaño previsto de la base de datos y de las estadísticas relacionadas.

Database Tuning Assistant de Microsoft, por ejemplo, permite que el usuario formule preguntas del tipo "¿qué pasaría si...?", por lo que el usuario puede escoger una vista y el optimizador estimará el efecto de materializarla en el coste total de la carga de trabajo y en los costes individuales de los diferentes tipos de consultas o de actualizaciones en la carga de trabajo.

Las técnicas de selección automática de índices y vistas materializadas se implementan generalmente enumerando varias alternativas y usando el optimizador de consultas para estimar los costes y las ventajas de seleccionar cada alternativa usando la carga de trabajo. Puesto que el número de los alternativas de diseño puede ser extremadamente grande, así como la carga de trabajo, las técnicas de la selección se deben diseñar cuidadosamente.

El primer paso es generar una carga de trabajo. Esto se hace generalmente registrando todas las consultas y actualizaciones que se ejecutan durante un cierto período. Después, las herramientas de selección realizan una *compresión de la carga de trabajo*, es decir, crean una representación de la carga de trabajo usando un número pequeño de actualizaciones y de consultas. Por ejemplo, las actualizaciones con la misma forma se pueden representar con una sola actualización con un peso que corresponde a cuántas veces ocurrió la actualización. Las consultas con la misma forma se pueden sustituir análogamente por un representante con el peso apropiado. Después de esto, las preguntas que son muy infrecuentes y no tienen un alto coste se pueden desechar del análisis. Las preguntas más costosas se pueden elegir para tratarlas en primer lugar. La compresión de la carga de trabajo es esencial para grandes cargas de trabajo.

Con la ayuda del optimizador, la herramienta calcularía un conjunto de índices y de vistas materializadas que mejorarían las consultas y actualizaciones en la compresión de la carga de trabajo. Se pueden probar diversas combinaciones de estos índices y vistas materializadas para encontrar la mejor combinación. Sin embargo, un enfoque exhaustivo sería impracticable, puesto que el número potencial de índices y vistas materializadas es ya grande, y cada subconjunto de éstos es un alternativa potencial de diseño, conduciendo a un número exponencial de alternativas. Se usan heurísticas para la reducción del espacio de alternativas, es decir, para reducir el número de las combinaciones consideradas.

Las heurísticas impacientes de la selección de índices y vistas materializadas operan como se explica a continuación. Se estiman las ventajas de la materialización de las diferentes vistas e índices (usando la estimación del coste del optimizador). Se escoge la vista o índice que ofrece la máxima ganancia o la máxima ventaja por unidad de espacio (es decir, la ventaja dividida por el espacio necesario para guardar la vista o el índice). El coste del mantenimiento de la vista o índice también se debe tomar en consideración al calcular la ganancia. Una vez que la heurística ha seleccionado una vista o índice puede que hayan cambiado las ganancias de otras vistas o índices, por lo que la heurística vuelve a calcularlas y escoge la segunda mejor vista para su materialización. El proceso continúa hasta que se agota el espacio de disco disponible para guardar las vistas materializadas e índices, o bien el coste del mantenimiento del resto de candidatos es superior a la ganancia de las consultas que los usen.

Las herramientas del mundo real para la selección de índices y vistas materializadas incorporan generalmente algunos elementos de la selección impaciente, pero usan otras técnicas para conseguir mejores resultados. También contemplan otros aspectos del diseño físico de bases de datos, tales como decidir la forma de dividir las relaciones en las bases de datos paralelas, o el mecanismo físico de almacenaje a usar para una relación.

23.1.8 Ajuste de las transacciones

En este apartado se estudian dos enfoques de la mejora del rendimiento de las transacciones:

- Mejora de la orientación a conjuntos.
- Reducción de la contención de los bloqueos.

En el pasado los optimizadores de muchos sistemas de bases de datos no eran especialmente buenos, por lo que el modo en que se había escrito la consulta tenía gran influencia en el modo en que se ejecutaba y, por tanto, en el rendimiento. Los optimizadores avanzados de hoy en día pueden transformar incluso las consultas mal escritas y ejecutarlas de manera eficiente, por lo que la necesidad de ajustar las consultas

una a una es menos importante que en el pasado. No obstante, las consultas complejas que contienen subconsultas anidadas no las suelen optimizar muy bien. La mayor parte de los sistemas proporcionan un mecanismo para averiguar el plan de ejecución preciso de las consultas; esta información puede usarse para volver a escribir la consulta de forma que el optimizador pueda trabajar mejor con ella.

En SQL incorporado, si se ejecuta con frecuencia una consulta con valores diferentes de un parámetro puede que resulte de ayuda combinar las llamadas en una consulta más orientada al conjunto que sólo se ejecute una vez. El coste de comunicación de las consultas SQL es generalmente elevado en los sistemas cliente-servidor, por lo que la combinación de las llamadas de SQL incorporado resulta de especial ayuda en esos sistemas

Por ejemplo, considérese un programa que pase por cada departamento especificado en una lista invocando una consulta de SQL para buscar los gastos totales del departamento usando la estructura **group by** en la relación *gastos* (*fecha*, *empleado*, *departamento*, *importe*). Si la relación *gastos* no tiene un índice con agrupación en *departamento*, cada consulta de ese tipo dará lugar a una exploración de la relación. En lugar de eso, se puede usar una sola consulta SQL para averiguar los gastos totales de todos los departamentos; la consulta puede evaluarse con una sola exploración. Los departamentos importantes pueden examinarse luego en esta relación temporal (mucho más pequeña) que contiene el agregado. Aunque haya un índice que permita el acceso eficiente a las tuplas de un departamento dado, el uso de varias consultas SQL puede suponer una gran sobrecarga de comunicaciones en los sistemas cliente−servidor. El coste de comunicaciones puede reducirse usando una sola consulta SQL, capturando los resultados para el lado cliente y pasando por los resultados para hallar las tuplas necesarias. Otra técnica muy usada en los sistemas cliente−servidor para reducir el coste de las comunicaciones y la compilación de SQL es usar procedimientos almacenados, en los que las consultas se guardan en el servidor en forma de procedimientos almacenados, que pueden estar compilados con antelación. Los clientes pueden invocar estos procedimientos almacenados en lugar de comunicar consultas enteras.

La ejecución concurrente de diferentes tipos de transacciones puede llevar a veces a un rendimiento bajo por la contención de los bloqueos. Considérese, por ejemplo, una base de datos bancaria. De día se ejecutan de manera casi continua numerosas transacciones de actualización de pequeño tamaño. Supóngase que se ejecuta al mismo tiempo una consulta de gran tamaño que calcula estadísticas de las sucursales. Si la consulta ejecuta una exploración de la relación puede que bloquee todas las actualizaciones de la relación mientras se ejecuta, y eso puede tener un efecto desastroso en el rendimiento del sistema.

Algunos sistemas de bases de datos (Oracle y SQL Serverde Microsoft, por ejemplo) permiten el control de concurrencia multiversión, por lo que las consultas se ejecutan sobre una instantánea de los datos, y las actualizaciones pueden seguir de manera concurrente. Esta característica debe usarse si está disponible. Si no se encuentra disponible, una opción alternativa es ejecutar las consultas de gran tamaño en momentos en que las actualizaciones sean pocas o no haya ninguna. Para las bases de datos que soporten los sitios Web puede que no exista ese periodo de calma para las actualizaciones.

Otra opción es usar niveles de consistencia más débiles, por lo que la evaluación de la consulta tendrá un impacto mínimo en las actualizaciones concurrentes, pero no se garantiza que los resultados de la consulta sean consistentes. La semántica de la aplicación determina si son aceptables las respuestas aproximadas (inconsistentes). Las aplicaciones mantienen generalmente contadores de secuencias actualizados por muchas transacciones, que pueden convertirse en puntos de la contención de bloqueos. El Ejercicio práctico 23.1 explora la forma en los contadores de secuencias de las bases de datos pueden ayudar a mejorar la concurrencia usando niveles más débiles de consistencia.

Las transacciones de actualización de larga duración pueden crear problemas de rendimiento con los registros del sistema e incrementar el tiempo que éste tarda en recuperarse de las caídas. Si una transacción lleva a cabo muchas actualizaciones puede que el registro del sistema se llene antes incluso de que se complete la transacción, en cuyo caso habrá que hacer retroceder la transacción. Si una transacción de actualización se ejecuta durante mucho tiempo (aunque tenga pocas actualizaciones) puede que bloquee la eliminación de las partes más antiguas del registro, si el sistema de registros no está bien diseñado. Nuevamente, este bloqueo puede llevar a que se llene el registro histórico.

Para evitar estos problemas muchos sistemas de bases de datos imponen límites estrictos para el número de actualizaciones que puede llevar a cabo una sola transacción. Aunque el sistema no imponga estos límites suele resultar de ayuda fraccionar las transacciones de actualización de gran tamaño en

conjuntos de transacciones de actualización de menor tamaño siempre que sea posible. Por ejemplo, una transacción que dé un aumento a cada empleado de una gran empresa puede dividirse en una serie transacciones de pequeño tamaño, cada una de las cuales es un pequeño rango de identificadores de empleados. Estas transacciones se denominan **transacciones procesadas por minilotes**. No obstante, las transacciones procesadas por minilotes deben usarse con cuidado. En primer lugar, si hay actualizaciones concurrentes en el conjunto de empleados puede que el resultado del conjunto de transacciones de menor tamaño no sea equivalente al de la transacción única de gran tamaño. En segundo lugar, si se produce un fallo las transacciones confirmadas habrán aumentado el salario de algunos empleados pero no el de los demás. Para evitar este problema, en cuanto el sistema se recupere del fallo, hay que ejecutar las restantes transacciones del lote.

23.1.9 Simulación del rendimiento

Para comprobar el rendimiento de un sistema de bases de datos incluso antes de instalarlo se puede crear un modelo de simulación del rendimiento de ese sistema. En la simulación se modela cada servicio que aparece en la Figura 23.1, como la CPU, cada disco, la memoria intermedia y el control de concurrencia. En lugar de modelar los detalles de un servicio, puede que el modelo de simulación sólo capture algunos aspectos de cada uno, como el **tiempo de servicio**—es decir, el tiempo que tarda en acabar de procesar una solicitud una vez comenzado el procesamiento. Por tanto, la simulación puede modelar el acceso a disco partiendo sólo del tiempo medio de acceso a disco.

Dado que las solicitudes de un servicio suelen tener que aguardar su turno, cada servicio tiene asociada una cola en el modelo de simulación. Cada transacción consiste en una serie de solicitudes. Las solicitudes se disponen en una cola según llegan y se atienden de acuerdo con la política de cada servicio, como puede ser que el primero en llegar sea el primero en ser atendido. Los modelos de los servicios como la CPU y los discos operan conceptualmente en paralelo, para tener en cuenta el hecho de que estos subsistemas operan en paralelo en los sistemas reales.

Una vez creado el modelo de simulación para el procesamiento de las transacciones, el administrador del sistema puede ejecutar en él varios experimentos. El administrador puede usar los experimentos con transacciones simuladas que lleguen con diferentes velocidades para averiguar el modo en que se comportaría el sistema bajo diferentes condiciones de carga. También puede ejecutar otros experimentos que varíen los tiempos de servicio de cada servicio para averiguar la sensibilidad del rendimiento a cada uno de ellos. También se pueden variar los parámetros del sistema de modo que se pueda realizar el ajuste del rendimiento en el modelo de simulación.

23.2 Pruebas de rendimiento

A medida que se van estandarizando los servidores de bases de datos, el factor diferenciador entre los productos de los diferentes fabricantes es el rendimiento de esos productos. Las **pruebas de rendimiento** son conjuntos de tareas que se usan para cuantificar el rendimiento de los sistemas de software.

23.2.1 Familias de tareas

Dado que la mayor parte de los sistemas de software, como las bases de datos, son complejos hay bastante variación en su implementación por parte de los diferentes fabricantes. En consecuencia, hay una variación significativa en su rendimiento en las diferentes tareas. Puede que un sistema sea el más eficiente en una tarea concreta y puede que otro lo sea en una tarea diferente. Por tanto, una sola tarea no suele resultar suficiente para cuantificar el rendimiento del sistema. En lugar de eso, el rendimiento de un sistema se mire mediante familias de tareas estandarizadas, denominados *pruebas de rendimiento*.

La combinación de los resultados de rendimiento de varias tareas debe realizarse con cuidado. Supóngase que se tienen dos tareas, T_1 y T_2, y que se mide la productividad de un sistema como el número de transacciones de cada tipo que se ejecutan en un tiempo dado (digamos, un segundo). Supóngase que el sistema A ejecuta T_1 a noventa y nueve transacciones por segundo y que T_2 se ejecuta a una transacción por segundo. De manera parecida, supóngase que el sistema B ejecuta T_1 y T_2 a cincuenta transacciones por segundo. Supóngase también que una carga de trabajo tiene una mezcla a partes iguales de los dos tipos de transacciones.

Si se toma el promedio de los dos pares de resultados (es decir, noventa y nueve y uno frente a cincuenta y cincuenta), pudiera parecer que los dos sistemas tienen el mismo rendimiento. Sin embargo, sería *erróneo* tomar los promedios de esta manera (si se ejecutaran cincuenta transacciones de cada tipo el sistema *A* tardaría unos 50.5 segundos en concluirlas, mientras que el sistema *B* las terminaría ¡en sólo 2 segundos!).

Este ejemplo muestra que una sola medida del rendimiento induce a error si hay más de un tipo de transacción. El modo correcto de promediar los números es tomar el **tiempo para concluir** la carga de trabajo, en vez de la **productividad** promedio de cada tipo de transacción. Se puede así calcular el rendimiento del sistema en transacciones por segundo para una carga de trabajo concreta con exactitud. Por tanto, el sistema A tarda $50.5/100$, que son 0.505 segundos por transacción, mientras que el sistema B tarda 0.02 segundos por transacción, en promedio. En términos de productividad, el sistema A se ejecuta a un promedio de 1.98 transacciones por segundo, mientras que el sistema B se ejecuta a 50 transacciones por segundo. Suponiendo que las transacciones de todos los tipos son igual de probables el modo correcto de promediar la productividad respecto de los diferentes tipos de transacciones es tomar la **media armónica** de las productividades. La media armónica de n productividades $f_1, \ldots, f_n$ se define como

$$\frac{n}{\frac{1}{f_1} + \frac{1}{f_2} + \cdots + \frac{1}{f_n}}$$

Para este ejemplo la media armónica de las productividades en el sistema A es 1.98. Para el sistema B es 50. Por tanto, el sistema B es aproximadamente veinticinco veces más rápido que el sistema A para una carga de trabajo consistente en una mezcla a partes iguales de los dos tipos de transacciones de ejemplo.

23.2.2 Clases de aplicaciones de bases de datos

El **procesamiento en conexión de transacciones** (Online Transaction Processing, OLTP) y la **ayuda a la toma de decisiones** (incluyendo el **procesamiento en conexión analítico** (Online Analytical Processing, OLAP)) son dos grandes clases de aplicaciones manejadas por los sistemas de bases de datos. Estas dos clases de tareas tienen necesidades diferentes. La elevada concurrencia y las técnicas inteligentes para acelerar el procesamiento de las operaciones de compromiso se necesitan para soportar una elevada tasa de transacciones de actualización. Por otro lado, los buenos algoritmos para la evaluación de consultas y la optimización de las consultas son necesarios para la ayuda a la toma de decisiones. La arquitectura de algunos sistemas de bases de datos se ha ajustado para el procesamiento de las transacciones; la de otros, como la serie DBC de Teradata de sistemas paralelos de bases de datos, para el ayuda a la toma de decisiones. Otros fabricantes intentan conseguir un equilibrio entre las dos tareas.

Las aplicaciones suelen tener una mezcla de necesidades de procesamiento de transacciones y ayuda a la toma de decisiones. Por tanto, el mejor sistema de bases de datos para cada aplicación depende de la mezcla de las dos necesidades que tenga la aplicación.

Supóngase que se tienen resultados de productividad para las dos clases de aplicaciones por separado y la aplicación en cuestión tiene una mezcla de transacciones de las dos clases. Hay que ser precavido incluso al tomar la media armónica de los resultados de productividad, debido a la **interferencia** entre las transacciones. Por ejemplo, una transacción de ayuda a la toma de decisiones que tarde mucho en ejecutarse puede adquirir varios bloqueos, lo que puede evitar el progreso de las transacciones de actualización. La media armónica de las productividades sólo debe usarse si las transacciones no interfieren entre sí.

23.2.3 Las pruebas de rendimiento TPC

El **Consejo para el rendimiento del procesamiento de las transacciones** (Transaction Processing Performance Council, TPC) ha definido una serie de normas de pruebas de rendimiento para los sistemas de bases de datos.

Las pruebas TPC se definen con gran minuciosidad. Definen el conjunto de relaciones y el tamaño de las tuplas. Definen el número de tuplas de las relaciones no como un número fijo, sino como un

múltiplo del número de transacciones por segundo que se afirma que se realizan, para reflejar que una tasa mayor de ejecución de transacciones probablemente se halle correlacionada con un número mayor de cuentas. La métrica del rendimiento es la productividad, expresado como **transacciones por segundo** (**TPS**). Cuando se mide el rendimiento, el sistema debe proporcionar un tiempo de respuesta que se halle dentro de ciertos límites, de modo que una productividad elevada no pueda obtenerse a expensas de tiempos de respuesta muy elevados. Además, para las aplicaciones profesionales, el coste es de gran importancia. Por tanto, la prueba TPC también mide el rendimiento en términos de **precio por TPS**. Puede que los sistemas de gran tamaño tengan un elevado número de transacciones por segundo, pero puede que resulten caros (es decir, que tengan un precio elevado por TPS). Además, una compañía no puede afirmar que tiene resultados de las pruebas TPC en sus sistemas *sin* una auditoría externa que asegure que el sistema sigue fielmente la definición de la prueba, incluyendo el soporte pleno de las propiedades ACID de las transacciones.

La primera de la serie fue la prueba **TPC-A**, que se definió en 1989. Esta prueba simula una aplicación bancaria típica mediante un solo tipo de transacción que modela la retirada y el depósito de efectivo en un cajero automático. La transacción actualiza varias relaciones—como el saldo del banco, el saldo del cajero y el saldo del cliente—y añade un registro a una relación de seguimiento para la auditoría. La prueba también incorpora la comunicación con los terminales para modelar el rendimiento de extremo a extremo del sistema de manera realista. La prueba **TPC-B** se diseñó para probar el rendimiento central del sistema de bases de datos, junto con el sistema operativo en el que se ejecuta el sistema. Elimina las partes de la prueba TPC-A que tratan de los usuarios, de las comunicaciones y de los terminales para centrarse en el servidor de bases de datos dorsal. Ni TPC-A ni TPC-B se usan mucho hoy en día.

La prueba **TPC-C** se diseñó para modelar un sistema más complejo que el de la prueba TPC-A. La prueba TPC-C se concentra en las actividades principales de un entorno de admisión de pedidos, como son la entrada y la entrega de pedidos, el registro de los pagos, la verificación del estado de los pedidos y el seguimiento de los niveles de inventarios. La prueba TPC-C se sigue usando con profusión para el procesamiento de transacciones.

La prueba **TPC-D** se diseñó para probar el rendimiento de los sistemas de bases de datos en consultas de ayuda a la toma de decisiones. Los sistemas de ayuda a la toma de decisiones se están haciendo cada vez más importantes hoy en día. Las pruebas TPC-A, TPC-B y TPC-C miden el rendimiento de las cargas de procesamiento de transacciones y no deben usarse como medida del rendimiento en consultas de ayuda a la toma de decisiones. La D de TPC-D viene de **ayuda a la toma de decisiones**. El esquema de la prueba TPC-D modela una aplicación de ventas/distribución, con componentes, clientes y pedidos, junto con cierta información auxiliar. El tamaño de las relaciones se define como una relación y el tamaño de la base de datos es el tamaño total de todas las relaciones, expresado en gigabytes. TPC-D con un factor de escala de uno representa la prueba TPC-D para una base de datos de un gigabyte, mientras que el factor de escala diez representa una base de datos de diez gigabytes. La carga de trabajo de la prueba consiste en un conjunto de diecisiete consultas SQL que modelan tareas que se ejecutan frecuentemente en los sistemas de ayuda a la toma de decisiones. Parte de las consultas usan características complejas de SQL, como las consultas de agregación y las consultas anidadas.

Los usuarios de las pruebas se dieron cuenta pronto de que las diferentes consultas TPC-D podían acelerarse de manera significativa usando las vistas materializadas y otra información redundante. Hay aplicaciones, como las tareas periódicas de información, donde las consultas se conocen con antelación y las vistas materializadas pueden seleccionarse con cuidado para acelerar las consultas. Resulta necesario, no obstante, tener en cuenta la sobrecarga que supone mantener las vistas materializadas.

La prueba **TPC-R** (donde la R viene de **informar**—Reporting) supone un refinamiento de la prueba TPC-D. El esquema es el mismo, pero hay veintidós consultas, de las cuales dieciséis provienen de TPC-D. Además, hay dos actualizaciones, un conjunto de inserciones y un conjunto de eliminaciones. Se permite que la base de datos que ejecuta la prueba utilice vistas materializadas y otra información redundante.

A diferencia de esto, la prueba **TPC-H** (donde la H representa **ad hoc**) usa el mismo esquema y la misma carga de trabajo que TPC-R pero prohíbe las vistas materializadas y otra información redundante y permite las pruebas sólo en las claves principales y ajenas. Esta prueba modela consultas ad hoc donde las consultas no se conocen con anterioridad, por lo que no es posible crear con antelación vistas materializadas adecuadas.

Tanto TPC-H como TPC-R miden el rendimiento de esta manera: el **test de potencia** ejecuta las consultas y las actualizaciones una a una de manera secuencial y tres mil seiscientos segundos divididos por la media geométrica de los tiempos de ejecución de las consultas (en segundos) da una medida de las consultas por hora. El **test de productividad** ejecuta varias corrientes en paralelo, cada una de las cuales ejecuta las veintidós consultas. También hay una corriente de actualizaciones paralela. Aquí el tiempo total de toda la ejecución se usa para calcular el número de consultas por hora.

La **métrica compuesta consultas por hora**, que es la métrica global, se obtiene como la raíz cuadrada del producto de las métricas de potencia y de productividad. Se define una **métrica compuesta precio/rendimiento** dividiendo el precio del sistema por la métrica compuesta.

La prueba de comercio Web **TPC-W** es una prueba de extremo a extremo que modela los sitios Web que tienen contenido estático (imágenes, sobre todo) y contenido dinámico generado a partir de una base de datos. Se permite de manera explícita el almacenamiento en caché del contenido dinámico, dado que resulta muy útil para acelerar los sitios Web. La prueba modela una librería electrónica y, como otras pruebas TPC, proporciona diferentes factores de escala. La métrica de rendimiento principal son las **interacciones Web por segundo (Web Interactions Per Second, WIPS)** y el precio por WIPS.

23.2.4 Las pruebas de rendimiento OODB

La naturaleza de las aplicaciones de las bases de datos orientadas a objetos (OODB, Object Oriented Databases) es diferente de las de las aplicaciones típicas de procesamiento de transacciones. Por tanto, se ha propuesto un conjunto diferente de pruebas para las OODBs.

La prueba operaciones con objetos, versión 1, popularmente conocida como prueba **OO1**, fue una de las primeras propuestas. La prueba **OO7** sigue una filosofía diferente de la de las pruebas TPC. Las pruebas TPC proporcionan uno o dos resultados (en términos del promedio de transacciones por segundo y de transacciones por segundo y por dólar); la prueba OO7 proporciona un conjunto de resultados, que contienen un resultado de prueba independiente para cada una de las diferentes clases de operaciones. El motivo de este enfoque es que no está todavía claro lo que es la transacción OODB *típica*. Está claro que esa transacción llevará a cabo ciertas operaciones, como recorrer un conjunto de objetos conectados o recuperar todos los objetos de una clase, pero no está claro exactamente la mezcla de tales operaciones que se usará. Por tanto, la prueba proporciona resultados separados para cada clase de operaciones; los resultados pueden combinarse de la manera adecuada, en función de la aplicación concreta.

23.3 Normalización

Las **normas** definen las interfaces de los sistemas de software; por ejemplo, las normas definen la sintaxis y la semántica de los lenguajes de programación o las funciones en la interfaz de los programas de aplicaciones o, incluso, los modelos de datos (como las normas de las bases de datos orientadas a los objetos). Hoy en día los sistemas de bases de datos son complejos y suelen estar constituidos por varias partes creadas de manera independiente que deben interactuar entre sí. Por ejemplo, puede que los programas clientes se creen de manera independiente de los sistemas dorsales, pero todos ellos deben poder interactuar entre sí. Puede que una empresa que tenga varios sistemas de bases de datos heterogéneos necesite intercambiar datos entre las bases de datos. En una situación de este tipo las normas desempeñan un papel importante.

Las **normas formales** son las que han sido desarrolladas por una organización de normalización o por grupos de empresas mediante un procedimiento público. Los productos dominantes se convierten a veces en una **norma de facto**, en el sentido de que resultan aceptados de manera general como normas sin necesidad de ningún procedimiento formal de reconocimiento. Algunas normas formales, como muchos aspectos de las normas SQL-92 y SQL:1999 son **normas anticipativas** que lideran el mercado; definen las características que los fabricantes implementan posteriormente en los productos. En otros casos, las normas, o partes de las normas, son **normas reaccionarias**, en el sentido de que intentan normalizar las características que ya han implementado algunos fabricantes, y que pueden haberse convertido, incluso, en normas de facto. SQL-89 era, en muchos sentidos, reaccionario, dado que estandarizaba características, como la comprobación de la integridad, que ya estaban presentes en la norma SAA SQL de IBM y en otras bases de datos.

Los comités para las normas formales suelen estar compuestos por representantes de los fabricantes y por miembros de grupos de usuarios y de organizaciones de normalización como la Organización internacional de normalización (International Organization for Standardization, ISO) o el Instituto nacional americano de normalización (American National Standards Institute, ANSI) o de organismos profesionales como el Instituto de ingenieros eléctricos y electrónicos (Institute of Electrical and Electronics Engineers, IEEE). Los comités para las normas formales se reúnen de manera periódica y sus componentes presentan propuestas de características de la norma que hay que añadir o modificar. Tras un periodo de discusión (generalmente amplio), de modificaciones de la propuesta y de examen público, los integrantes de los comités votan la aceptación o el rechazo de las características. Tras cierto tiempo de haber definido e implementado de una norma, quedan claras sus carencias y aparecen nuevas necesidades. El proceso de actualización de la norma comienza entonces y tras unos cuantos años suele publicarse una nueva versión de la misma. Este ciclo suele repetirse cada pocos años hasta que, finalmente (quizás muchos años más tarde) la norma se vuelve tecnológicamente irrelevante o pierde su base de usuarios.

La norma CODASYL de DBTG para bases de datos en red, formulada por el Grupo de trabajo para bases de datos (Database Task Group, DBTG) fue una de las primeras normas formales en este campo. Los productos de bases de datos de IBM solían establecer las normas de facto, dado que IBM ocupaba gran parte de este mercado. Con el crecimiento de las bases de datos relacionales aparecieron nuevos competidores en el negocio de las bases de datos; por tanto, surgió la necesidad de normas formales. En los últimos años Microsoft ha creado varias especificaciones que también se han convertido en normas de facto. Un ejemplo destacable es ODBC, que se usa actualmente en entornos que no son de Microsoft. JDBC, cuya especificación la creó Sun Microsystems, es otra norma de facto muy usada.

Este apartado ofrece una introducción de muy alto nivel a las diferentes normas, concentrándose en los objetivos de cada norma. Las notas bibliográficas al final del capítulo ofrecen referencias de las descripciones detalladas de las normas mencionados en este apartado.

23.3.1 Normas de SQL

Dado que SQL es el lenguaje de consultas más usado se ha trabajado mucho en su normalización. ANSI e ISO, con los diferentes fabricantes de bases de datos, han desempeñado un papel protagonista en esta labor. La norma SQL-86 fue la versión inicial. La norma Arquitectura de aplicaciones de sistemas (Systems Application Architecture, SAA) de IBM para SQL se publicó en 1987. A medida que la gente identificaba la necesidad de más características se desarrollaron versiones actualizadas de la norma formal de SQL, denominadas SQL-89 y SQL-92.

La versión SQL:1999 de la norma SQL añadió varias características al lenguaje. Ya se han visto muchas de estas características en los capítulos anteriores. La versión SQL:2003 de la norma SQL es una extenxión menor de la norma SQL:1999. Algunas de las características OLAP (Apartado 18.2.3) de SQL:1999 se especificaron como una enmienda en lugar de esperar a la versión de SQL:2003.

La norma SQL:2003 se ha dividido en varias partes:

- Parte 1: SQLFramework proporciona una introducción a la norma.
- Parte 2: SQL/Foundation define los fundamentos de la norma: tipos, esquemas, tablas, vistas, consultas y sentencias de actualización, expresiones, modelo de seguridad, predicados, reglas de asignación, gestión de transacciones, etc.
- Parte 3: SQL/CLI (Call Level Interface) define las interfaces de los programas de aplicaciones con SQL.
- Parte 4: SQL/PSM (Persistent Stored Modules) define las extensiones de SQL para hacerlo procedimental.
- Parte 9: SQL/MED (Management of External Data) define las normas para la realización de interfaces en los sistemas SQL con orígenes externos. Al escribir envolturas, los diseñadores de los sistemas pueden tratar los orígenes externos de datos, como pueden ser los archivos o los datos de bases de datos no relacionales, como si fueran tablas "externas".
- Parte 10: SQL/OLB (Object Language Bindings) define las normas para la incrustación de SQL en Java.

- Parte 11: SQL/Schemata (Information and Definition Schema) define una interfaz estándar para el catálogo.
- Parte 13: SQL/JRT (Java Routines and Types) define una norma para el acceso a rutinas y tipos de Java.
- Parte 14: SQL/XML define especificaciones relacionadas con XML.

Los números que faltan tratan características como los datos temporales, el procesamiento de transacciones distribuidas y los datos multimedia para las que todavía no hay ningún acuerdo sobre las normas.

23.3.2 Normas de conectividad de las bases de datos

La norma **ODBC** es una norma muy usada para la comunicación entre las aplicaciones clientes y los sistemas de bases de datos. ODBC se basa en las normas SQL **Interfaz de nivel de llamada** (Call Level Interface, CLI) desarrollados por el consorcio industrial *X/Open* y el Grupo SQL Access, pero tienen varias extensiones. La API ODBC define una CLI, una definición de sintaxis SQL y reglas sobre las secuencias admisibles de llamadas CLI. La norma también define los niveles de conformidad para la CLI y la sintaxis SQL. Por ejemplo, el nivel central de la CLI tiene comandos para conectarse con bases de datos, para preparar y ejecutar sentencias SQL, para devolver resultados o valores de estado y para administrar transacciones. El siguiente nivel de conformidad (nivel uno) exige el soporte de la recuperación de información de los catálogos y otras características que superan la CLI del nivel central; el nivel dos exige más características, como la capacidad de enviar y recuperar arrays de valores de parámetros y de recuperar información de catálogo más detallada.

ODBC permite que un cliente se conecte de manera simultánea con varios orígenes de datos y que conmute entre ellos, pero las transacciones en cada uno de ellos son independientes entre sí; ODBC no soporta el compromiso de dos fases.

Los sistemas distribuidos ofrecen un entorno más general que los sistemas cliente–servidor. El consorcio X/Open también ha desarrollado las normas **X/Open XA** para la interoperación de las bases de datos. Estas normas definen las primitivas de gestión de las transacciones (como pueden ser el comienzo de las transacciones, su compromiso, su anulación y su preparación para el compromiso) que deben proporcionar las bases de datos que cumplan con la norma; los administradores de las transacciones pueden invocar estas primitivas para implementar las transacciones distribuidas mediante compromiso de dos fases. Las normas XA son independientes de los modelos de datos y de las interfaces concretas entre los clientes y las bases de datos para el intercambio de datos. Por tanto, se pueden usar los protocolos XA para implementar sistemas de transacciones distribuidas en los que una sola transacción pueda tener acceso a bases de datos relacionales y orientadas a objetos y que el administrador de transacciones asegure la consistencia global mediante el compromiso de dos fases.

Hay muchos orígenes de datos que no son bases de datos relacionales y, de hecho, puede que no sean ni siquiera bases de datos. Ejemplos de esto son los archivos planos y los almacenes de correo electrónico. **OLE-DB** de Microsoft es una API de C++ con objetivos parecidos a los de ODBC, pero para orígenes de datos que no son bases de datos y que puede que sólo proporcionen servicios limitados de consulta y de actualización. Al igual que ODBC, OLE-DB proporciona estructuras para la conexión con orígenes de datos, el inicio de una sesión, la ejecución de comandos y la devolución de resultados en forma de conjunto de filas, que es un conjunto de filas de resultados.

Sin embargo, OLE-DB se diferencia de ODBC en varios aspectos. Para dar soporte a los orígenes de datos con soporte de características limitadas, las características de OLE-DB se dividen entre varias interfaces y cada origen de datos sólo puede implementar un subconjunto de esas interfaces. Los programas OLE-DB pueden negociar con los orígenes de datos para averiguar las interfaces que soportan. En ODBC los comandos siempre están en SQL. En OLE-DB los comandos pueden estar en cualquier lenguaje soportado por el origen de datos; aunque puede que algunos orígenes de datos soporten SQL, o un subconjunto limitado de SQL, puede que otros orígenes sólo ofrezcan posibilidades sencillas como el acceso a los datos de los archivos planos, sin ninguna capacidad de consulta. Otra diferencia importante de OLE-DB con ODBC es que los conjuntos de filas son objetos que pueden compartir varias aplicaciones

mediante la memoria compartida. Los objetos conjuntos de filas los puede actualizar una aplicación, y a las otras aplicaciones que compartan ese objeto se les notificará la modificación.

La API **objetos activos de datos** (Active Data Objects, ADO), también creada por Microsoft, ofrece una interfaz sencilla de usar con la funcionalidad OLE-DB que puede llamarse desde los lenguajes de guiones, como VBScript y JScript. La API más reciente **ADO.NET** se ha diseñado para las aplicaciones escritas en los lenguaje de .NET; por ejemplo, C# y Visual Basic.NET. Además de proporcionar interfaces simplificados, proporciona una abstracción llamada el *DataSet* que permite el acceso desconectado a los datos.

23.3.3 Normas de las bases de datos de objetos

Hasta ahora las normas en el área de las bases de datos orientadas a objetos las han impulsado fundamentalmente todo los fabricantes de OODB. El *Grupo de gestión de bases de datos de objetos* (Object Database Management Group, ODMG) fue un grupo formado por fabricantes de BDOO para normalizar los modelos de datos y las interfaces de lenguaje con las OODBs. La interfaz del lenguaje C++ especificada por ODMG se describió brevemente en el Capítulo 9. ODMG ya no está activo, JDO es una norma para añadir persistencia a Java.

El *Grupo de administración de objetos* (Object Management Group, OMG) es un consorcio de empresas, formado con el objetivo de desarrollar una arquitectura estándar para las aplicaciones de software distribuido basadas en el modelo orientado a objetos. OMG creó el modelo de referencia *Arquitectura de gestión de objetos* (Object Management Architecture, OMA). El *Agente para solicitudes de objetos* (Object Request Broker, ORB) es un componente de la arquitectura OMA que proporciona de manera transparente la entrega de mensajes a los objetos distribuidos, de modo que la ubicación física del objeto no tiene importancia. La **Arquitectura común de agente para solicitudes de objetos** (Common Object Request Broker Architecture, CORBA) proporciona una especificación detallada de ORB, e incluye un **Lenguaje de descripción de interfaces** (Interface Description Language, IDL), que se usa para definir los tipos de datos usados para el intercambio de datos. IDL ayuda a dar soporte a la conversión de datos cuando se intercambian entre sistemas con diferentes representaciones de los datos.

23.3.4 Normas basadas en XML

Se ha definido una amplia variedad de normas basadas en XML (véase el Capítulo 10) para gran número de aplicaciones. Muchas de estas normas están relacionadas con el comercio electrónico. Entre ellas hay normas promulgadas por consorcios sin ánimo de lucro y esfuerzos por parte de las empresas para crear normas de facto.

RosettaNet, que pertenece a la primera categoría, es un consorcio industrial que usa normas basadas en XML para facilitar la gestión de las cadenas de abastecimiento en las industrias informáticas y de tecnologías de la información. Las cadenas de abastecimiento se refieren a las compras del material y de los servicios que una organización necesita para funcionar. En cambio, la gestión de la relación de los clientes se refiere a la parte visible de la interacción con los clientes de una empresa. Las cadenas de abastecimiento requieren la normalización de varias cosas, tales como:

- **Identificador global de la compañía.** RosettaNet especifica un sistema para identificar unívocamente a las compañías, usando un identificador de 9 dígitos denominado *sistema de numeración universal de datos* (Data Universal Numbering System, DUNS).
- **Identificador global de productos.** RosettaNet especifica el número de 14 dígitos *número global de productos comerciales* (Global Trade Item Number, GTIN) para identificar productos y servicios.
- **Identificador global de clases.** Se trata de un código jerárquico de 10 dígitos para clasificar productos y servicios denominado *código estándar de productos y servicios de las Naciones Unidas* (United Nations/Standard Product and Services Code, UN/SPSC).
- **Interfaces entre los socios implicados.** Los *procesos de interfaz entre socios* (Partner Interface Processes, PIPs) de RosettaNet definen procesos de negocio entre socios. Estos procesos son diálogos entre sistemas basados en XML: definen los formatos y la semántica de los documentos de negocio implicados en el proceso y los pasos implicados en terminar una transacción. Algunos

ejemplos de pasos serían conseguir la información de producto y de servicio, los pedidos de compras, facturación de los pedidos, pagos, peticiones del estado de los pedidos, gestión de inventarios, soporte postventa incluyendo garantía del servicio, etc. El intercambio de la información del diseño, configuración, procesos y calidad también es posible para coordinar entre diferentes organizaciones las actividades de fabricación.

Los participantes en los mercados electrónicos pueden guardar los datos en gran variedad de sistemas de bases de datos. Puede que estos sistemas utilicen diferentes modelos, formatos y tipos de datos. Además, puede que haya diferencias semánticas (sistema métrico frente a sistema imperial británico, distintas divisas, etc.) en los datos. Las normas para los mercados electrónicos incluyen los métodos para *envolver* cada uno de estos sistemas heterogéneos con un esquema XML. Estas *envolturas* XML forman la base de una vista unificada de los datos para todos los participantes del mercado.

El *protocolo simple de acceso a objetos* (Simple Object Access Protocol, SOAP) es una norma para llamadas a procedimientos remotos que usa XML para codificar los datos (tanto los parámetros como los resultados) y usa HTTP como protocolo de transporte; es decir, las llamadas a procedimientos se transforman en solicitudes HTTP. SOAP está apoyado por el Consorcio World Wide Web (World Wide Web Consortium, W3C) y ha logrado una amplia aceptación en la industria. SOAP puede usarse en gran variedad de aplicaciones. Por ejemplo, en el comercio electrónico entre empresas, comercio electrónico, las aplicaciones que se ejecutan en un sitio pueden tener acceso a los datos de otros sitios y ejecutar acciones mediante SOAP.

SOAP y los servicios Web se tratan con más detalle en el Apartado 10.7.3.

23.4 Migración de aplicaciones

Los **sistemas heredados** son sistemas de aplicaciones de generaciones anteriores en uso, pero que la organización desea sustituir por otros. Por ejemplo, muchas organizaciones desarrollaron aplicaciones, pero pueden decidir sustituirlas por productos comerciales. En algunos casos un sistema heredado puede usar tecnologías antiguas incompatibles con las normas y sistemas de la generación actual. Algunos sistemas heredados operativos tienen varias décadas y se basan hoy en tecnologías tales como bases de datos de red o jerárquicas, o usan Cobol y sistemas de ficheros sin base de datos. Estos sistemas pueden contener datos valiosos y pueden dar soporte a aplicaciones críticas.

El reemplazamiento de las aplicaciones heredadas por aplicaciones más modernas suele resultar costoso tanto en términos de tiempo como de dinero, dado que suelen ser de tamaño muy grande, con millones de líneas de código desarrolladas por equipos de programadores a lo largo de varias décadas. Contienen grandes cantidades de datos que se deban trasladar a la nueva aplicación, que puede usar un esquema totalmente diferente. El cambio de la aplicación vieja a la nueva implica volver a formar a una gran cantidad de personal. El cambio se debe hacer generalmente sin ninguna interrupción, con los datos incorporados en el viejo sistema disponibles también en el nuevo.

Muchas organizaciones intentan evitar reemplazar los sistemas heredados, intentando interoperar con los nuevos sistema. Un enfoque usado para interoperar entre las bases de datos relacionales y las bases de datos heredadas es crear una capa, denominada **envoltura**, por encima de los sistemas heredados que pueda hacer que el sistema heredado parezca una bases de datos relacional. Puede que la envoltura ofrezca soporte para ODBC u otras normas de interconexión como OLE-DB, que puede usarse para consultar y actualizar el sistema heredado. La envoltura es responsable de la conversión de las consultas y actualizaciones relacionales en consultas y actualizaciones del sistema heredado.

Cuando una organización decide sustituir un sistema heredado por otro nuevo puede seguir un proceso denominado **ingeniería inversa**, que consiste en repasar el código del sistema heredado para obtener el diseño de los esquemas del modelo de datos requerido (como puede ser el modelo E-R o un modelo de datos orientado a los objetos). La ingeniería inversa también examina el código para averiguar los procedimientos y los procesos que se implementan con objeto de obtener un modelo de alto nivel del sistema. La ingeniería inversa es necesaria porque los sistemas heredados no suelen tener documentación de alto nivel de sus esquemas ni del diseño global de sus sistemas. Al presentarse el diseño de un nuevo sistema los desarrolladores repasan el diseño de modo que pueda mejorarse en lugar de volver a implementarlo tal como estaba. Se necesita una amplia labor de codificación para dar soporte

a toda la funcionalidad (como pueden ser las interfaces de usuario y los sistemas de información) que proporcionaba el sistema heredado. El proceso completo se denomina **reingeniería**.

Cuando se ha creado y probado el sistema nuevo hay que llenarlo con los datos del sistema heredado y todas las actividades posteriores se deben realizar en el sistema nuevo. No obstante, la transición abrupta al sistema nuevo, que se denomina **enfoque gran explosión**, conlleva varios riesgos. En primer lugar, puede que los usuarios no estén familiarizados con las interfaces del sistema nuevo. En segundo lugar, puede haber fallos o problemas de rendimiento en el sistema nuevo que no se hayan descubierto al probarlo. Estos problemas pueden provocar grandes pérdidas a las empresas, dado que puede resultar gravemente afectada su capacidad para realizar transacciones críticas como las compras y las ventas. En algunos casos extremos se ha llegado a abandonar el sistema nuevo y se ha vuelto a usar el sistema heredado después de que fallara el intento de cambio.

Un enfoque alternativo, denominado **enfoque incremental**, sustituye la funcionalidad del sistema heredado de manera incremental. Por ejemplo, puede que las nuevas interfaces de usuario se utilicen con el sistema antiguo en el dorsal, o viceversa. Otra opción es usar el sistema nuevo sólo para algunas funcionalidades que puedan desgajarse del sistema heredado. En cualquier caso, los sistemas heredados y los nuevos coexisten durante algún tiempo. Por tanto, existe una necesidad de desarrollo y uso de envolturas del sistema heredado para proporcionar la funcionalidad requerida para interoperar con el sistema nuevo. Este enfoque, por tanto, tiene asociado un coste de desarrollo superior.

23.5 Resumen

- El ajuste de los parámetros del sistema de bases de datos, así como el diseño de nivel superior de la base de datos—como pueden ser el esquema, los índices y las transacciones—resultan importantes para lograr un buen rendimiento. La mejor forma de realizar el ajuste es identificar los cuellos de botella y eliminarlos.
- Las pruebas de rendimiento desempeñan un papel importante en las comparaciones entre sistemas de bases de datos, especialmente a medida que cumplen cada vez más las normas. El conjunto de pruebas TPC se usa mucho y las diferentes pruebas TPC resultan útiles para la comparación del rendimiento de las bases de datos con diferentes cargas de trabajo.
- Las normas son importantes debido a la complejidad de los sistemas de bases de datos y a la necesidad de interoperatividad. Hay normas formales para SQL. Las normas de facto, como ODBC y JDBC, y las normas adoptadas por grupos de empresas, como CORBA, han desempeñado un papel importante en el crecimiento de los sistemas de bases de datos cliente–servidor. Hay grupos de empresas desarrollando normas para las bases de datos orientadas a objetos, como ODMG.
- Los sistemas heredados son sistemas basados en tecnologías de generaciones anteriores como las bases de datos no relacionales o, incluso, directamente en sistemas de archivos. Suele ser importante el establecimiento de las interfaces entre los sistemas heredados y los sistemas de última generación cuando ejecutan sistemas de misión crítica. La migración desde los sistemas heredados a los sistemas de última generación debe realizarse con cuidado para evitar interrupciones, que pueden resultar muy costosas.

Términos de repaso

- Ajuste del rendimiento.
- Cuellos de botella.
- Sistemas de colas.
- Parámetros ajustables.
- Ajuste del hardware.
- Regla de los cinco minutos.
- Regla del minuto.
- Ajuste del esquema.
- Ajuste de los índices.
- Vistas materializadas.
- Mantenimiento inmediato de vistas.
- Mantenimiento diferido de vistas.
- Ajuste de las transacciones.
- Mejora de la orientación del conjunto.

- Procesamiento de transacciones por minilotes.
- Simulación del rendimiento.
- Pruebas de rendimiento.
- Tiempo de servicio.
- Tiempo hasta su finalización.
- Clases de aplicaciones de bases de datos.
- Pruebas TPC:
  - ☐ TPC-A.
  - ☐ TPC-B.
  - ☐ TPC-C.
  - ☐ TPC-D.
  - ☐ TPC-R.
  - ☐ TPC-H.
  - ☐ TPC-W.
- Interacciones Web por segundo.
- Pruebas OODB:
  - ☐ OO1.
  - ☐ OO7.
- Normalización.
  - ☐ Normas formales.
  - ☐ Normas de facto.
  - ☐ Normas anticipativas.
  - ☐ Normas reaccionarias.
- Normas de conectividad para bases de datos.
  - ☐ ODBC.
  - ☐ OLE-DB.
  - ☐ Normas X/Open XA.
- Normas de bases de datos de objetos:
  - ☐ ODMG.
  - ☐ CORBA.
- Normas basadas en XML.
- Sistemas heredados.
- Ingeniería inversa.
- Reingeniería.

Ejercicios prácticos

23.1 Muchas aplicaciones necesitan generar los números de secuencia para cada transacción.

a. Si un contador de secuencia se bloquea en dos fases, puede producirse un cuello de botella en la concurrencia. Explíquese por qué puede ocurrir.

b. Muchos sistemas de bases de datos soportan contadores de secuencia predefinidos, que no se bloquean en dos fases; cuando una transacción solicita un número de secuencia, se bloquea el contador, se incrementa y se desbloquea.

I. Explíquese cómo pueden mejorar concurrencia estos contadores.

II. Explíquese por qué puede haber ausencias en los números de secuencia del conjunto final de transacciones comprometidas.

23.2 Supóngase una relación $r(a, b, c)$.

a. Dese un ejemplo de una situación bajo la cual el rendimiento de las consultas de selección con igualdad sobre el atributo a pueda verse seriamente afectada dependiendo de cómo se agrupe r.

b. Supóngase que también se tengan consultas de selección de rangos sobre el atributo b. ¿Se puede agrupar r de forma que las consultas de selección con igualdad sobre $r.a$ y las de rango sobre $r.b$ puedan resolverse eficientemente? Explíquese la respuesta.

c. Si no es posible este agrupamiento, sugiérase cómo se pueden ejecutar eficientemente ambos tipos de preguntas eligiendo índices apropiados, asumiendo que la base de datos soporta sólo planes con índices (es decir, si toda la información requerida para una consulta está disponible en un índice, la base de datos puede generar un plan que utilice el índice pero sin acceder a la relación).

23.3 Supóngase que una aplicación de la base de datos parece no tener cuellos de botella; es decir, la CPU y el uso de disco son altos y prácticamente todas las consultas de la base de datos están equilibradas. ¿Significa esto que la aplicación no se puede ajustar mejor? Explíquese la respuesta.

23.4 Supóngase que un sistema ejecuta tres tipos de transacciones. Las transacciones de tipo A se ejecutan a razón de cincuenta por segundo, las transacciones de tipo B se ejecutan a cien por segundo y las transacciones de tipo C se ejecutan a doscientas por segundo. Supóngase que la mezcla de transacciones tiene un veinticinco por ciento del tipo A, otro veinticinco por ciento del tipo B y un cincuenta por ciento del tipo C.

a. ¿Cuál es la productividad promedio de transacciones del sistema, suponiendo que no hay interferencia entre las transacciones?
b. ¿Qué factores pueden generar interferencias entre las transacciones de los diferentes tipos, haciendo que la productividad calculada sea incorrecta?

23.5 Indíquense algunas ventajas e inconvenientes de las normas anticipativas frente a las normas reaccionarias.

Ejercicios

23.6 Determínese toda la información de rendimiento que proporciona su sistema preferido de bases de datos. Búsquese por lo menos lo siguiente: las consultas que se están ejecutando actualmente o ejecutado recientemente, los recursos que consumió (CPU y E/S) cada una de ellas, la fracción de peticiones que dieron lugar fallos de página en memoria intermedia (para cada consulta, si está disponible) y los bloqueos que tienen un alto grado de la contención. También se puede conseguir la información sobre el uso de la CPU y de E/S del sistema operativo.

23.7 **a.** ¿Cuáles son los tres niveles principales en los que se puede ajustar un sistema de bases de datos para mejorar su rendimiento?
b. Propónganse dos ejemplos del modo en que se puede realizar el ajuste para cada uno de los niveles.

23.8 Al realizar ajuste de rendimiento, ¿se ajustaría primero el hardware (añadiendo discos o memoria) o las transacciones (añadiendo índices o vistas materializadas)? Explíquese la respuesta.

23.9 Supóngase que una aplicación con transacciones por cada acceso y actualización de una única tupla en una relación muy grande almacenada en una organización de archivo B^+. Asúmase que todos los nodos internos del árbol B^+ están en memoria, pero solamente una fracción muy pequeña de las páginas hoja puede caber en memoria. Explíquese cómo calcular el número mínimo de discos requeridos para permitir una carga de trabajo de 1.000 transacciones por segundo. Calcúlese también el número requerido de discos usando los valores de los parámetros de disco del Apartado 11.2.

23.10 ¿Cuál es el motivo para separar una transacción de larga duración en una serie de transacciones más cortas? ¿Qué problemas pueden surgir como consecuencia y cómo pueden evitarse?

23.11 Supóngase que el precio de la memoria cae a la mitad y la velocidad de acceso al disco (número de accesos por segundo) se dobla mientras el resto de los factores permanecen iguales. ¿Cuál sería el efecto de este cambio en las reglas de los cinco minutos y del minuto?

23.12 Indíquense al menos cuatro de las características de las pruebas TPC que ayudan a hacerlas medidas realistas y dignas de confianza.

23.13 ¿Por qué se sustituyó TPC-D por las pruebas de rendimiento TPC-H y TPC-R?

23.14 Determínense las características de la aplicación que ayudarían a decidir TPC-C, TPC-H o TPC-R para modelar mejor la aplicación.

Notas bibliográficas

Una de las primeras propuestas de pruebas de calidad de sistemas de bases de datos (la prueba Wisconsin) la realizaron Bitton et al. [1983]. Las pruebas TPC-A, -B y -C se describen en Gray [1991]. Una versión en línea de todas las descripciones de los pruebas TPC, así como de los resultados de estos pruebas, está disponible en World Wide Web en el URL www.tpc.org; el sitio también contiene información actualizada sobre nuevas propuestas de pruebas. Poess y Floyd [2000] dan una introducción de las pruebas TPC-H, TPC-Ry TPC-W. La prueba OO1 para OODBs se describe en Cattell y Skeen [1992]; la prueba OO7 se describe en Carey et al. [1993].

Kleinrock [1975] y Kleinrock [1976] es un libro de texto popular de dos volúmenes sobre la teoría de colas.

Shasha y Bonnet [2002] ofrecen un tratamiento detallado del ajuste de bases de datos. O'Neil y O'Neil [2000] ofrece un tratamiento de libro de texto de muy buena calidad de la medida del rendimiento y de su ajuste. Las reglas de los cinco minutos y del minuto se describen en Gray y Putzolu [1987] y en Gray y Graefe [1997]. La selección de índices y de vistas materializadas se abordan en Ross et al. [1996], Labio et al. [1997], Gupta [1997], Chaudhuri y Narasayya [1997], Agrawal et al. [2000] y en Mistry et al. [2001]. En Zilio et al. [2004], Dageville et al. [2004] y Agrawal et al. [2004] se describe el ajuste en DB2 de IBM, Oracle y SQL Serverde Microsoft.

Para hallar referencias a las normas de SQL véanse las notas bibliográficas del Capítulo 3.

Se puede hallar información sobre ODBC, OLE-DB, ADO y ADO.NETen el sitio Web www.microsoft.com/data y en varios libros sobre el tema que pueden encontrarse mediante www.amazon.com. La norma ODMG 3.0 se define en La revista *Sigmod Record* de *ACM*, que se publica trimestralmente, tiene una sección fija sobre las normas de bases de datos.

Se halla disponible en línea gran cantidad de información sobre las normas basadas en XML en el sitio Web www.w3c.org. La información sobre RosettaNet se puede encontrar en www.rosettanet.org.

La reingeniería de procesos de negocio se trata en Cook [1996]. Umar [1997] trata la reingeniería y aspectos del trabajo con sistemas heredados.

CAPÍTULO 24

Tipos de datos avanzados y nuevas aplicaciones

Durante la mayor parte de la historia de las bases de datos, sus tipos de datos eran relativamente sencillos, y esto se reflejó en las primeras versiones de SQL. En los últimos años, sin embargo, ha habido una necesidad creciente de manejar tipos de datos nuevos en las bases de datos, como los datos temporales, los datos espaciales y los datos multimedia.

Otra tendencia importante de la última década ha creado sus propios problemas: el auge de la informática móvil, que comenzó con las computadoras portátiles y las agendas de bolsillo, se ha extendido en tiempos más recientes a los teléfonos móviles con capacidad de procesamiento y una gran variedad de computadoras *portátiles*, que se usan cada vez más en aplicaciones comerciales.

En este capítulo se estudian varios tipos nuevos de datos y también se estudian los problemas de las bases de datos relacionados con la informática móvil.

24.1 Motivación

Antes de abordar a fondo cada uno de los temas se resumirá la motivación de cada uno de estos tipos de datos y algunos problemas importantes del trabajo con ellos.

- **Datos temporales**. La mayor parte de los sistemas de bases de datos modelan un estado actual; por ejemplo, los clientes actuales, los estudiantes actuales y los cursos que se están ofertando. En muchas aplicaciones es muy importante almacenar y recuperar la información sobre estados anteriores. La información histórica puede incorporarse de manera manual en el diseño del esquema. No obstante, la tarea se simplifica mucho con el soporte de los datos temporales en las bases de datos, lo cual se estudia en el Apartado 24.2.

- **Datos espaciales**. Entre los datos espaciales están los **datos geográficos**, como los datos y la información asociada a ellos, y los **datos de diseño asistido por computadora**, como los diseños de circuitos integrados o los diseños de edificios. Las aplicaciones de datos espaciales en un principio almacenaban los datos como archivos de un sistema de archivos, igual que las aplicaciones de negocios de las primeras generaciones. Pero, a medida que la complejidad y el volumen de los datos, y el número de usuarios, han aumentado, se han mostrado insuficientes para cubrir las necesidades de muchas aplicaciones que usan datos espaciales los enfoques ad hoc para almacenar y recuperar los datos en un sistema de archivos.

  Las aplicaciones de datos espaciales necesitan los servicios ofrecidos por los sistemas de bases de datos—en especial, la posibilidad de almacenar y consultar de manera eficiente grandes cantidades de datos. Algunas aplicaciones también pueden necesitar otras características de las bases de datos, como las actualizaciones atómicas de parte de los datos almacenados, la durabilidad y el control de concurrencia. En el Apartado 24.3 se estudian las ampliaciones de los sistemas de bases de datos tradicionales necesarias para que soporten los datos espaciales.

- **Datos multimedia**. En el Apartado 24.4 se estudian las características necesarias en los sistemas de bases de datos que almacenan datos multimedia como los datos de imágenes, de vídeo o de audio. La principal característica que diferencia a los datos de vídeo y de audio es que la visualización de éstos exige la recuperación a una velocidad predeterminada constante; por tanto, esos datos se denominan **datos de medios continuos**.
- **Bases de datos móviles**. En el Apartado 24.5 se estudian los requisitos para las bases de datos de la nueva generación de sistemas informáticos portátiles, como las computadoras portátiles y los dispositivos informáticos de bolsillo, que se conectan con las estaciones base mediante redes de comunicación digitales inalámbricas. Estas computadoras necesitan poder operar mientras se hallan desconectadas de la red, a diferencia de los sistemas distribuidos de bases de datos estudiados en el Capítulo 22. También tienen una capacidad de almacenamiento limitada y, por tanto, necesitan técnicas especiales para la administración de la memoria.

24.2 El tiempo en las bases de datos

Las bases de datos modelan el estado de algunos aspectos del mundo real. Generalmente, las bases de datos sólo modelan un estado—el estado actual—del mundo real, y no almacenan información sobre estados anteriores, salvo posiblemente como registro de auditoría. Cuando se modifica el estado del mundo real, se actualiza la base de datos y se pierde la información sobre el estado anterior. Sin embargo, en muchas aplicaciones es importante almacenar y recuperar información sobre estados anteriores. Por ejemplo, una base de datos sobre pacientes debe almacenar información sobre el historial médico de cada paciente. El sistema de control de una fábrica puede almacenar información sobre las lecturas actuales y anteriores de los sensores de la fábrica para su análisis. Las bases de datos que almacenan información sobre los estados del mundo real a lo largo del tiempo se denominan **bases de datos temporales**.

Al considerar el problema del tiempo en los sistemas de bases de datos se distingue entre el tiempo medido por el sistema y el tiempo observado en el mundo real. El **tiempo válido** de un hecho es el conjunto de intervalos de tiempo en el que el hecho es cierto en el mundo real. El **tiempo de transacción** de un hecho es el intervalo de tiempo en el que el hecho es actual en el sistema de bases de datos. Este segundo tiempo se basa en el orden de secuenciación de las transacciones y el sistema lo genera de manera automática. Hay que tener en cuenta que los intervalos de tiempo válidos, que son un concepto del tiempo real, no pueden generarse de manera automática y deben proporcionarse al sistema.

Una **relación temporal** es una relación en la que cada tupla tiene un tiempo asociado en que es verdadera; el tiempo puede ser la hora real o el tiempo asociado a la transacción. Por supuesto, pueden almacenarse ambos, en cuyo caso la relación se denomina **relación bitemporal**. La Figura 24.1 muestra un ejemplo de relación temporal. Para simplificar la representación cada tupla tiene un asociado un único intervalo temporal; así, cada tupla se representa una vez para cada intervalo de tiempo disjunto en que es cierta. Los intervalos se muestran aquí como pares de atributos *de* y *a*; una implementación real tendría un tipo estructurado, acaso denominado *Intervalo*, que contuviera los dos campos. Obsérvese que algunas de la tuplas tienen un asterisco ("*") en la columna temporal *a*; esos asteriscos indican que la tupla es verdadera hasta que se modifique el valor de la columna temporal *a*; así, la tupla es cierta en el momento actual. Aunque los tiempos se muestran en forma textual, se almacenan internamente de una manera más compacta, como el número de segundos desde algún momento de una fecha fija (como las 12:00 A.M., 1 de enero, 1900) que puede volver a traducirse a la forma textual normal.

24.2.1 Especificación del tiempo en SQL

El estándar SQL define los tipos **date**, **time** y **timestamp**. El tipo **date** contiene cuatro cifras para los años (1–9999), dos para los meses (1–12) y dos más para los días (1–31). El tipo **time** contiene dos cifras para la hora, dos para los minutos y otras dos para los segundos, además de cifras decimales opcionales. El campo de los segundos puede superar el valor de sesenta para tener en cuenta los segundos extra que se añaden algunos años para corregir las pequeñas variaciones de velocidad en la rotación de la Tierra. El tipo **timestamp** contiene los campos de **date** y de **time** con seis cifras decimales para el campo de los segundos.

número_cuenta	*nombre_sucursal*	*saldo*	*de*		*a*	
C-101	Centro	500	1/ 1/ 1999	9:00	24/ 1/ 1999	11:30
C-101	Centro	100	24/ 1/ 1999	11:30	*	
C-215	Becerril	700	2/ 6/ 2000	15:30	8/ 8/ 2000	10:00
C-215	Becerril	900	8/ 8/ 2000	10:00	5/ 9/ 2000	8:00
C-215	Becerril	700	5/ 9/ 2000	8:00	*	
C-217	Galapagar	750	5/ 7/ 1999	11:00	1/ 5/ 2000	16:00

Figura 24.1 La relación temporal *cuenta*.

Como las diferentes partes del mundo tienen horas locales diferentes suele darse la necesidad de especificar la zona horaria junto con la hora. La **hora universal coordinada** (Universal Coordinated Time, UTC) es un punto estándar de referencia para la especificación de la hora y las horas locales se definen como diferencias respecto a UTC. (La abreviatura estándar es UTC, en lugar de UCT—Universal Coordinated Time en inglés—ya que es una abreviatura de "tiempo universal coordinado (Universal Coordinated Time)" escrito en francés como *universel temps coordonné*). SQL también soporta dos tipos, **time with time zone** y **timestamp with time zone**, que especifican la hora como la hora local más la diferencia de la hora local respecto de UTC. Por ejemplo, la hora podría expresarse en términos de la hora estándar del Este de EE.UU. (U.S. Eastern Standard Time), con una diferencia de -6:00, ya que la hora estándar del Este de Estados Unidos va retrasada seis horas respecto de UTC.

SQL soporta un tipo denominado **interval**, que permite hacer referencia a un periodo de tiempo como puede ser "1 día" o "2 días y 5 horas", sin especificar la hora concreta en que comienza el periodo. Este concepto se diferencia del concepto de intervalo usado anteriormente, que hace referencia a un intervalo de tiempo con horas de comienzo y de final específicos[1].

24.2.2 Lenguajes de consultas temporales

Las relaciones de base de datos sin información temporal se denominan a veces **relaciones instantáneas**, ya que reflejan el estado del mundo real en una instantánea. Así, una instantánea de una relación temporal en un momento del tiempo t es el conjunto de tuplas de la relación que son ciertas en el momento t, con los atributos de intervalos de tiempo eliminados. La operación instantánea para una relación temporal da la instantánea de la relación en un momento especificado (o en el momento actual, si no se especifica el momento.

Una **selección temporal** es una selección que implica a los atributos de tiempo; una **proyección temporal** es una proyección en la que las tuplas de la proyección heredan los valores de tiempo de las tuplas de la relación original. Una **reunión temporal** es una reunión en que los valores temporales de las tuplas del resultado son la intersección de los valores temporales de las tuplas de las que proceden. Si los valores temporales no se intersecan, esas tuplas se eliminan del resultado.

Los predicados *precede*, *solapa* y *contiene* pueden aplicarse a los intervalos; sus significados deben quedar claros. La operación *intersección* puede aplicarse a dos intervalos, para dar un único intervalo (posiblemente vacío). Sin embargo, la unión de dos intervalos puede que no sea un solo intervalo.

Las dependencias funcionales deben usarse con cuidado en las relaciones temporales. Aunque puede que el número de cuenta determine funcionalmente el saldo en cualquier momento, evidentemente el saldo puede cambiar con el tiempo. Una **dependencia funcional temporal** $X \xrightarrow{\tau} Y$ es válida para un esquema de relación R si, para todos los casos legales r de R, todas las instantáneas de r satisfacen la dependencia funcional $X \rightarrow Y$.

Se han realizado varias propuestas de ampliación de SQL para mejorar su soporte de los datos temporales, pero al menos hasta SQL:2003 no se ha proporcionado ningún soporte especial para los datos temporales aparte de los tipos de datos relativos a la fecha y hora y sus operaciones.

1. Muchos investigadores de bases de datos temporales consideran que este tipo de datos debería haberse denominado **span**, ya que no especifica un momento inicial ni un momento final exactos, sino tan sólo el tiempo transcurrido entre los dos.

24.3 Datos espaciales y geográficos

El soporte de los datos espaciales en las bases de datos es importante para el almacenaje, indexado y consulta eficientes de los datos basados en las posiciones espaciales. Por ejemplo, supóngase que se desea almacenar un conjunto de polígonos en una base de datos y consultar la base de datos para hallar todos los polígonos que intersecan un polígono dado. No se pueden usar las estructuras estándares de índices como los árboles B o los índices asociativos para responder de manera eficiente esas consultas. El procesamiento eficiente de esas consultas necesita estructuras de índices de finalidades especiales, como los árboles R (que se estudiarán posteriormente).

Dos tipos de datos espaciales son especialmente importantes:

- Los **datos de diseño asistido por computadora** (Computer Aided Design, CAD), que incluyen información espacial sobre el modo en que los objetos—como los edificios, los coches o los aviones—están construidos. Otros ejemplos importantes de bases de datos de diseño asistido por computadora son los diseños de circuitos integrados y de dispositivos electrónicos.
- Los **datos geográficos** como los mapas de carreteras, los mapas de uso de la tierra, los mapas topográficos, los mapas políticos que muestran fronteras, los mapas catastrales, etc. Los **sistemas de información geográfica** son bases de datos de propósito especial adaptadas para el almacenamiento de datos geográficos.

El soporte para los datos geográficos se ha añadido a muchos sistemas de bases de datos, como IBM DB2 Spatial Extender, Informix Spatial Datablade u Oracle Spatial.

24.3.1 Representación de la información geométrica

La Figura 24.2 muestra el modo en que se pueden representar de manera normalizada en las bases de datos varias estructuras geométricas. Hay que destacar aquí que la información geométrica puede representarse de varias maneras diferentes, de las que sólo se describen algunas.

Un *segmento rectilíneo* puede representarse mediante las coordinadas de sus extremos. Por ejemplo, en una bases de datos de mapas, las dos coordenadas de un punto serían su latitud y su longitud. Una *línea poligonal* (también denominada *línea quebrada*) consiste en una secuencia conectada de segmentos rectilíneos, y pueden representarse mediante una lista que contenga las coordenadas de los extremos de los segmentos, en secuencia. Se puede representar aproximadamente una curva arbitraria mediante líneas poligonales, dividiendo la curva en una serie de segmentos. Esta representación resulta útil para elementos bidimensionales como las carreteras; en este caso, la anchura de la carretera es lo bastante pequeña en relación con el tamaño de todo el mapa que puede considerarse bidimensional. Algunos sistemas también soportan como primitivas los *arcos de circunferencia*, lo que permite que las curvas se representen como secuencias de arcos.

Los *polígonos* pueden representarse indicando sus vértices en orden, como en la Figura 24.2[2]. La lista de los vértices especifica la frontera de cada región poligonal. En una representación alternativa cada polígono puede dividirse en un conjunto de triángulos, como se muestra en la Figura 24.2. Este proceso se denomina **triangulación**, y se puede triangular cualquier polígono. Se concede un identificador a cada polígono complejo, y cada uno de los triángulos en los que se divide lleva el identificador del polígono. Los círculos y las elipses pueden representarse por los tipos correspondientes, o aproximarse mediante polígonos.

Las representaciones de las líneas poligonales o de los polígonos basadas en listas suelen resultar convenientes para el procesamiento de consultas. Esas representaciones que no están en la primera forma normal se usan cuando están soportadas por la base de datos subyacente. Con objeto de que se puedan usar tuplas de tamaño fijo (en la primera forma normal) para la representación de líneas poligonales se puede dar a la línea poligonal o a la curva un identificador, y se puede representar cada segmento como una tupla separada que también lleva el identificador de la línea poligonal o de la curva. De manera parecida, la representación triangulada de los polígonos permite una representación relacional de los polígonos en su primera forma normal.

2. Algunas referencias usan el término *polígono cerrado* para hacer referencia a lo que aquí se denominan polígonos y se refieren a las líneas poligonales abiertas como polígonos abiertos.

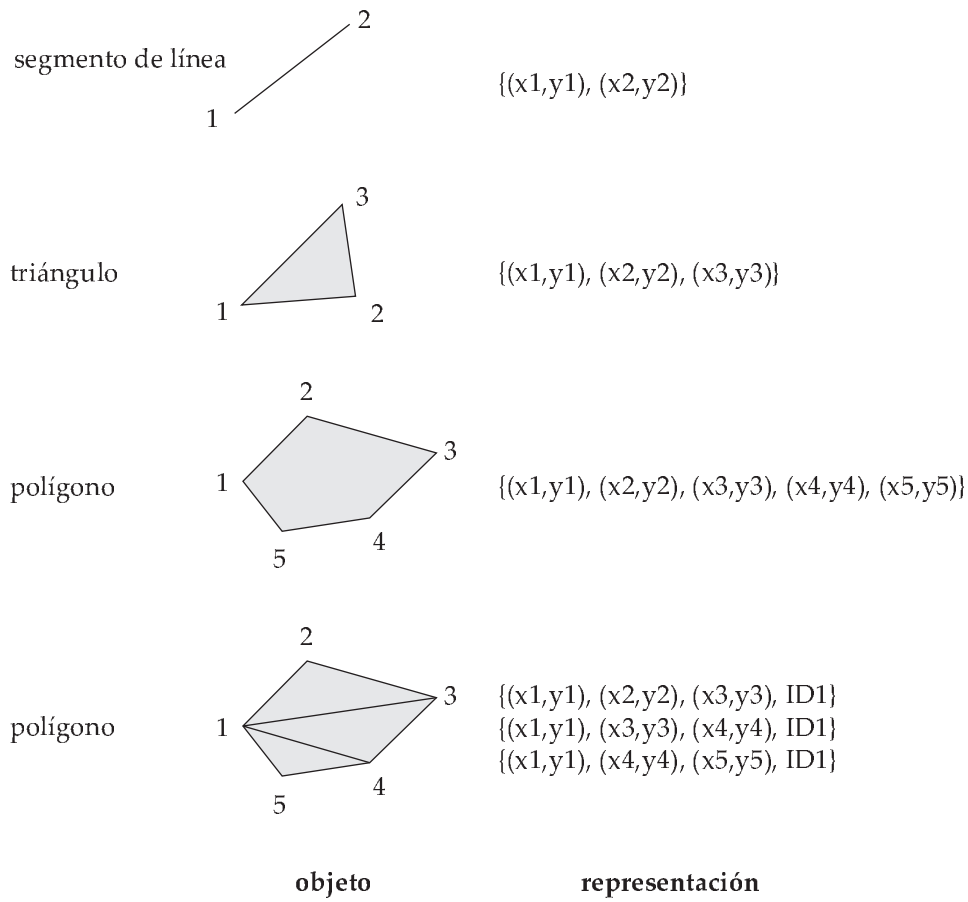


Figura 24.2 Representación de estructuras geométricas.

La representación de los puntos y de los segmentos rectilíneos en el espacio tridimensional es parecida a su representación en el espacio bidimensional, siendo la única diferencia que los puntos tienen un componente z adicional. De manera parecida, la representación de las figuras planas—como los triángulos, los rectángulos y otros polígonos—no cambia mucho cuando se consideran tres dimensiones. Los tetraedros y los paralelepípedos pueden representarse de la misma manera que los triángulos y los rectángulos. Es posible representar poliedros arbitrarios dividiéndolos en tetraedros, igual que se triangulan los polígonos. También se pueden representar indicando las caras, cada una de las cuales es, en sí misma, un polígono, junto con una indicación del lado de la cara que está por dentro del poliedro.

24.3.2 Bases de datos para diseño

Los **sistemas de diseño asistido por computadora** (Computer Aided Design, CAD) tradicionalmente almacenaban los datos en la memoria durante su edición u otro tipo de procesamiento y los volvían a escribir en archivos al final de la sesión de edición. Entre los inconvenientes de este esquema están el coste (la complejidad de programación y el coste temporal) de transformar los datos de una forma a otra, y la necesidad de leer todo un archivo aunque sólo sea necesaria una parte. Para los diseños de gran tamaño, como el diseño de circuitos integrados a gran escala o el diseño de todo un avión, puede que resulte imposible guardar en la memoria el diseño completo. Los diseñadores de bases de datos orientadas a objetos estaban motivados en gran parte por las necesidades de los sistemas de CAD. Las bases de datos orientadas a objetos representan los componentes del diseño como objetos y las conexiones entre los objetos indican el modo en que está estructurado el diseño.

Los objetos almacenados en las bases de datos de diseño suelen ser objetos geométricos. Entre los objetos geométricos bidimensionales sencillos se encuentran los puntos, las líneas, los triángulos, los

rectángulos y los polígonos en general. Los objetos bidimensionales complejos pueden formarse a partir de objetos sencillos mediante las operaciones de unión, intersección y diferencia. De manera parecida, los objetos tridimensionales complejos pueden formarse a partir de objetos más sencillos como las esferas, los cilindros y los paralelepípedos mediante las operaciones unión, intersección y diferencia, como en la Figura 24.3. Las superficies tridimensionales también pueden representarse mediante **modelos de alambres**, que esencialmente modelan las superficies como conjuntos de objetos más sencillos, como segmentos rectilíneos, triángulos y rectángulos.

Las bases de datos de diseño también almacenan información no espacial sobre los objetos, como el material del que están construidos. Esa información se suele poder modelar mediante técnicas estándar de modelado de datos. Aquí se centrará la atención únicamente en los aspectos espaciales.

Al diseñar hay que realizar varias operaciones espaciales. Por ejemplo, puede que el diseñador desee recuperar la parte del diseño que corresponde a una región de interés determinada. Las estructuras espaciales de índices, estudiadas en el Apartado 24.3.5, resultan útiles para estas tareas. Las estructuras espaciales de índices son multidimensionales, trabajan con datos de dos y de tres dimensiones, en vez de trabajar solamente con la sencilla ordenación unidimensional que proporcionan los árboles B^+.

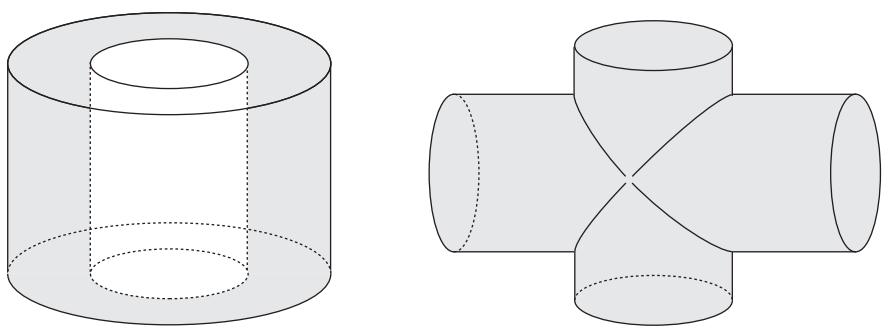
Las restricciones de integridad espacial, como "dos tuberías no deben estar en la misma ubicación", son importantes en las bases de datos de diseño para evitar errores por interferencias. Estos errores suelen producirse si el diseño se realiza a mano y sólo se detectan al construir un prototipo. En consecuencia, estos errores pueden resultar costosos de reparar. El soporte de las bases de datos para las restricciones de integridad espacial ayuda a los usuarios a evitar los errores de diseño, con lo que hacen que el diseño sea consistente. La implementación de esas verificaciones de integridad depende una vez más de la disponibilidad de estructuras multidimensionales de índices eficientes.

24.3.3 Datos geográficos

Los datos geográficos son de naturaleza espacial, pero se diferencian de los datos de diseño en ciertos aspectos. Los mapas y las imágenes de satélite son ejemplos típicos de datos geográficos. Los mapas pueden proporcionar no sólo información sobre la ubicación—sobre fronteras, ríos y carreteras, por ejemplo—sino también información mucho más detallada asociada con la ubicación, como la elevación, el tipo de suelo, el uso de la tierra y la cantidad anual de lluvia.

Los **datos geográficos** pueden clasificarse en dos tipos:

- **Datos por líneas**. Estos datos consisten en mapas de bits o en mapas de píxeles en dos o más dimensiones. Un ejemplo típico de imagen de líneas bidimensional son las imágenes de satélite de cobertura nubosa, en las que cada píxel almacena la visibilidad de nubes en un área concreta. Estos datos pueden ser tridimensionales—por ejemplo, la temperatura a diferentes altitudes en distintas regiones, también medidas con la ayuda de un satélite. El tiempo puede formar otra dimensión—por ejemplo, las medidas de la temperatura superficial en diferentes momentos. Las bases de datos de diseño no suelen almacenar datos por líneas.



(a) Diferencia de cilindros (b) Unión de cilindros

Figura 24.3 Objetos tridimensionales complejos.

- **Datos vectoriales**. Los datos vectoriales están formados a partir de objetos geométricos básicos como los puntos, los segmentos rectilíneos, los triángulos y otros polígonos en dos dimensiones y los cilindros, las esferas, los paralelepípedos y otros poliedros en tres dimensiones.

  Los datos cartográficos suelen representarse en formato vectorial. Los ríos y las carreteras pueden representarse como uniones de varios segmentos rectilíneos. Los estados y los países pueden representarse como polígonos. La información topológica como la altura puede representarse mediante una superficie dividida en polígonos que cubren las regiones de igual altura, con un valor de altura asociado a cada polígono.

24.3.3.1 Representación de los datos geográficos

Los accidentes geográficos, como los estados y los grandes lagos, se representan como polígonos complejos. Algunos accidentes, como los ríos, pueden representarse como curvas complejas o como polígonos complejos, en función de si su anchura es importante o no.

La información geográfica relativa a las regiones, como la cantidad anual de lluvia, puede representarse como un array—es decir, en forma de líneas. Para reducir el espacio necesario, el array se puede almacenar de manera comprimida. En el Apartado 24.3.5 se estudia una representación alternativa de estas arrays mediante una estructura de datos denominada *árbol cuadrático*.

Como se indicó en el Apartado 24.3.3, se puede representar la información regional en forma vectorial usando polígonos, cada uno de los cuales es una región en la que el valor del array es el mismo. La representación vectorial es más compacta que la de líneas en algunas aplicaciones. También es más precisa para algunas tareas, como el dibujo de carreteras, en que la división de la región en píxeles (que pueden ser bastante grandes) lleva a una pérdida de precisión en la información de ubicación. No obstante, la representación vectorial resulta inadecuada para las aplicaciones en que los datos se basan en líneas de manera intrínseca, como las imágenes de satélite.

24.3.3.2 Aplicaciones de los datos geográficos

Las bases de datos geográficas tienen gran variedad de usos, incluidos los servicios de mapas en línea, los sistemas de navegación para vehículos, la información de redes de distribución para las empresas de servicios públicos como son los sistemas de telefonía, electricidad y suministro de agua y la información de uso de la tierra para ecologistas y planificadores.

Los servicios de mapas de carreteras basados en Web constituyen una aplicación muy usada de datos cartográficos. En su nivel más sencillo estos sistemas pueden usarse para generar mapas de carreteras en línea de la región deseada. Una ventaja importante de los mapas interactivos es que resulta sencillo dimensionar los mapas al tamaño deseado—es decir, acercarse y alejarse para ubicar los accidentes importantes. Los servicios de mapas de carretera también almacenan información sobre carreteras y servicios, como el trazado de las carreteras, los límites de velocidad, las condiciones de las vías, las conexiones entre carreteras y los tramos de sentido único. Con esta información adicional sobre las carreteras se pueden usar los mapas para obtener indicaciones para desplazarse de un sitio a otro y para localizar, por ejemplo, hoteles, gasolineras o restaurantes con las ofertas y gamas de precios deseadas.

Los sistemas de navegación para los vehículos son sistemas montados en automóviles que proporcionan mapas de carreteras y servicios para la planificación de los viajes. Un añadido útil a los sistemas de información geográfica móviles como los sistemas de navegación de los vehículos es una unidad del **sistema de posicionamiento global** (Global Positioning System, GPS), que usa la información emitida por los satélites GPS para hallar la ubicación actual con una precisión de decenas de metros. Con estos sistemas el conductor no puede perderse nunca[3]—la unidad GPS halla la ubicación en términos de latitud, longitud y elevación y el sistema de navegación puede consultar la base de datos geográfica para hallar el lugar en que se encuentra y la carretera en que se halla el vehículo.

Las bases de datos geográficas para información de utilidad pública se están volviendo cada vez más importantes a medida que crece la red de cables y tuberías enterrados. Sin mapas detallados las obras realizadas por una empresa de servicio público pueden dañar los cables de otra, lo que daría lugar a una

3. Bueno, ¡casi nunca!

interrupción del servicio a gran escala. Las bases de datos geográficas, junto con los sistemas precisos de determinación de la posición, pueden ayudar a evitar estos problemas.

Hasta ahora se ha explicado el motivo de que las bases de datos espaciales resulten útiles. En el resto del apartado se estudiarán los detalles técnicos, como la representación y el indexado de la información espacial.

24.3.4 Consultas espaciales

Existen varios tipos de consultas con referencia a ubicaciones espaciales.

- Las **consultas de proximidad** solicitan objetos que se hallen cerca de una ubicación especificada. La consulta para hallar todos los restaurantes que se hallan a menos de una distancia dada de un determinado punto es un ejemplo de consulta de proximidad. La **consulta de vecino más próximo** solicita el objeto que se halla más próximo al punto especificado. Por ejemplo, puede que se desee hallar la gasolinera más cercana. Obsérvese que esta consulta no tiene que especificar un límite para la distancia y, por tanto, se puede formular aunque no se tenga idea de la distancia a la que se halla la gasolinera más próxima.
- Las **consultas regionales** tratan de regiones espaciales. Estas consultas pueden preguntar por objetos que se hallen parcial o totalmente en el interior de la región especificada. Un ejemplo es la consulta para hallar todas las tiendas minoristas dentro de los límites geográficos de una ciudad dada.
- Puede que las consultas también soliciten **intersecciones** y **uniones** de regiones. Por ejemplo, dada la información regional, como pueden ser la lluvia anual y la densidad de población, una consulta puede solicitar todas las regiones con una baja cantidad de lluvia anual y una elevada densidad de población.

Las consultas que calculan las intersecciones de regiones pueden considerarse como si calcularan la **reunión espacial** de dos relaciones espaciales—por ejemplo, una que represente la cantidad de lluvia y otra que represente la densidad de población—con la ubicación en el papel de atributo de reunión. En general, dadas dos relaciones, cada una de las cuales contiene objetos espaciales, la reunión espacial de las dos relaciones genera, o bien pares de objetos que se intersectan o bien las regiones de intersección de esos pares.

Existen diferentes algoritmos de reunión que calculan eficazmente las reuniones espaciales de datos vectoriales. Aunque se pueden usar las reuniones de bucles anidados o de bucles anidados indexados (con índices espaciales), las reuniones de asociación y las reuniones por mezcla–ordenación no pueden usarse con datos espaciales. Los investigadores han propuesto técnicas de reunión basadas en el recorrido coordinado de las estructuras espaciales de los índices de las dos relaciones. Véanse las notas bibliográficas para obtener más información.

En general, las consultas de datos espaciales pueden tener una combinación de requisitos espaciales y no espaciales. Por ejemplo, puede que se desee averiguar el restaurante más cercano que tenga menú vegetariano y que cueste menos de diez euros por comida.

Dado que los datos espaciales son inherentemente gráficos, se suelen consultar mediante un lenguaje gráfico de consulta. El resultado de esas consultas también se muestra gráficamente, en vez de mostrarse en tablas. El usuario puede realizar varias operaciones con la interfaz, como escoger el área que desea ver (por ejemplo, apuntando y pulsando en los barrios del oeste de Arganzuela), acercarse y alejarse, escoger lo que desea mostrar de acuerdo con las condiciones de selección (por ejemplo, casas con más de tres habitaciones), superponer varios mapas (por ejemplo, las casas con más de tres habitaciones superpuestas sobre un mapa que muestre las zonas con bajas tasas de delincuencia), etc. La interfaz gráfica constituye la parte visible para el usuario. Se han propuesto extensiones de SQL para permitir que las bases de datos relacionales almacenen y recuperen información espacial de manera eficiente y también permitir que las consultas mezclen las condiciones espaciales con las no espaciales. Las extensiones incluyen la autorización de tipos de datos abstractos como las líneas, los polígonos y los mapas de bits y la autorización de condiciones espaciales como *contiene* o *solapa*.

24.3.5 Índices sobre los datos espaciales

Los índices son necesarios para el acceso eficiente a los datos espaciales. Las estructuras de índices tradicionales, como los índices de asociación y los árboles B, no resultan adecuadas, ya que únicamente trabajan con datos unidimensionales, mientras que los datos espaciales suelen ser de dos o más dimensiones.

24.3.5.1 Los árboles k-d

Para comprender el modo de indexar los datos espaciales que constan de dos o más dimensiones se considera en primer lugar el indexado de los puntos de los datos unidimensionales. Las estructuras arbóreas, como los árboles binarios y los árboles B, operan dividiendo el espacio en partes más pequeñas de manera sucesiva. Por ejemplo, cada nodo interno de un árbol binario divide un intervalo unidimensional en dos. Los puntos que quedan en la partición izquierda van al subárbol izquierdo; los puntos que quedan en la partición de la derecha van al subárbol derecho. En los árboles binarios equilibrados la partición se escoge de modo que, aproximadamente, la mitad de los puntos almacenados en el subárbol caigan en cada partición. De manera parecida, cada nivel de un árbol B divide un intervalo unidimensional en varias partes.

Se puede usar esa intuición para crear estructuras arbóreas para el espacio bidimensional, así como para espacios de más dimensiones. Una estructura arbórea denominada **árbol k-d** fue una de las primeras estructuras usadas para la indexación en varias dimensiones. Cada nivel de un árbol k-d divide el espacio en dos. La división se realiza según una dimensión en el nodo del nivel superior del árbol, según otra dimensión en los nodos del nivel siguiente, etc., alternando cíclicamente las dimensiones. La división se realiza de tal modo que, en cada nodo, aproximadamente la mitad de los puntos almacenados en el subárbol cae a un lado y la otra mitad al otro. La división se detiene cuando un nodo tiene menos puntos que un valor máximo dado. La Figura 24.4 muestra un conjunto de puntos en el espacio bidimensional y una representación en árbol k-d de ese conjunto de puntos. Cada línea corresponde a un nodo del árbol, y el número máximo de puntos en cada nodo hoja se ha definido como uno. Cada línea de la figura (aparte del marco exterior) corresponde a un nodo del árbol k-d. La numeración de las líneas en la figura indica el nivel del árbol en el que aparece el nodo correspondiente.

El **árbol k-d B** extiende el árbol k-d para permitir varios nodos hijo por cada nodo interno, igual que los árboles B extienden los árboles binarios, para reducir la altura del árbol, los árboles k-dB están mejor adaptados al almacenamiento secundario que los árboles k-d.

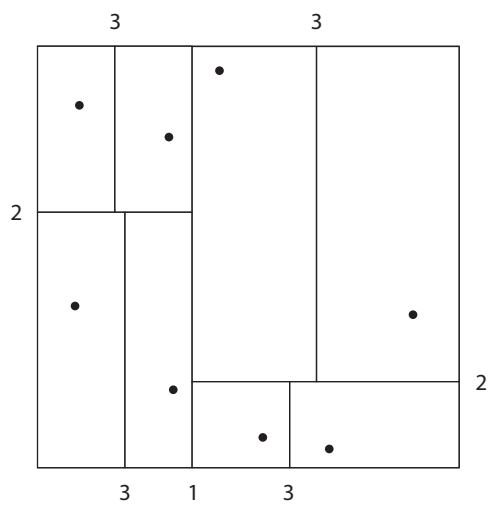


Figura 24.4 División del espacio por un árbol k-d.

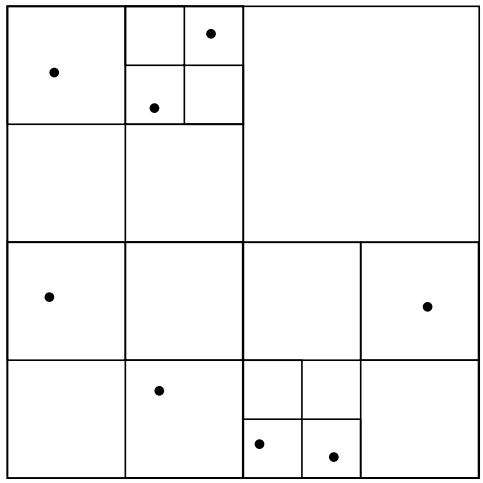


Figura 24.5 División del espacio por un árbol cuadrático.

24.3.5.2 Árboles cuadráticos

Una representación alternativa de los datos bidimensionales son los **árboles cuadráticos**. En la Figura 24.5 aparece un ejemplo de la división del espacio mediante un árbol cuadrático. El conjunto de puntos es el mismo que en la Figura 24.4. Cada nodo de un árbol cuadrático está asociado con una región rectangular del espacio. El nodo superior está asociado con todo el espacio objetivo. Cada nodo que no sea un nodo hoja del árbol cuadrático divide su región en cuatro cuadrantes del mismo tamaño y, a su vez, cada uno de esos nodos tiene cuatro nodos hijo correspondientes a los cuatro cuadrantes. Los nodos hoja tienen un número de puntos que varía entre cero y un número máximo fijado. A su vez, si la región correspondiente a un nodo tiene más puntos que el máximo fijado, se crean nodos hijo para ese nodo. En el ejemplo de la Figura 24.5, el número máximo de puntos de cada nodo hoja está fijado en uno.

Este tipo de árbol cuadrático se denomina **árbol cuadrático PR**, para indicar que almacena los puntos y que la división del espacio se basa en regiones, en vez de en el conjunto real de puntos almacenados. Se pueden usar **árboles cuadráticos regionales** para almacenar información de arrays (de líneas). Cada nodo de los árboles cuadráticos regionales es un nodo hoja si todos los valores del array de la región que abarca son iguales. En caso contrario, se vuelve a subdividir en cuatro nodos hijo con la misma área y es, por tanto, un nodo interno. Cada nodo del árbol cuadrático regional corresponde a un subarray de valores. Los subarrays correspondientes a las hojas contienen un solo elemento del array o varios, todos ellos con el mismo valor.

El indexado de los segmentos rectilíneos y de los polígonos presenta problemas nuevos. Hay extensiones de los árboles k-d y de los árboles cuadráticos para esta labor. No obstante, un segmento rectilíneo o un polígono puede cruzar una línea divisoria. Si lo hace, hay que dividirlo y representarlo en cada uno de los subárboles en que aparezcan sus fragmentos. La aparición múltiple de un segmento lineal o de un polígono puede dar lugar a ineficiencias en el almacenamiento, así como a ineficiencias en las consultas.

24.3.5.3 Árboles R

La estructura de almacenamiento denominada **árbol R** resulta útil para el indexado de objetos como puntos, segmentos de línea, rectángulos y otros polígonos. Un árbol R es una estructura arbórea equilibrada con los objetos indexados almacenados en los nodos hoja, de manera parecida a los árboles B^+. No obstante, en lugar de un rango de valores, se asocia una **caja límite** con cada nodo del árbol. La caja límite de un nodo hoja es el rectángulo mínimo paralelo a los ejes que contiene todos los objetos almacenados en el nodo hoja. La caja límite de los nodos internos, de manera parecida, es el rectángulo mínimo paralelo a los ejes que contiene las cajas límite de sus nodos hijo. La caja límite de un objeto (como un polígono) viene definida, de manera parecida, como el rectángulo mínimo paralelo a los ejes que contiene al objeto.

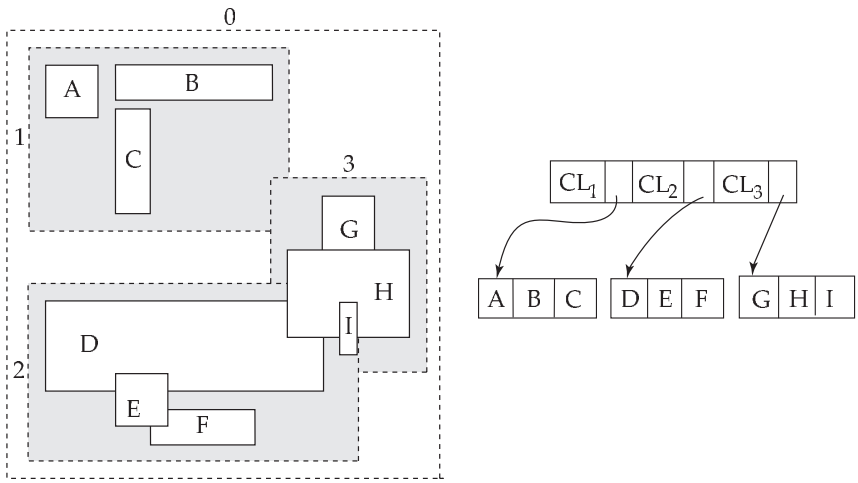


Figura 24.6 Árbol R.

Cada nodo interno almacena las cajas límite de los nodos hijo junto con los punteros para los nodos hijo. Cada nodo hijo almacena los objetos indexados y puede que, opcionalmente, almacene las cajas límite de esos objetos; las cajas límite ayudan a acelerar las comprobaciones de solapamientos de los rectángulos con los objetos indexados—si el rectángulo de una consulta no se solapa con la caja límite de un objeto, o se puede solapar tampoco el objeto (si los objetos indexados son rectángulos, por supuesto no hace falta almacenar las cajas límite, ya que son idénticas a los rectángulos).

La Figura 24.6 muestra el ejemplo de un conjunto de rectángulos (dibujados con línea continua) y de sus cajas límite (dibujadas con línea discontinua) de los nodos de un árbol R para el conjunto de rectángulos. Hay que tener en cuenta que las cajas límite se muestran con espacio adicional en su interior, para hacerlas visibles en el dibujo. En realidad, las cajas son menores y se ajustan fielmente a los objetos que contienen; es decir, cada cara de una caja límite C toca como mínimo uno de los objetos o de las cajas límite que se contienen en C.

El árbol R en sí se halla a la derecha de la Figura 24.6. La figura hace referencia a las coordenadas de la caja límite i como CL_i.

Ahora se verá el modo de implementar las operaciones de búsqueda, inserción y eliminación en los árboles R.

- **Búsqueda**. Como muestra la figura, las cajas límite con nodos hermanos pueden solaparse; en los árboles B^+, los árboles k-d y los árboles cuadráticos, sin embargo, los rangos no se solapan. La búsqueda de objetos que contengan un punto, por tanto, tiene que seguir *todos* los nodos hijo cuyas cajas limites asociadas contengan el punto; en consecuencia, puede que haya que buscar por varios caminos. De manera parecida, una consulta para buscar todos los objetos que intersecten con uno dado tiene que bajar por todos los nodos en que el rectángulo asociado intersecte al objeto dado.

- **Inserción**. Cuando se inserta un objeto en un árbol R se selecciona un nodo hoja para que lo guarde. Lo ideal sería seleccionar un nodo hoja que tuviera espacio para guardar una nueva entrada, y cuya caja límite contuviera a la caja límite del objeto. Sin embargo, puede que ese nodo no exista; aunque exista, hallarlo puede resultar muy costoso, ya que no es posible hallarlo mediante un solo recorrido desde la raíz. En cada nodo interno se pueden encontrar varios nodos hijo cuyas cajas límite contengan la caja límite del objeto, y hay que explorar cada uno de esos nodos hijo. Por tanto, como norma heurística, en un recorrido desde la raíz, si alguno de los nodos hijo tiene una caja límite que contenga a la caja límite del objeto, el algoritmo del árbol R escoge una de ellas de manera arbitraria. Si ninguno de los nodos hijo satisface esta condición, el algoritmo escoge un nodo hijo cuya caja límite tenga el solapamiento máximo con la caja límite del objeto para continuar el recorrido.

Una vez que se ha llegado al nodo hoja, si el nodo ya está lleno, el algoritmo lleva a cabo una división del nodo (y propaga hacia arriba la división si es necesario) de manera muy parecida a la inserción de los árboles B^+. Igual que con la inserción en los árboles B^+, el algoritmo de inserción de los árboles R asegura que el árbol siga equilibrado. Además, asegura que las cajas límite de los nodos hoja, así como los nodos internos, sigan siendo consistentes; es decir, las cajas límite de los nodos hoja contienen todas las cajas limites de los polígonos almacenados en el nodo hoja, mientras que las cajas límite de los nodos internos contienen todas las cajas límite de los nodos hijo.

La principal diferencia del procedimiento de inserción con la inserción en los árboles B^+ radica en el modo en que se dividen los nodos. En los árboles B^+ es posible hallar un valor tal que la mitad de los de las entradas sea menor que el punto medio y la mitad sea mayor que el valor. Esta propiedad no se generaliza más allá de una dimensión; es decir, para más de una dimensión, no siempre es posible dividir las entradas en dos conjuntos tales que sus cajas límite no se solapen. En lugar de eso, como norma heurística, el conjunto de entradas S puede dividirse en dos conjuntos disjuntos S_1 y S_2 tales que las cajas límite de S_1 y S_2 tengan un área total mínima; otra norma heurística sería dividir las entradas en dos conjuntos S_1 y S_2 de modo que S_1 y S_2 tengan un solapamiento mínimo. Los dos nodos resultantes de la división contendrían las entradas de S_1 y S_2, respectivamente. El coste de hallar las divisiones con área total o solapamiento mínimos puede ser elevado, por lo que se usan normas heurísticas más económicas, como la heurística de la *división cuadrática* (la heurística recibe el nombre del hecho de que toma el cuadrado del tiempo en el número de entradas).

La principal diferencia del procedimiento de inserción con la inserción en los árboles B^+ radica en el modo en que se dividen los nodos. En los árboles B^+ es posible hallar un valor tal que la mitad de los de las entradas sea menor que el punto medio y la mitad sea mayor que el valor. Esta propiedad no se generaliza más allá de una dimensión; es decir, para más de una dimensión, no siempre es posible dividir las entradas en dos conjuntos tales que sus cajas límite no se solapen. En lugar de eso, como norma heurística, el conjunto de entradas S puede dividirse en dos conjuntos disjuntos S_1 y S_2 tales que las cajas límite de S_1 y S_2 tengan un área total mínima; otra norma heurística sería dividir las entradas en dos conjuntos S_1 y S_2 de modo que S_1 y S_2 tengan un solapamiento mínimo. Los dos nodos resultantes de la división contendrían las entradas de S_1 y S_2, respectivamente. El coste de hallar las divisiones con área total o solapamiento mínimos puede ser elevado, por lo que se usan normas heurísticas más económicas, como la heurística de la *división cuadrática* (la heurística recibe el nombre del hecho de que toma el cuadrado del tiempo en el número de entradas).

La heurística de la **división cuadrática** funciona de esta manera: en primer lugar, selecciona un par de entradas a y b de S tales que al ponerlas en el mismo nodo den lugar a una caja límite con el máximo de espacio desaprovechado; es decir, el área de la caja límite mínima de a y b menos la suma de las áreas de a y b es máxima. La heurística sitúa las entradas a y b en los conjuntos S_1 y S_2, respectivamente.

Luego añade iterativamente las entradas restantes, una por cada iteración, a uno de los dos conjuntos S_1 o S_2. En cada iteración, para cada entrada restante e, $i_{e,1}$ denota el incremento en el tamaño de la caja límite de S_1 si e se añade a S_1 e $i_{e,2}$ denota el incremento correspondiente de S_2. En cada iteración la heurística escoge una de las entradas con la máxima diferencia entre $i_{e,1}$ e $i_{e,2}$ y la añade a S_1 si $i_{e,1}$ es menor que $i_{e,2}$, y a S_2 en caso contrario. Es decir, se escoge en cada iteración una entrada con "preferencia máxima" por S_1 o S_2. Las iteraciones se detienen cuando se han asignado todas las entradas o cuando uno de los conjuntos S_1 o S_2 tiene entradas suficientes como para que todas las demás entradas haya que añadirlas al otro conjunto de modo que los nodos creados a partir de S_1 y S_2 tengan la ocupación mínima exigida. La heurística añade luego todas las entradas no asignadas al conjunto con menos entradas.

- **Eliminación**. La eliminación puede llevarse a cabo como si fuera una eliminación de árbol B^+, tomando prestadas las entradas de los nodos hermanos, o mezclando nodos hermanos si un nodo se queda menos lleno de lo exigido. Un enfoque alternativo redistribuye todas las entradas

de los nodos menos llenos de lo necesario a los nodos hermanos, con el objetivo de mejorar el agrupamiento de las entradas en el árbol R.

Véanse las referencias bibliográficas para obtener más detalles de las operaciones de inserción y de eliminación en los árboles R, así como de las variantes de los árboles R, denominados árboles R^* o árboles R^+.

La eficiencia del almacenamiento de los árboles R es mayor que la de los árboles k-d y que la de los árboles cuadráticos, ya que cada polígono sólo se almacena una vez, y se puede asegurar que cada nodo está, como mínimo, medio lleno. No obstante, las consultas pueden resultar más lentas, ya que hay que buscar por varios caminos. Las reuniones espaciales son más sencillas con los árboles cuadráticos que con los árboles R, ya que todos los árboles cuadráticos de cada región están divididos de la misma manera. Sin embargo, debido a su mayor eficiencia de almacenamiento, y a su parecido con los árboles B, los árboles R y sus variantes se han hecho populares en los sistemas de bases de datos que soportan datos espaciales.

24.4 Bases de datos multimedia

Los datos multimedia, como las imágenes, el sonido y el vídeo—una modalidad de datos cada vez más popular—se almacenan hoy en día casi siempre fuera de las bases de datos, en sistemas de archivos. Este tipo de almacenamiento no supone ningún problema cuando el número de objetos multimedia es relativamente pequeño, ya que las características proporcionadas por las bases de datos no suelen ser importantes.

Sin embargo, las características de las bases de datos se vuelven importantes cuando el número de objetos multimedia almacenados es grande. Aspectos como las actualizaciones transaccionales, las facilidades de consulta y el indexado se vuelven importantes. Los objetos multimedia suelen tener atributos descriptivos, como los que indican su fecha de creación, su creador y la categoría a la que pertenecen. Un enfoque de la creación de una base de datos para esos objetos multimedia es usar las bases de datos para almacenar los atributos descriptivos y realizar un seguimiento de los archivos en los que se almacenan los objetos multimedia.

Sin embargo, el almacenamiento de los objetos multimedia fuera de la base de datos hace más difícil proporcionar la funcionalidad de la base de datos, como el indexado con base en el contenido real de datos multimedia. También puede llevar a inconsistencias, como que un archivo esté registrado en la base de datos pero que sus contenidos falten, o viceversa. Por tanto, resulta deseable almacenar los propios datos en la base de datos.

Hay que abordar varios aspectos si se pretende almacenar los datos multimedia en una base de datos.

- La base de datos debe soportar objetos de gran tamaño, ya que los datos multimedia como los vídeos pueden ocupar varios gigabytes de espacio de almacenamiento. Los objetos de mayor tamaño pueden dividirse en fragmentos menores y almacenarse en la base de datos. De manera alternativa, los objetos multimedia pueden almacenarse en un sistema de archivos, pero la base de datos puede contener un puntero hacia el objeto; el puntero suele ser un nombre de archivo. El estándar SQL/MED (MED significa Management of External Data, gestión de datos externos) permite que los datos externos, como los archivos, se traten como si formaran parte de la base de datos. Con SQL/MED parece que los objetos son parte de la base de datos, pero pueden almacenarse externamente.

  Los formatos de los datos multimedia se estudian en el Apartado 24.4.1.

- La recuperación de algunos tipos de datos, como los de sonido y los de vídeo, tiene la exigencia de que la entrega de los datos debe realizarse a una velocidad constante garantizada. Estos datos se denominan a veces **datos isócronos** o **datos de medios continuos**. Por ejemplo, si los datos de sonido no se proporcionan a tiempo, habrá saltos en el sonido. Si los datos se proporcionan demasiado deprisa, se pueden desbordar las memorias intermedias, lo que dará lugar a la pérdida de datos. Los datos de medios continuos se estudian en el Apartado 24.4.2.

- La recuperación basada en la semejanza es necesaria en muchas aplicaciones de bases de datos multimedia. Por ejemplo, en una base de datos que almacene imágenes de huellas digitales, se proporciona una consulta de la imagen de una huella dactilar y hay que recuperar las huellas dactilares de la base de datos que sean parecidas a la huella dactilar de la consulta. Las estructuras de índices como los árboles B^+ y los árboles R no se pueden usar para esta finalidad; hay que crear estructuras especiales de índices. La recuperación basada en el parecido se estudia en el Apartado 24.4.3

24.4.1 Formatos de datos multimedia

Debido al gran número de bytes necesarios para representar los datos multimedia es fundamental que se almacenen y transmitan de manera comprimida. Para los datos de imágenes el formato más usado es *JPEG*, que recibe su nombre del organismo de normalización que lo creó: el grupo conjunto de expertos en imágenes (*Joint Picture Experts Group*). Se pueden almacenar los datos de vídeo codificando cada fotograma de vídeo en formato JPEG, pero esa codificación supone un desperdicio, ya que los fotogramas de vídeo sucesivos suelen ser casi iguales. El grupo de expertos en películas (*Moving Picture Experts Group*) desarrolló la serie de estándares *MPEG* para codificar los datos de vídeo y de sonido; estas codificaciones aprovechan las similitudes de las secuencias de fotogramas para conseguir un grado de compresión mayor. El estándar *MPEG-1* almacena un minuto de vídeo y sonido a treinta fotogramas por segundo en unos 12.5 megabytes (en comparación con los aproximadamente 75 megabytes sólo para vídeo en JPEG). No obstante, la codificación MPEG-1 introduce alguna pérdida de calidad del vídeo, a un nivel aproximadamente equivalente al de las cintas de vídeo VHS. El estándar *MPEG-2* se diseñó para los sistemas de radiodifusión digitales y para los discos de vídeo digitales (DVD); sólo introduce una pérdida de calidad de vídeo despreciable. MPEG-2 comprime un minuto de vídeo y de sonido en aproximadamente 17 megabytes. *MPEG-4* proporciona técnicas de mayor compresión de vídeo con ancho de banda variable para proporcionar la difusión de vídeo en redes con un amplio rango de anchos de banda. Se usan varios estándares competidores para la codificación de sonido, entre ellos *MP3* (que significa MPEG-1 Capa 3), RealAudio y otros formatos.

24.4.2 Datos de medios continuos

Los tipos más importantes de datos de medios continuos son los datos de vídeo y los de sonido (por ejemplo, una base de datos de películas). Los sistemas de medios continuos se caracterizan por sus requisitos de entrega de información en tiempo real:

- Los datos deben entregarse lo bastante rápido como para que no haya saltos en el resultado de sonido o de vídeo.
- Los datos deben entregarse a una velocidad que no cause un desbordamiento de las memorias intermedias del sistema.
- La sincronización entre los distintos flujos de datos debe conservarse. Esta necesidad surge, por ejemplo, cuando el vídeo de una persona que habla muestra sus labios moviéndose de manera sincronizada con el sonido de su voz.

Para proporcionar los datos de manera predecible en el momento correcto a un gran número de consumidores de los datos la captura de los datos desde el disco debe coordinarse cuidadosamente. Generalmente los datos se capturan en ciclos periódicos. En cada ciclo, digamos de n segundos, se capturan n segundos de datos para cada consumidor y se almacenan en las memorias intermedias, mientras los datos capturados en el ciclo anterior se envían a los consumidores desde las memorias intermedias. El periodo de los ciclos es un compromiso: un periodo corto usa menos memoria pero necesita más movimientos del brazo del disco, lo que supone un desperdicio de recursos, mientras que un periodo largo reduce el movimiento del brazo del disco pero aumenta las necesidades de memoria y puede retrasar la entrega inicial de datos. Cuando llega una nueva solicitud entra en acción el **control de admisión**: es decir, el sistema comprueba si se puede satisfacer la solicitud con los recursos disponibles (en cada periodo); si es así, se admite; en caso contrario, se rechaza.

La extensa investigación realizada sobre la entrega de datos de medios continuos ha tratado aspectos como el manejo de arrays de discos y el tratamiento de los fallos de los discos. Véanse las referencias bibliográficas para obtener más detalles.

Varios fabricantes ofrecen servidores de vídeo bajo demanda. Los sistemas actuales están basados en los sistemas de archivos, ya que los sistemas de bases de datos existentes no proporcionan la respuesta en tiempo real que necesitan estas aplicaciones. La arquitectura básica de un sistema de vídeo bajo demanda comprende:

- **Servidor de vídeo**. Los datos multimedia se almacenan en varios discos (generalmente en una configuración RAID). Puede que los sistemas que contienen un gran volumen de datos utilicen medios de almacenamiento terciario para los datos a los que se tiene acceso con menor frecuencia.
- **Terminales**. Los datos multimedia se examinan mediante varios dispositivos, colectivamente denominados *terminales*. Ejemplos son las computadoras personales y los televisores conectados a una computadora pequeña y de coste reducido denominada **microcomputadora**.
- **Red**. La transmisión de los datos multimedia desde el servidor hasta los terminales necesita una red de gran capacidad.

El servicio de vídeo bajo demanda en redes de cable está disponible en muchos lugares actualmente y acabará siendo ubicuo, igual que lo son ahora la televisión por ondas hercianas y la televisión por cable.

24.4.3 Recuperación basada en la semejanza

En muchas aplicaciones multimedia los datos sólo se describen en la base de datos de manera aproximada. Un ejemplo son los datos de huellas dactilares del Apartado 24.4. Otros ejemplos son:

- **Datos gráficos**. Dos gráficos o imágenes que sean ligeramente diferentes en su representación en la base de datos pueden ser considerados iguales por un usuario. Por ejemplo, una base de datos puede almacenar diseños de marcas comerciales. Cuando haya que registrar una nueva marca puede que el sistema necesite identificar antes todas las marcas parecidas que se registraron anteriormente.
- **Datos de sonido**. Se están desarrollando interfaces de usuario basadas en el reconocimiento de la voz que permiten a los usuarios dar un comando o identificar un elemento de datos por la voz. Debe comprobarse la semejanza de la entrada del usuario con los comandos o los elementos de datos almacenados en el sistema.
- **Datos manuscritos**. La entrada manuscrita puede usarse para identificar un elemento de datos manuscrito o una orden manuscrita almacenados en la base de datos. Una vez más, se necesita comprobar la semejanza.

El concepto de semejanza suele ser subjetivo y específico del usuario. No obstante, la verificación de la semejanza suele tener más éxito que el reconocimiento de voz o de letras manuscritas, ya que la entrada puede compararse con datos que ya se hallan en el sistema y, por tanto, el conjunto de opciones disponibles para el sistema es limitado.

Existen varios algoritmos para hallar las mejores coincidencias con una entrada dada mediante la comprobación de la semejanza. Algunos sistemas, incluidos un sistema telefónico de llamada por nombre activado por la voz, se han distribuido comercialmente. Véanse las notas bibliográficas para hallar más referencias.

24.5 Computadoras portátiles y bases de datos personales

Las bases de datos comerciales de gran tamaño se han almacenado tradicionalmente en las instalaciones informáticas centrales. En las aplicaciones de bases de datos distribuidas ha habido generalmente una fuerte administración central de las bases de datos y de la red. Se han combinado las siguientes ten-

dencias tecnológicas para crear aplicaciones en las cuales la suposición de una administración y de un control centralizados no es completamente correcta:

1. El uso cada vez más extendido de computadoras personales y, sobre todo, de computadoras portátiles.
2. El desarrollo de una infraestructura inalámbrica de comunicaciones digitales de coste relativamente bajo, basada en redes inalámbricas de área local, redes celulares de paquetes digitales y otras tecnologías.

La **informática móvil** se ha mostrado útil en muchas aplicaciones. Muchos profesionales usan computadoras portátiles para poder trabajar y tener acceso a los datos durante sus viajes. Los servicios de mensajería usan computadoras portátiles para ayudar al seguimiento de los paquetes. Los servicios de emergencia usan computadoras portátiles en el escenario de los desastres, en las emergencias médicas y similares para tener acceso a la información y para introducir datos relativos a la situación. Los teléfonos móviles (celulares) se están convirtiendo en dispositivos que no sólo proporcionan servicios de telefonía, sino que permiten el correo electrónico y el acceso Web. Siguen surgiendo nuevas aplicaciones de las computadoras portátiles.

Las computadoras comunicadas por radio crean una situación en que las máquinas ya no tienen ubicaciones fijas ni direcciones de red. Las **consultas dependientes de la ubicación** son una clase interesante de consultas que está motivada por las computadoras portátiles; en estas consultas la ubicación del usuario (computadora) es un parámetro de la consulta. El valor del parámetro de ubicación lo proporciona el usuario o, cada vez más, un sistema de posicionamiento global (GPS). Un ejemplo son los sistemas de información para viajeros que proporcionan a los conductores datos sobre los hoteles, los servicios de carretera y similares. El procesamiento de las consultas sobre los servicios que se hallan más adelante en la ruta actual debe basarse en la ubicación del usuario, en su dirección de movimiento y en su velocidad. Se ofrecen cada vez más ayudas a la navegación como una característica integrada en los automóviles.

La energía (la carga de las baterías) es un recurso escaso para la mayor parte de las computadoras portátiles. Esta limitación influye en muchos aspectos del diseño de los sistemas. Entre las consecuencias más interesantes de la necesidad de eficiencia energética es que los pequeños dispositivos móviles emplean la mayor parte del tiempo hibernándose, despertándose durante una fracción de segundo cada segundo aproximadamente para comprobar si hay datos entrantes y para enviar datos. Este comportamiento tiene un impacto significativo en los protocolos usados para comunicarse con los dispositivos móviles. El uso de emisiones programadas de datos en los sistemas móviles para reducir la necesidad de transmisión de consultas es otra forma de reducir los requisitos de energía.

Cantidades cada vez mayores de datos residen en máquinas administradas por los usuarios en lugar de por administradores de bases de datos. Además, estas máquinas pueden estar, a veces, desconectadas de la red. En muchos casos hay un conflicto entre la necesidad del usuario de seguir trabajando mientras está desconectado y la necesidad de consistencia global de los datos.

En los Apartados 24.5.1 a 24.5.4 se estudian técnicas en uso y en desarrollo para tratar los problemas de las computadoras portátiles y de la informática personal.

24.5.1 Un modelo de informática móvil

El entorno de la informática móvil consiste en computadoras portátiles, denominadas **anfitriones móviles**, y una red de computadoras conectadas por cable. Los anfitriones móviles se comunican con la red de cable mediante computadoras denominadas **estaciones para el soporte de movilidad**. Cada estación para el soporte de movilidad gestiona los anfitriones móviles de su **celda**—es decir, del área geográfica que cubre. Los anfitriones móviles pueden moverse de unas celdas a otras, por lo que necesitan el **relevo** del control de una estación para el soporte de movilidad a otra. Dado que los anfitriones móviles pueden, a veces, estar apagados, un anfitrión puede abandonar una celda y aparecer más tarde en otra distante. Por tanto, los movimientos de unas celdas a otras no se realizan necesariamente entre celdas adyacentes. Dentro de un área pequeña, como un edificio, los anfitriones móviles pueden conectarse

mediante una red inalámbrica de área local que proporciona conectividad de coste más reducido que las redes celulares de área amplia, y que reduce la sobrecarga de entregas.

Es posible que los anfitriones móviles se comuniquen directamente sin intervención de ninguna estación para el soporte de movilidad. No obstante, esa comunicación sólo puede ocurrir entre anfitriones cercanos. Estas formas directas de comunicación se están haciendo más frecuentes con la llegada del estándar **Bluetooth**. Bluetooth usa radio digital de corto alcance para permitir la conectividad por radio a alta velocidad (hasta 721 kilobits por segundo) a distancias inferiores a diez metros. Concebido inicialmente como una sustitución de los cables, lo más prometedor de Bluetooth es la conexión ad hoc sencilla entre computadoras portátiles, PDAs, teléfonos celulares y las denominadas aplicaciones inteligentes.

Los sistemas de redes de área local inalámbricas basados en las normas 801.11 (a/b/g) se usan mucho actualmente y los sistemas basados en 802.16 (Wi-Max) aparecieron en 2005.

La infraestructura de red para la informática móvil consiste en gran parte en dos tecnologías: redes inalámbricas locales y redes de telefonía celular basadas en paquetes. Los primeros sistemas celulares usaban tecnología analógica y estaban diseñados para la comunicación de voz. Los sistemas digitales de segunda generación siguieron centrándose en las aplicaciones de voz. Los sistemas de tercera generación (3G) y los denominados sistemas 2.5G usan redes basadas en paquetes y están más adaptados a las aplicaciones de datos. En estas redes la voz es sólo una más de las aplicaciones (aunque una económicamente importante).

Bluetooth, las redes de área local inalámbricas y las redes celulares 2.5G y 3G hacen posible que se comuniquen a bajo coste gran variedad de dispositivos. Aunque esta comunicación en sí misma no encaja en el dominio de las aplicaciones habituales de bases de datos, los datos de la contabilidad, del control y de la administración correspondientes a esta comunicación generan bases de datos enormes. La inmediatez de la comunicación por radio genera la necesidad de acceso en tiempo real a muchas de estas bases de datos. Esta necesidad de inmediatez añade otra dimensión a las restricciones del sistema —un asunto que se abordará en profundidad en el Apartado 24.3.

El tamaño y las limitaciones de potencia de muchas computadoras portátiles han llevado a la creación de jerarquías de memoria alternativas. En lugar de, o además de, el almacenamiento en disco, puede incluirse la memoria flash, que se estudió en el Apartado 11.1. Si el anfitrión móvil incluye un disco duro, puede que se permita que el disco deje de girar cuando no se utilice, para ahorrar energía. Las mismas consideraciones de tamaño y de energía limitan el tipo y el tamaño de las pantallas usadas en los dispositivos portátiles. Los diseñadores de los dispositivos portátiles suelen crear interfaces de usuario especiales para trabajar con estas restricciones. No obstante, la necesidad de presentar datos basados en Web ha exigido la creación de estándares para presentaciones. El **protocolo de aplicaciones inalámbrico** (Wireless Application Protocol, WAP) es un estándar para el acceso inalámbrico a Internet. Los exploradores basados en WAP tienen acceso a páginas Web especiales que usan el **lenguaje de marcas inalámbrico** (Wireless Markup Language, WML), un lenguaje basado en XML diseñado para las restricciones de la exploración Web móvil e inalámbrica.

24.5.2 Encaminamiento y procesamiento de consultas

La ruta entre cada par de anfitriones puede cambiar con el tiempo si alguno de los dos anfitriones es móvil. Este sencillo hecho tiene un efecto espectacular en el nivel de la red, ya que las direcciones de red basadas en las ubicaciones ya no son constantes en el sistema.

La informática móvil también afecta directamente al procesamiento de consultas de las bases de datos. Como se vio en el Capítulo 22, hay que considerar los costes de comunicación cuando se escoge una estrategia de procesamiento distribuido de las consultas. La informática móvil hace que los costes de comunicación cambien de manera dinámica, lo que complica el proceso de optimización. Además, varios conceptos de coste que hay que considerar en relación con los demás:

- El **tiempo del usuario** es una materia prima muy valiosa en muchas aplicaciones profesionales.
- El **tiempo de conexión** es la unidad por la que se asignan los costes monetarios en algunos sistemas de telefonía celular.

- El **número de bytes, o de paquetes, transferidos** es la unidad por la que se calculan los costes en algunos sistemas de telefonía celular digital.
- Los **costes basados en la hora del día** varían, en función de si la comunicación se produce durante los periodos pico o durante los periodos valle.
- La **energía** es limitada. A menudo la energía de las baterías es un recurso escaso cuyo uso debe optimizarse. Un principio básico de las comunicaciones inalámbricas es que hace falta menos energía para recibir señales de radio que para emitirlas. Así, la transmisión y la recepción de los datos imponen demandas de energía diferentes al anfitrión móvil.

24.5.3 Datos de difusión

Suele ser deseable para los datos que se solicitan con frecuencia que los transmitan las estaciones de soporte de las computadoras portátiles en un ciclo continuo, en lugar de que se transmitan a los anfitriones móviles a petición de éstos. Una aplicación típica de estos **datos de difusión** es la información de las cotizaciones bursátiles. Hay dos motivos para usar los datos de difusión. En primer lugar, los anfitriones móviles evitan el coste energético de transmitir las solicitudes de datos. En segundo lugar, los datos de difusión pueden recibirlos simultáneamente gran número de anfitriones móviles, sin coste adicional. Por tanto, el ancho de banda disponible para transmisiones se usa de manera más efectiva.

Así, los anfitriones móviles pueden recibir los datos a medida que se transmiten, en lugar de consumir energía transmitiendo solicitudes. Puede que los anfitriones móviles tengan almacenamiento no volátil disponible para guardar en la caché los datos de difusión para su uso posterior. Dada una consulta, los anfitriones móviles pueden minimizar los costes energéticos determinando si pueden procesarla sólo con los datos guardados en la caché. Si los datos guardados en la caché son insuficientes, hay dos opciones: esperar a que los datos se transmitan o transmitir una solicitud de datos. Para tomar esta decisión los anfitriones móviles deben conocer el momento en que se transmitirán los datos en cuestión.

Los datos de difusión pueden transmitirse de acuerdo con una programación fija o según una programación variable. En el primer caso, los anfitriones móviles usan la programación fija conocida para determinar el momento en que se transmitirán los datos en cuestión. En el segundo caso, se debe transmitir la propia programación de transmisiones en una frecuencia de radio conocida y a intervalos de tiempos conocidos.

En efecto, el medio transmitido puede modelarse como un disco con una latencia elevada. Las solicitudes de datos pueden considerarse atendidas cuando los datos solicitados se transmiten. Las programaciones de transmisión se comportan como los índices de los discos. Las notas bibliográficas citan trabajos de investigación recientes en el área de administración de los datos de difusión.

24.5.4 Desconexiones y consistencia

Dado que puede que las comunicaciones inalámbricas se paguen con arreglo al tiempo de conexión, hay un incentivo para que se desconecten determinados anfitriones móviles durante periodos de tiempo considerables. Las computadoras portátiles sin conectividad inalámbrica están desconectadas la mayor parte del tiempo en que se usan, excepto cuando se conectan de manera periódica a sus computadoras anfitrionas, físicamente o mediante una red informática.

Durante esos periodos de desconexión puede que el anfitrión móvil siga operativo. Puede que el usuario del anfitrión móvil formule consultas y solicite actualizaciones de los datos que residen localmente o que se han guardado en la caché local. Esta situación crea varios problemas, en especial:

- **Recuperabilidad**. Las actualizaciones introducidas en una máquina desconectada pueden perderse si el anfitrión móvil sufre un fallo catastrófico. Dado que el anfitrión móvil representa un único punto de fallo, no se puede simular bien el almacenamiento estable.
- **Consistencia**. Los datos guardados en la caché local pueden quedar obsoletos, pero el anfitrión móvil no puede descubrir la situación hasta que vuelva a conectarse. De manera parecida, las actualizaciones que se produzcan en el anfitrión móvil no pueden transmitirse hasta que no se produzca de nuevo la conexión.

El problema de la consistencia se estudió en el Capítulo 22, donde se estudiaron las particiones de la red, y aquí se partirá de esa base. En los sistemas distribuidos conectados por redes físicas las particiones se consideran un modo de fallo; en la informática móvil las particiones mediante desconexiones son parte del modo de operación normal. Por tanto, es necesario permitir que continúe el acceso a los datos a pesar de las particiones, pese al riesgo de que se produzca una pérdida de consistencia.

Para los datos actualizados sólo por el anfitrión móvil, es sencillo transmitir las actualizaciones cuando el anfitrión móvil vuelve a conectarse. No obstante, si el anfitrión móvil guarda en la caché copias de los datos sólo para lectura que pueden actualizar otras computadoras, puede que los datos guardados en la caché acaben siendo inconsistentes. Cuando se conecta el anfitrión móvil, puede recibir **informes de invalidación** que lo informen de las entradas de la caché que están obsoletas. No obstante, cuando el anfitrión móvil esté desconectado puede perder algún informe de invalidación. Una solución sencilla a este problema es invalidar toda la caché al volver a conectar el anfitrión móvil, pero una solución tan extrema resulta muy costosa. En las notas bibliográficas se citan varios esquemas para el almacenamiento en la caché.

Si se pueden producir actualizaciones tanto en el anfitrión móvil como en el resto del sistema, la detección de las actualizaciones conflictivas resulta más difícil. Los esquemas basados en la **numeración de versiones** permiten las actualizaciones de los archivos compartidos desde los anfitriones desconectados. Estos esquemas no garantizan que las actualizaciones sean consistentes. Más bien, garantizan que, si dos anfitriones actualizan de manera independiente la misma versión del documento, el conflicto se acabará descubriendo, cuando los anfitriones intercambien información directamente o mediante un anfitrión común.

El **esquema del vector de versiones** detecta las inconsistencias cuando las copias de un documento se actualizan de manera independiente. Este esquema permite que las copias de un *documento* se almacenen en varios anfitriones. Aunque se utilice el término *documento*, el esquema puede aplicarse a otros elementos de datos, como las tuplas de una relación.

La idea básica es que cada anfitrión i almacene, con su copia de cada documento d, un **vector de versiones**—es decir, un conjunto de números de versiones $\{V_{d,i}[j]\}$, con una entrada para cada uno de los demás anfitriones d en los que se puede actualizar potencialmente el documento. Cuando un anfitrión i actualiza un documento d, incrementa el número de versión $V_{d,i}[i]$ en una unidad.

Siempre que dos anfitriones i y d se conectan entre sí, intercambian los documentos actualizados, de modo que los dos obtienen versiones nuevas de los documentos. No obstante, antes de intercambiar los documentos, los anfitriones tienen que averiguar si las copias son consistentes:

1. Si los vectores de versiones son iguales en los dos anfitriones—es decir, si para cada k, $V_{d,i}[k] = V_{d,j}[k]$—las copias del documento d son idénticas.
2. Si, para cada k, $V_{d,i}[k] \leq V_{d,j}[k]$ y los vectores de versiones no son idénticos, la copia del documento d del anfitrión i es más antigua que la del anfitrión d. Es decir, la copia del documento d del anfitrión d se obtuvo mediante una o más modificaciones de la copia del documento del anfitrión i. El anfitrión i sustituye su copia de d, así como su copia del vector de versiones de d, por las copias del anfitrión d.
3. Si hay un par de anfitriones k y m tales que $V_{d,i}[k] < V_{d,j}[k]$ y $V_{d,i}[m] > V_{d,j}[m]$, las copias son *inconsistentes*; es decir, la copia de d de i contiene actualizaciones realizadas por el anfitrión k que no se han transmitido al anfitrión d y, de manera parecida, la copia de d de d contiene actualizaciones llevadas a cabo por el anfitrión m que no se han transmitido al anfitrión i. Entonces, las copias de d son inconsistentes, ya que se han realizado de manera independiente dos o más actualizaciones de d. Puede que se necesite la intervención manual para mezclar las actualizaciones.

El esquema de vectores de versiones se diseñó inicialmente para tratar los fallos en los sistemas de archivos distribuidos. El esquema adquirió mayor importancia porque las computadoras portátiles suelen almacenar copias de los archivos que también se hallan presentes en los sistemas servidores, lo que constituye de facto un sistema de archivos distribuido que suele estar desconectado. Otra aplicaciones de este esquema son los sistemas en grupo, en que los anfitriones se conectan de manera periódica, en lugar de hacerlo de manera continua, y deben intercambiar los documentos actualizados. El esquema del vector de versiones también tiene aplicaciones en las bases de datos replicadas.

No obstante, el esquema del vector de versiones no logra abordar el problema más difícil e importante que plantean las actualizaciones de los datos compartidos—la reconciliación de las copias inconsistentes de los datos. Muchas aplicaciones pueden llevar a cabo de manera automática la reconciliación ejecutando en cada computadora las operaciones que han conducido a las actualizaciones en las computadoras remotas durante el periodo de desconexión. Esta solución funciona si las operaciones de actualización conmutan—es decir, generan el mismo resultado, independientemente del orden en que se ejecuten. Puede que se disponga de técnicas alternativas en ciertas aplicaciones; en el peor de los casos, no obstante, debe dejarse a los usuarios que resuelvan las inconsistencias. El tratamiento automático de estas inconsistencias y la ayuda a los usuarios para que resuelvan las que no puedan tratarse de manera automática sigue siendo un área de investigación.

Otra debilidad es que el esquema del vector de versiones exige una comunicación sustancial entre el anfitrión móvil que vuelve a conectarse y su estación para el soporte de movilidad. Las verificaciones de la consistencia pueden posponerse hasta que se necesiten los datos, aunque este retraso puede incrementar la inconsistencia global de la base de datos.

La posibilidad de desconexión y el coste de las comunicaciones inalámbricas limitan el aspecto práctico de las técnicas de procesamiento de las transacciones para los sistemas distribuidos estudiadas en el Capítulo 22. A menudo resulta preferible dejar que los usuarios preparen las transacciones en los anfitriones móviles y exigir que, en lugar de ejecutarlas localmente, las remitan al servidor para su ejecución. Las transacciones que afectan a más de una computadora y que incluyen un anfitrión móvil afrontan bloqueos de larga duración durante el compromiso de la transacción, a menos que las desconexiones sean raras o predecibles.

24.6 Resumen

- El tiempo desempeña un papel importante en los sistemas de bases de datos. Las bases de datos son modelos del mundo real. Aunque la mayor parte de las bases de datos modelan el estado del mundo real en un momento dado (en el momento actual), las bases de datos temporales modelan los estados del mundo real a lo largo del tiempo.
- Los hechos de las relaciones temporales tienen momentos asociados cuando son válidos, que pueden representarse como una unión de intervalos. Los lenguajes de consultas temporales simplifican el modelado del tiempo, así como las consultas relacionadas con el tiempo.
- Las bases de datos espaciales se usan cada vez más hoy en día para almacenar datos de diseño asistido por computadora y datos geográficos.
- Los datos de diseño se almacenan sobre todo como datos vectoriales; los datos geográficos consisten en una combinación de datos vectoriales y lineales. Las restricciones de integridad espacial son importantes para los datos de diseño.
- Los datos vectoriales pueden codificarse como datos de la primera forma normal o almacenarse mediante estructuras que no sean la primera forma normal, como las listas. Las estructuras de índices de finalidad espacial resultan especialmente importantes para tener acceso a los datos espaciales y para procesar las consultas espaciales.
- Los árboles R son una extensión multidimensional de los árboles B; con variantes como los árboles R+ y los árboles R^*, se han hecho populares en las bases de datos espaciales. Las estructuras de índices que dividen el espacio de manera regular, como los árboles cuadráticos, ayudan a procesar las consultas de mezcla espaciales.
- Las bases de datos multimedia están aumentando de importancia. Problemas como la recuperación basada en la semejanza y la entrega de datos a velocidades garantizadas son temas de investigación actuales.
- Los sistemas de informática móvil se han vuelto de uso común, lo que ha llevado al interés por los sistemas de bases de datos que pueden ejecutarse en ellos. El procesamiento de las consultas en estos sistemas puede implicar la búsqueda en las bases de datos de los servidores. El modelo

de coste de las consultas debe incluir el coste de la comunicación, incluido el coste monetario y el coste de la energía de las baterías, que resulta relativamente elevado para los sistemas portátiles.

- La transmisión resulta mucho más económica por receptor que la comunicación punto a punto, y la transmisión de datos como los datos bursátiles ayuda a los sistemas portátiles a recoger los datos de manera económica.
- La operación en desconexión, el uso de los datos de difusión y el almacenamiento de los datos en la caché son tres problemas importantes que se están abordando hoy en día en la informática móvil.

Términos de repaso

- Datos temporales.
- Tiempo válido.
- Tiempo de transacción.
- Relación temporal.
- Relación bitemporal.
- Tiempo universal coordinado (UTC).
- Relación instantánea.
- Lenguajes de consultas temporales.
- Selección temporal.
- Proyección temporal.
- Reunión temporal.
- Mezcla temporal.
- Datos espaciales y geográficos.
- Datos de diseño asistido por computadora (Computer Aided Design, CAD).
- Datos geográficos.
- Sistemas de información geográfica.
- Triangulación.
- Bases de datos de diseño.
- Datos geográficos.
- Datos por líneas (raster).
- Datos vectoriales.
- Sistema de posicionamiento global (Global Positioning System, GPS).
- Consultas espaciales.
- Consultas de proximidad.
- Consultas de vecino más próximo.
- Consultas regionales.
- Reunión espacial.
- Indexado de los datos espaciales.
- Árboles k-d.
- Árboles k-d B.
- Árboles cuadráticos:
  - PR.
  - Regional.
- Árboles R.
  - Caja límite.
  - División cuadrática.
- Bases de datos multimedia.
- Datos isócronos.
- Datos de medios continuos.
- Recuperación basada en la semejanza.
- Formatos de datos multimedia.
- Servidores de vídeo.
- Informática móvil.
  - Anfitriones móviles.
  - Estaciones de soporte de las computadoras portátiles.
  - Celda.
  - Relevo.
- Consultas dependientes de la ubicación.
- Datos de difusión.
- Consistencia.
  - Informes de invalidación.
  - Esquema del vector de versiones.

Ejercicios prácticos

24.1 Indíquense los dos tipos de tiempo y en lo que se diferencian. Explíquese el motivo de que haya dos tipos de tiempo asociados con cada tupla.

24.2 Supóngase que se tiene una relación que contiene las coordenadas x, y y los nombres de varios restaurantes. Supóngase también que las únicas consultas que se formularán serán de la forma

siguiente: la consulta especifica un punto y pregunta si hay algún restaurante exactamente en ese punto. Indíquese el tipo de índice que sería preferible, árbol R o árbol B. Explíquese el motivo.

24.3 Supóngase que se dispone de una base de datos espacial que soporta consultas regionales (con regiones circulares) pero no consultas de vecino más próximo. Descríbase un algoritmo para hallar el vecino más próximo haciendo uso de varias consultas regionales.

24.4 Supóngase que se desean almacenar segmentos rectilíneos en un árbol R. Si un segmento rectilíneo no es paralelo a los ejes, su caja límite puede ser grande y contener una gran área vacía.

- Descríbase el efecto en el rendimiento de tener cajas límite de gran tamaño en las consultas que piden los segmentos rectilíneos que intersectan una región dada.
- Descríbase brevemente una técnica para mejorar el rendimiento de esas consultas y dese un ejemplo de sus ventajas. *Sugerencia*: se pueden dividir los segmentos en partes más pequeñas.

24.5 Dese un procedimiento recursivo para calcular de manera eficiente la mezcla espacial de dos relaciones con índices de árbol R. *Sugerencia*: úsense cajas límite para comprobar si las entradas hojas bajo un par de nodos internos pueden intersectarse.

24.6 Descríbase el modo en que las ideas subyacentes a la organización RAID (Apartado 11.3) pueden usarse en un entorno de datos de difusión, donde puede que haya ocasionalmente ruido que impida la recepción de parte de los datos que se están transmitiendo.

24.7 Defínase un modelo en que se difundan repetidamente los datos en el que el medio de transmisión se modele como un disco virtual. Descríbase el modo en que el tiempo de acceso y la velocidad de transferencia de datos del disco virtual se diferencian de los valores correspondientes a un disco duro normal.

24.8 Considérese una base de datos de documentos en la que todos los documentos se conserven en una base de datos central. En las computadoras portátiles se guardan copias de algunos documentos. Supóngase que la computadora portátil A actualiza una copia del documento 1 mientras está desconectada y que, al mismo tiempo, la computadora portátil B actualiza una copia del documento 2 mientras está desconectada. Muéstrese el modo en que el esquema del vector versión puede asegurar la actualización adecuada de la base de datos central y de las computadoras portátiles cuando se vuelva a conectar una computadora portátil.

Ejercicios

24.9 Indíquese si se conservarán las dependencias funcionales si se convierte una relación en una relación temporal añadiéndole un atributo temporal. Indíquese el modo en que se resuelve el problema en las bases de datos temporales.

24.10 Considérense dos datos vectoriales bidimensionales en que los elementos de datos no se solapan. Indíquese si es posible convertir esos datos vectoriales en datos lineales. En caso de que sea posible, indíquense los inconvenientes de almacenar los datos lineales obtenidos de esa conversión en lugar de los datos vectoriales originales.

24.11 Estúdiese el soporte de los datos espaciales ofrecido por el sistema de bases de datos que se está usando e impleméntese lo siguiente:

a. Un esquema para representar la ubicación geográfica de los restaurantes y características como la cocina que se sirve en cada restaurante y su nivel de precios.
b. Una consulta para hallar los restaurantes económicos que sirven comida india y que se hallan a menos de nueve kilómetros de casa del lector (supóngase cualquier ubicación para la casa del lector).
c. Una consulta para hallar para cada restaurante su distancia al restaurante más cercano que sirve la misma cocina y con el mismo nivel de precios.

24.12 Indíquense los problemas que se producen en un sistema de medios continuos si los datos se entregan demasiado lento o demasiado rápido.

24.13 Indíquense tres características principales de la informática móvil en redes inalámbricas que son diferentes de las de los sistemas distribuidos tradicionales.

24.14 Indíquense tres factores que haya que considerar en la optimización de las consultas para la informática móvil que no se consideren en los optimizadores de consultas tradicionales.

24.15 Dese un ejemplo para mostrar que el esquema del vector versión no asegura la secuenciabilidad. *Sugerencia*: utilícese el ejemplo del Ejercicio práctico 24.8, con la suposición de que los documentos 1 y 2 están disponibles en las dos computadoras portátiles A y B, y téngase en cuenta la posibilidad de que un documento pueda leerse sin que se actualice.

Notas bibliográficas

Stam y Snodgrass [1988] y Soo [1991] proporcionan estudios sobre la administración de los datos temporales. Jensen et al. [1994] presentan un glosario de conceptos de las bases de datos temporales, con la intención de unificar la terminología. Tansel et al. [1993] es una colección de artículos sobre diferentes aspectos de las bases de datos temporales. Chomicki [1995] presenta técnicas para administrar las restricciones para la integridad temporal.

Heywood et al. [2002] es un libro que estudia los sistemas de información geográfica. Samet [1995b] proporciona una introducción a la gran cantidad de trabajo realizado sobre las estructuras espaciales de índices. Samet [1990] proporciona una cobertura en el nivel de los libros de texto de las estructuras espaciales de datos. Una de las primeras descripciones de los árboles cuadráticos se proporciona en Finkel y Bentley [1974]. Samet [1990] y Samet [1995b] describen numerosas variantes de los árboles cuadráticos. Bentley [1975] describe los árboles k-d, y Robinson [1981] describe los árboles k-d B. Los árboles R se presentaron originalmente en Guttman [1984]. Las extensiones de los árboles R se presentan en Sellis et al. [1987], que describen los árboles R^+, y Beckmann et al. [1990], que describen los árboles R^* tree.

Brinkhoff et al. [1993] estudian una implementación de las mezclas espaciales mediante árboles R. Lo y Ravishankar [1996] y Patel y DeWitt [1996] presentan los métodos basados en las particiones para el cálculo de las mezclas espaciales. Samet y Aref [1995] proporcionan una introducción de los modelos espaciales de datos, de las operaciones espaciales y de la integración de los datos espaciales con los no espaciales.

Revesz [2002] es un libro que estudia el área de las bases de datos con restricciones; los intervalos temporales y las regiones espaciales se pueden pensar como un caso especial de restricciones.

Samet [1995a] describe los campos de investigación en las bases de datos multimedia. El indexado de los datos multimedia se estudia en Faloutsos y Lin [1995].

Dashti et al. [2003] es un libro que describe el diseño de servidores de flujos de datos de medios, incluyendo un estudio extensivo de la organización de datos en subsistemas de discos. Los servidores de vídeo se estudian en Anderson et al. [1992], Rangan et al. [1992], Ozden et al. [1994], Freedman y DeWitt [1995] y Ozden et al. [1996b]. La tolerancia a los fallos se estudia en Berson et al. [1995] y Ozden et al. [1996a].

La administración de la información en los sistemas que incluyen computadoras portátiles se estudia en Alonso y Korth [1993] y en Imielinski y Badrinath [1994]. Imielinski y Korth [1996] presentan una introducción a la informática móvil y una colección de trabajos de investigación sobre el tema.

El esquema del vector de versiones para la detección de la inconsistencia en los sistemas de archivos distribuidos se describe en Popek et al. [1981] y en Parker et al. [1983].

CAPÍTULO 25

Procesamiento avanzado de transacciones

En los Capítulos 15, 16 y 17 se introdujo el concepto de transacción, que es una unidad de programa que tiene acceso—y posiblemente actualiza—varios elementos de datos, y cuya ejecución asegura la conservación de las propiedades ACID. En esos capítulos se estudiaron gran variedad de esquemas para asegurar las propiedades ACID en entornos en los que pueden producirse fallos, y en los que las transacciones pueden ejecutarse de manera concurrente.

En este capítulo se irá más allá de los esquemas básicos estudiados anteriormente y se abordarán los conceptos del procesamiento avanzado de las transacciones, incluidos los monitores de procesamiento de transacciones, los flujos de trabajo de las transacciones y el procesamiento de transacciones en el contexto del comercio electrónico. También se estudian las bases de datos en memoria principal, las bases de datos en tiempo real, las transacciones de larga duración, las transacciones anidadas y las transacciones con múltiples bases de datos.

25.1 Monitores de procesamiento de transacciones

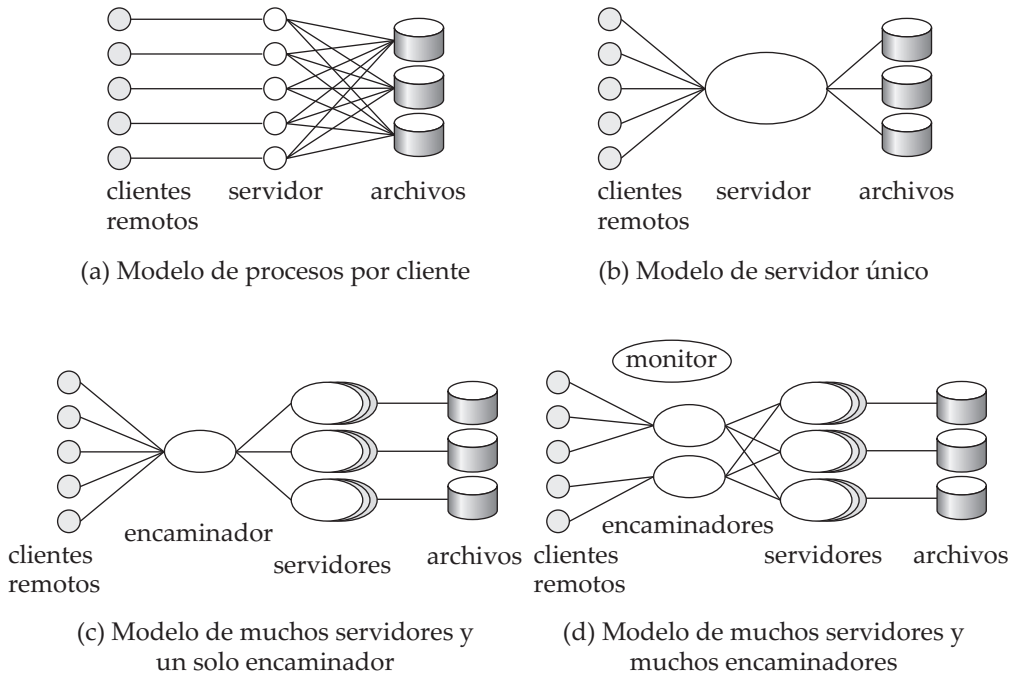
Los **monitores de procesamiento de transacciones** (Transaction Processing monitors, monitores TP) son sistemas que se desarrollaron en los años setenta y ochenta del siglo pasado, inicialmente como respuesta a la necesidad de soportar gran número de terminales remotas (como los terminales de reserva de las líneas aéreas) desde una sola computadora. El término *monitor TP* significaba inicialmente *monitor de teleprocesamiento* (Teleprocessing Monitor).

Los monitores TP han evolucionado desde entonces para ofrecer el soporte central para el procesamiento distribuido de las transacciones, y el término monitor TP ha adquirido su significado actual. El monitor TP CICS de IBM fue uno de los primeros monitores TP, y se ha usado mucho. Entre los monitores TP de la generación actual están Tuxedo y Top End (los dos actualmente de BEA Systems), Encina (de Transarc, que ahora forma parte de IBM) y Transaction Server (de Microsoft).

Las arquitecturas de servidores de aplicaciones Web, incluyendo los servlets, que se estudiaron anteriormente en el Apartado 8.4, proporcionan soporte a muchas de las características de los monitores TP y se conocen a veces como "TP ligeros". Los servidores de aplicaciones Web se usan ampliamente y han reemplazado a los monitores TP tradicionales para muchas aplicaciones. Sin embargo, los conceptos subyacentes en ellos, que se estudian en este apartado, son esencialmente iguales.

25.1.1 Arquitecturas de los monitores TP

Los sistemas de procesamiento de transacciones a gran escala se construyen en torno a una arquitectura cliente–servidor. Una manera de crear estos sistemas es tener un proceso servidor para cada cliente; el servidor realiza la autentificación, y luego ejecuta las acciones solicitadas por el cliente. Este **modelo de proceso por cliente** se ilustra en la Figura 25.1a. Este modelo presenta varios problemas con respecto al uso de la memoria y a la velocidad de procesamiento:



(a) Modelo de procesos por cliente

(b) Modelo de servidor único

(c) Modelo de muchos servidores y un solo encaminador

(d) Modelo de muchos servidores y muchos encaminadores

Figura 25.1 Arquitecturas de los monitores TP.

- Los requisitos de memoria para cada proceso son elevados. Aunque se comparta la memoria para el código de los programas entre todos los procesos, cada proceso consume memoria para los datos locales y los descriptores de los archivos abiertos, así como para la sobrecarga del sistema operativo, como las tablas de páginas para soportar la memoria virtual.
- El sistema operativo divide el tiempo disponible de CPU entre los procesos conmutando entre ellos; esta tarea se denomina **multitarea**. Cada **cambio de contexto** entre un proceso y el siguiente supone una sobrecarga considerable de la CPU; incluso en los sistemas rápidos de hoy en día un cambio de contexto puede tardar cientos de microsegundos.

Los problemas anteriores pueden evitarse teniendo un proceso con un solo servidor al que se conecten todos los servidores; este modelo se denomina **modelo de servidor único**, ilustrado en la Figura 25.1b. Los clientes remotos envían las solicitudes al proceso del servidor, que ejecuta entonces esas solicitudes. Este modelo también se usa en los entornos cliente–servidor, en los que los clientes envían solicitudes a un proceso de un solo servidor. El proceso servidor asume las tareas, como la autentificación de los usuarios, que normalmente asumiría el sistema operativo. Para evitar bloquear otros clientes al procesar una solicitud de larga duración de un cliente, el servidor usa **multienhebramiento**: el proceso servidor dispone una hebra de control para cada cliente y, en efecto, implementa su propia multitarea de baja sobrecarga. Ejecuta el código en nombre de un cliente durante un rato, luego guarda el contexto interno y conmuta al código de otro cliente. A diferencia de la sobrecarga de la multitarea, el coste de la conmutación entre hebras es reducido (generalmente sólo unos pocos microsegundos).

Los sistemas basados en el modelo de servidor único, como la versión original del monitor TP CICS de IBM y los servidores de archivos como NetWare de Novell, proporcionaban con éxito tasas de transacciones elevadas con recursos limitados. No obstante, tenían problemas, especialmente cuando varias aplicaciones tenían acceso a la misma base de datos:

- Dado que todas las aplicaciones se ejecutan como un único proceso, no hay protección entre ellas. Un fallo en una aplicación puede afectar también a todas las demás aplicaciones. Sería mejor ejecutar cada aplicación como un proceso separado.

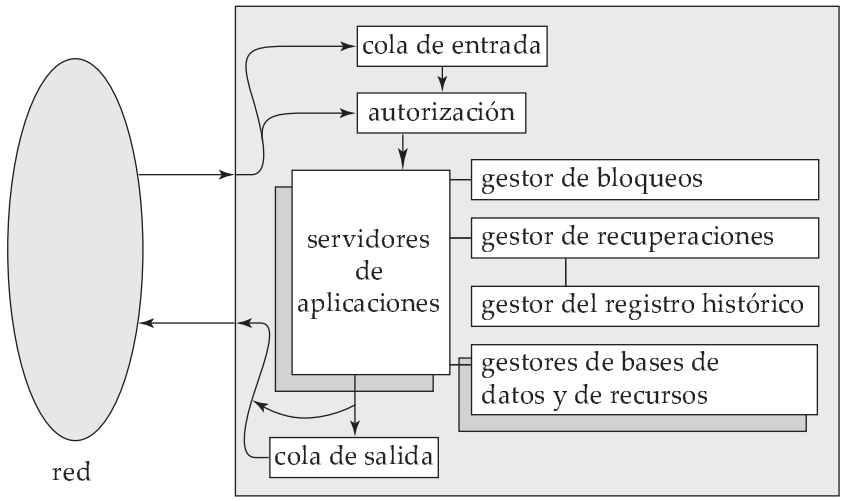


Figura 25.2 Componentes de los monitores TP.

- Estos sistemas no están adecuados a las bases de datos paralelas o distribuidas, ya que un proceso servidor no puede ejecutarse simultáneamente en varios servidores (sin embargo, las hebras concurrentes de un proceso pueden soportarse en un sistema multiprocesador de memoria compartida). Se trata de un inconveniente serio para las organizaciones de gran tamaño en las que el procesamiento paralelo resulta fundamental para el tratamiento de grandes cargas de trabajo, y los datos distribuidos son cada vez más frecuentes.

Una manera de resolver estos problemas es ejecutar varios procesos del servidor de aplicaciones que tengan acceso a una base de datos común y dejar que los clientes se comuniquen con la aplicación mediante un único proceso de comunicaciones que encamine las solicitudes. Este modelo se denomina **modelo de varios servidores y un solo encaminador**, ilustrado en la Figura 25.1c. Este modelo soporta procesos de servidor independientes para varias aplicaciones; además, cada aplicación puede tener un grupo de procesos de servidor, cualquiera de los cuales puede manejar una sesión cliente. La solicitud puede, por ejemplo, encaminarse al servidor con menor carga de un grupo. Como antes, cada proceso de servidor puede tener, a su vez, varias hebras, de modo que puede atender de manera concurrente varios clientes. Como generalización adicional, los servidores de aplicaciones pueden ejecutarse en sitios diferentes de una base de datos paralela o distribuida y el proceso de comunicaciones puede manejar las comunicaciones entre los procesos.

La arquitectura anterior también se usa mucho en los servidores Web. Un servidor Web tiene un proceso principal que recibe las solicitudes HTTP, y luego asigna la tarea de manejar cada solicitud a un proceso diferente (escogido de entre un grupo de procesos). Cada uno de los procesos tiene, a su vez, varias hebras, por lo que puede atender varias solicitudes. El uso de lenguajes de programación seguros, tales como Java, C#, o Visual Basic, permite que los servidores de aplicaciones Web protejan las hebras frente a errores en otras. En cambio, con un lenguaje como C o C++, los errores como los de asignación de memoria en una hebra pueden provocar el fallo de otras.

Una arquitectura más general tiene varios procesos, en lugar de uno solo, para comunicarse con los clientes. Los procesos de comunicación con los clientes interactúan con uno o varios procesos encaminadores, que encaminan las solicitudes hacia el servidor correspondiente. Los monitores TP de generaciones posteriores, por tanto, tienen una arquitectura diferente, denominada **modelo de varios servidores y varios encaminadores**, ilustrado en la Figura 25.1d. Un proceso controlador inicia los demás procesos y supervisa su funcionamiento. Pathway de Tandem es un ejemplo de un monitor TP que usa esta arquitectura. Los sistemas servidores Web de rendimiento muy elevado también adoptan una arquitectura de este tipo. Los procesos del encaminador son generalmente encaminadores de red que dirigen el tráfico de la misma dirección Internet a distintas computadoras servidoras, dependiendo de dónde venga el tráfico. Lo que parece al mundo exterior un solo servidor con una sola dirección puede ser una colección de servidores.

La estructura detallada de un monitor TP aparece en la Figura 25.2. Un monitor TP hace más cosas que pasar mensajes a los servidores de aplicaciones. Cuando llegan los mensajes, puede que haya que ubicarlos en una cola; por tanto, hay un **gestor de colas** para los mensajes entrantes. Puede que la cola sea una **cola duradera**, cuyas entradas sobreviven a los fallos del sistema. El uso de colas duraderas ayuda a asegurar que se acaben procesando los mensajes una vez recibidos y guardados en la cola, independientemente de los fallos del sistema. La gestión de las autorizaciones y de los servidores de aplicaciones (por ejemplo, el inicio de los servidores y el encaminamiento de los mensajes hacia los servidores) son otras funciones de los monitores TP. Los monitores TP suelen proporcionar recursos para la elaboración de registros históricos, recuperación y control de concurrencia, lo que permite a los servidores de aplicaciones implementar directamente, si fuera necesario, las propiedades ACID de las transacciones.

Finalmente, los monitores TP también proporcionan soporte para la mensajería persistente. Hay que recordar que la mensajería persistente (Apartado 22.4.3) proporciona una garantía de que el mensaje se entregue si (y sólo si) la transacción se compromete.

Además de estos servicios, muchos monitores TP también proporcionaban *recursos para presentaciones* para crear interfaces de menús o de formularios o para los clientes no inteligentes como los terminales; estos recursos ya no son importantes porque los clientes no inteligentes ya no se usan mucho.

25.1.2 Coordinación de las aplicaciones mediante los monitores TP

Hoy en día las aplicaciones suelen tener que interactuar con múltiples bases de datos. Puede que tengan que interactuar con sistemas heredados, como los sistemas de almacenamiento de finalidad especial construidos directamente con base en los sistemas de archivos. Finalmente, puede que tengan que comunicarse con usuarios o con otras aplicaciones en sitios remotos. Por tanto, también tienen que interactuar con subsistemas de comunicaciones. Es importante poder coordinar los accesos a los datos e implementar las propiedades ACID de las propiedades a través de esos sistemas.

Los monitores TP modernos proporcionan soporte para la construcción y la gestión de aplicaciones de un tamaño tan grande, creadas a partir de varios subsistemas como las bases de datos, los sistemas heredados y los sistemas de comunicaciones. Los monitores TP tratan cada subsistema como un **gestor de recursos** que proporciona acceso transaccional a algún conjunto de recursos. La interfaz entre el monitor TP y el gestor de recursos se define mediante un conjunto de primitivas de las transacciones como *begin_transaction* (iniciar transacción), *commit_transaction* (comprometer transacción), *abort_transaction* (abortar transacción) y *prepare_to_commit_transaction* (preparar para comprometer transacción, para el compromiso de dos fases). Por supuesto, el gestor de recursos también debe proporcionar otros servicios, como proporcionar datos, a la aplicación.

La interfaz del gestor de recursos está definida por el estándar de procesamiento de transacciones distribuidas X/Open (X/Open Distributed Transaction Processing). Muchos sistemas de bases de datos soportan los estándares X/Open, y pueden actuar como gestores de recursos. Los monitores TP—así como otros productos, como los sistemas SQL, que soportan los estándares X/Open—pueden conectarse con los gestores de recursos.

Además, los servicios proporcionados por los monitores TP, como la mensajería persistente y las colas duraderas, actúan como gestores de recursos que soportan las transacciones. Los monitores TP pueden actuar como coordinadores de los compromisos de dos fases para las transacciones que tienen acceso a estos servicios y a los sistemas de bases de datos. Por ejemplo, cuando se ejecuta una transacción de actualización encolada, se entrega un mensaje y se elimina la transacción solicitada de la cola de solicitudes. El compromiso de dos fases entre la base de datos y los gestores de recursos para las colas duraderas y para la mensajería persistente ayuda a asegurar que, independientemente de los fallos, pueden producirse todas estas acciones o ninguna de ellas.

También se pueden usar los monitores TP para administrar los sistemas complejos cliente–servidor que consisten en varios servidores y gran número de clientes. El monitor TP coordina las actividades como los puntos de control y los cierres del sistema. Proporciona la seguridad y la autentificación de los clientes. Administra los grupos de servidores añadiendo o eliminando servidores sin ninguna interrupción del sistema de bases de datos. Finalmente, controla el ámbito de los fallos. Si falla algún servidor, el monitor TP puede detectar ese fallo, abortar las transacciones en curso y reiniciarlas. Si falla algún nodo,

el monitor TP puede migrar las transacciones a servidores de otros nodos y, una vez más, cancelar las transacciones incompletas. Cuando los nodos que fallan se reinician, el monitor TP puede gobernar la recuperación de los gestores de recursos del nodo.

Los monitores TP pueden usarse para ocultar fallos de las bases de datos en los sistemas replicados; los sistemas remotos de copia de seguridad (Apartado 17.9) son un ejemplo de sistemas replicados. Las solicitudes de transacciones se remiten al monitor TP, que transfiere los mensajes a una de las réplicas de la base de datos (al sitio principal, en el caso de sistemas remotos de copia de seguridad). Si falla algún sitio, el monitor TP puede encaminar los mensajes de manera transparente hacia un sitio de copia de seguridad, enmascarando el fallo del primer sitio.

En los sistemas cliente–servidor los clientes suelen interactuar con los servidores mediante un mecanismo de **llamada a procedimientos remotos** (Remote Procedure Call, RPC), en el que el cliente realiza la llamada a un procedimiento, que se ejecuta realmente en el servidor, y los resultados se devuelven al cliente. En lo relativo al código cliente que invoca al RPC, la llamada tiene el mismo aspecto que la invocación a un procedimiento local. Los sistemas de monitores TP, como Encina, proporcionan una interfaz para **RPC transaccionales** con sus servicios. En esta interfaz el mecanismo RPC proporciona llamadas que pueden usarse para encerrar una serie de llamadas RPC dentro de una transacción. Por tanto, las actualizaciones llevadas a cabo por el RPC se ejecutan dentro del ámbito de la transacción y se pueden hacer retroceder si hay algún fallo.

25.2 Flujos de trabajo de transacciones

Un **flujo de trabajo** es una actividad en la que varias entidades de procesamiento ejecutan varias tareas de manera coordinada. Una **tarea** define un trabajo que hay que hacer y puede especificarse de varias maneras, incluidos una descripción textual en un archivo o en un mensaje de correo electrónico, un formulario, un mensaje o un programa de computadora. La **entidad de procesamiento** que lleva a cabo las tareas puede ser una persona o un sistema de software (por ejemplo, un sistema de envío de correo electrónico, un programa de aplicación o un sistema gestor de bases de datos).

La Figura 25.3 muestra algunos ejemplos de flujos de trabajo. Un ejemplo de sencillo es el de un sistema de correo electrónico. La entrega de un solo mensaje de correo implica varios sistemas de envío de correo que reciben y transmiten el mensaje de correo, hasta que el mensaje alcance su destino, donde se almacena. Cada sistema de envío de correo lleva a cabo una tarea—transmitir el mensaje al siguiente sistema de envío de correo—y puede ser necesaria la tarea de varios sistemas de envío de correo para encaminar el mensaje desde su origen hasta su destino. Otros términos empleados en la literatura de bases de datos y similares para hacer referencia a los flujos de trabajo son **flujo de tareas** y **aplicaciones multisistema**. Las tareas del flujo de trabajo a veces se denominan **pasos**.

En general, los flujos de trabajo pueden implicar a una o varias personas. Por ejemplo, considérese el procesamiento de un préstamo. El flujo de trabajo correspondiente aparece en la Figura 25.4. La persona que desea un préstamo rellena un formulario, que es revisado por el encargado de los préstamos. Un empleado que procesa las solicitudes de préstamos comprueba los datos del formulario, usando fuentes como las sucursals de referencia de préstamo. Cuando se ha reunido toda la información solicitada, el encargado de los préstamos puede que decida conceder el préstamo; puede que esa decisión tenga que ser aprobada por uno o más empleados de rango superior, después de lo cual se podrá conceder el prés-

Aplicación de flujo de trabajo	Tarea habitual	Entidad típica de procesamiento
encaminamiento de correo electrónico	mensaje de correo electrónico	sistemas de envío de correo electrónico
procesamiento de préstamos	procesamiento de formularios	personas, software de aplicaciones
procesamiento de pedidos de compra	procesamiento de formularios	personas, software de aplicaciones, SGBD

Figura 25.3 Ejemplos de flujos de trabajo.

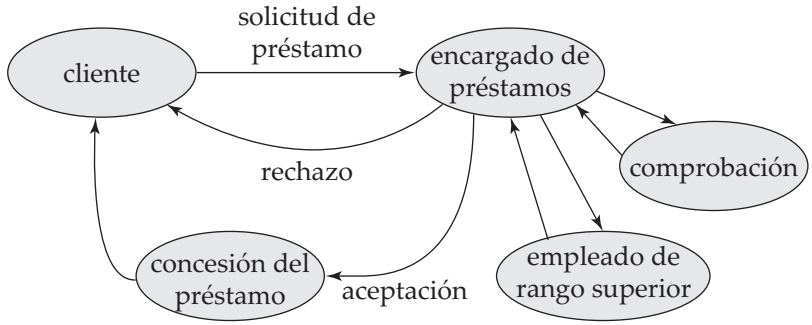


Figura 25.4 Flujo de trabajo en el procesamiento de préstamos.

tamo. Cada persona de este flujo de trabajo realiza una tarea; en un banco que no tenga automatizada la tarea de procesamiento de los préstamos, la coordinación de las tareas suele ejecutarse pasando la solicitud del préstamo con notas y otra información adjuntas de un empleado al siguiente. Otros ejemplos de flujos de trabajo son el procesamiento de notas de gastos, de pedidos de compra y de transacciones de tarjetas de préstamo.

Hoy en día es más probable que toda la información relativa a un flujo de trabajo se almacene en forma digital en una o más computadoras y, con el auge de las redes, la información puede transferirse con facilidad de una computadora a otra. Por tanto, es viable que las organizaciones automaticen sus flujos de trabajo. Por ejemplo, para automatizar las tareas implicadas en el procesamiento de los préstamos, se puede almacenar la solicitud de préstamo y la información asociada en una base de datos. El propio flujo de trabajo implica, entonces, la transferencia de la responsabilidad de una persona a la siguiente y, posiblemente, incluso a programas que pueden capturar de manera automática la información necesaria. Las personas implicadas pueden coordinar sus actividades mediante el correo electrónico.

Los flujos de trabajo están llegando a ser cada vez más importantes por muchas razones en y entre las organizaciones. Muchas organizaciones tienen actualmente múltiples sistemas de software que necesitan trabajar juntos. Por ejemplo, cuando un empleado entra a trabajar en una organización, su información puede tener que ser proporcionada al sistema de nóminas, al sistema de bibliotecas, a los sistemas de autentificación que permiten que el usuario inicie sesión en las computadoras, a las cuentas de un sistema que gestione las cuentas de la cafetería, etc. Las actualizaciones, como cuando el empleado cambia su estado o deja la organización, también tienen que ser propagadas a todos los sistemas.

Las organizaciones están automatizando cada vez más sus servicios; por ejemplo, un proveedor puede proporcionar un sistema automatizado para que los clientes realicen sus pedidos. Cuando se realiza un pedido es posible que se deban llevar a cabo varias tareas, como reservar tiempo de producción para crear el producto pedido y tiempo del servicio de entrega para entregar el producto.

Hay que abordar dos actividades, en general, para automatizar un flujo de trabajo. La primera es la **especificación del flujo de trabajo**: detallar las tareas que hay que ejecutar y definir los requisitos de la ejecución. El segundo problema es la **ejecución del flujo de trabajo**, que hay que llevar a cabo mientras se proporcionan las salvaguardas de los sistemas tradicionales de bases de datos relativas a corrección de los cálculos e integridad y durabilidad de los datos. Por ejemplo, no resulta aceptable que se pierda una solicitud de préstamo o una nota, ni que se procese más de una vez, debido a un fallo del sistema. La idea subyacente a los flujos de trabajo transaccionales es usar y ampliar los conceptos de las transacciones al contexto de los flujos de trabajo.

Las dos actividades se complican por el hecho de que muchas organizaciones usan varios sistemas de procesamiento de la información administrados de manera independiente que, en la mayor parte de los casos, se desarrollaron por separado para automatizar funciones diferentes. Puede que las actividades del flujo de trabajo exijan interacciones entre varios de esos sistemas, cada uno de las cuales lleva a cabo una tarea, así como interacciones con las personas.

En los últimos años se han desarrollado varios sistemas de flujo de trabajo. Aquí se estudiarán las propiedades de los sistemas de flujo de trabajo en un nivel relativamente abstracto, sin descender a los detalles de ningún sistema concreto.

25.2.1 Especificación de los flujos de trabajo

No hace falta modelar los aspectos internos de cada tarea con vistas a la especificación y gestión de un flujo de trabajo. En una vista abstracta de las tareas, cada tarea puede usar los parámetros almacenados en sus variables de entrada, recuperar y actualizar los datos del sistema local, almacenar los resultados en sus variables de salida y se la puede consultar sobre su estado de ejecución. En cualquier momento de la ejecución el **estado del flujo de trabajo** consiste en el conjunto de estados de las tareas constituyentes del flujo de trabajo, y los estados (valores) de todas las variables de la especificación del flujo de trabajo.

La coordinación de las tareas puede especificarse de manera estadística o dinámica. La especificación estática define las tareas—y las dependencias entre ellas—antes de que comience la ejecución del flujo de trabajo. Por ejemplo, las tareas del flujo de trabajo de las notas de gastos pueden consistir en la aprobación de las notas por una secretaria, un gestor y un contable, en ese orden, y, finalmente, en la entrega de un cheque. Las dependencias entre las tareas puede ser sencilla—hay que completar cada tarea antes de que comience la siguiente.

Una generalización de esta estrategia es la imposición de una condición previa a la ejecución de cada tarea del flujo de trabajo, de modo que todas las tareas posibles del flujo de trabajo y sus dependencias se conozcan por anticipado, pero que sólo se ejecuten aquellas tareas cuyas condiciones previas se satisfagan. Las condiciones previas pueden definirse mediante dependencias como las siguientes:

- **El estado de ejecución** de otras tareas—por ejemplo, "la tarea t_i no puede comenzar hasta que la tarea t_j haya finalizado", o "la tarea t_i debe abortarse si la tarea t_j se ha comprometido".
- **Los resultados** de otras tareas—por ejemplo, "la tarea t_i puede comenzar si la tarea t_j devuelve un valor mayor que veinticinco", o "la tarea de aprobación por el gestor puede comenzar si la tarea de aprobación por la secretaria devuelve el resultado 'Aceptar'".
- **Las variables externas** modificadas por los eventos externos—por ejemplo, "la tarea t_i no puede iniciarse antes de las nueve de la mañana", o "la tarea t_i debe iniciarse antes de que transcurran veinticuatro horas desde la finalización de la tarea t_j".

Las dependencias pueden combinarse mediante los conectores lógicos (**or**, **and**, **not**) para formar condiciones previas complejas de planificación.

Un ejemplo de la planificación dinámica de las tareas son los sistemas de encaminamiento del correo electrónico. La tarea que hay que programar a continuación para cada mensaje de correo depende de la dirección de destino de ese mensaje y de los encaminadores intermedios se hallan en funcionamiento.

25.2.2 Requisitos de atomicidad ante fallos de los flujos de trabajo

El diseñador del flujo de trabajo puede especificar sus requisitos de **atomicidad ante fallos** de acuerdo con su semántica. El concepto tradicional de atomicidad ante fallos exige que el fallo de cualquier tarea de lugar al fallo del flujo de trabajo. Sin embargo, un flujo de trabajo puede, en muchos casos, sobrevivir al fallo de una de sus tareas—por ejemplo, ejecutando una tarea funcionalmente equivalente en otro sitio. Por tanto, se debe permitir al diseñador que defina los requisitos de atomicidad ante fallos del flujo de trabajo. El sistema debe garantizar que cada ejecución de un flujo de trabajo termine en un estado que satisfaga los requisitos de atomicidad ante fallos definidos por el diseñador. Esos estados se denominan **estados de terminación aceptables** del flujo de trabajo. Todos los demás estados del flujo de trabajo constituyen un conjunto de **estados de terminación no aceptables**, en los que puede que se violen los requisitos de atomicidad de los fallos.

Los estados aceptables de terminación pueden declararse comprometidos o abortados. Un **estado aceptable de terminación comprometido** es un estado de ejecución en el que los objetivos del flujo de trabajo se han conseguido. Por el contrario, un **estado aceptable de terminación abortado** es un estado válido de terminación en el que el flujo de trabajo no ha logrado alcanzar sus objetivos. Si se ha alcanzado un estado aceptable de terminación abortado hay que deshacer todos los efectos indeseables de la ejecución parcial del flujo de trabajo de acuerdo con los requisitos de atomicidad ante fallos del flujo de trabajo.

El flujo de trabajo debe alcanzar un estado aceptable de terminación *incluso en caso de fallo del sistema*. Por tanto, si el flujo de trabajo se hallaba en un estado no aceptable de terminación en el momento del fallo, durante la recuperación del sistema hay que llevarlo a un estado aceptable de terminación (bien sea abortado, bien comprometido).

Por ejemplo, en el flujo de trabajo del procesamiento de los préstamos, en el estado final, o bien se comunica al solicitante del préstamo que no se le puede conceder, o se le abona el importe solicitado. En caso de fallo como puede ser un fallo de larga duración del sistema de verificación, puede devolverse la solicitud de préstamo al solicitante con una explicación adecuada; este resultado constituiría una terminación abortada aceptable. Una terminación comprometida aceptable sería la aceptación o el rechazo de la solicitud.

En general, las tareas pueden comprometer y liberar sus recursos antes de que el flujo de trabajo alcance un estado de terminación. Sin embargo, si la transacción multitarea aborta posteriormente, su atomicidad ante fallos puede que exija que se deshagan todos los efectos de las tareas ya completadas (por ejemplo, las subtransacciones comprometidas) ejecutando tareas compensadoras (como las subtransacciones). La semántica de la compensación exige que la transacción compensadora acabe completando su ejecución con éxito, quizás tras varios reenvíos.

En el flujo de trabajo del procesamiento de las notas de gastos, por ejemplo, puede que se reduzca el importe del presupuesto del departamento debido a la aprobación inicial de una nota de gastos por el gestor. Si posteriormente se rechaza esa nota, debido a un fallo o por otro motivo, puede que haya que restaurar el presupuesto mediante una transacción compensadora.

25.2.3 Ejecución de los flujos de trabajo

La ejecución de las tareas puede controlarla un coordinador humano o un sistema de software denominado **sistema gestor de flujos de trabajo**. Los sistemas gestores de flujos de trabajo consisten en un planificador, los agentes para las tareas y un mecanismo para consultar el estado del sistema del flujo de trabajo. Cada agente de tarea controla la ejecución de una tarea por una entidad de procesamiento. El planificador es un programa que procesa los flujos de trabajo remitiendo diferentes tareas para su ejecución, controlando los diferentes eventos y evaluando las condiciones relativas a las dependencias entre las tareas. El planificador puede remitir una tarea para su ejecución (a un agente de tareas) o solicitar que se aborte una tarea previamente remitida. En el caso de las transacciones con múltiples bases de datos, las tareas son subtransacciones y las entidades de procesamiento son sistemas gestores de bases de datos locales. De acuerdo con las especificaciones del flujo de trabajo, el planificador hace que se cumplan las dependencias de planificación y es responsable de asegurar que las tareas alcancen estados aceptables de terminación.

Hay tres alternativas de desarrollo en la arquitectura de los sistemas gestores de flujos de trabajo. La **arquitectura centralizada** tiene un solo planificador que programa las tareas de todos los flujos de trabajo que se ejecutan de manera concurrente. La **arquitectura parcialmente distribuida** tiene un planificador para cada flujo de trabajo. Cuando los problemas de la ejecución concurrente pueden separarse de la función de planificación, esta opción es una elección natural. La **arquitectura completamente distribuida** no tiene planificador, pero los agentes de tareas coordinan su ejecución comunicándose entre sí para satisfacer las dependencias entre las tareas y otros requisitos de ejecución del flujo de trabajo.

Los sistemas de ejecución de flujos de trabajo siguen el enfoque totalmente distribuido que se acaba de describir y están basados en la mensajería. La mensajería puede implementarse mediante mecanismos de mensajería persistente. Algunas implementaciones usan el correo electrónico para la mensajería; estas implementaciones proporcionan muchas de las características de la mensajería persistente, pero generalmente no garantizan la atomicidad de la entrega de los mensajes y el compromiso de las transacciones. Cada sitio tiene un agente de tareas que ejecuta las tareas recibidas mediante los mensajes. Puede que la ejecución también implique la entrega de mensajes a personas, que tienen que llevar a cabo alguna acción. Cuando se completa una tarea en un sitio y hay que procesarla en otro sitio, el agente de tareas transmite un mensaje al sitio siguiente. El mensaje contiene toda la información relevante sobre la tarea que hay que realizar. Estos sistemas de flujos de trabajo basados en mensajes resultan especial-

mente útiles en las redes que se pueden desconectar durante parte del tiempo, como las redes de acceso telefónico.

El enfoque centralizado se usa en sistemas de flujos de trabajo en que los datos se almacenan en una base de datos central. El planificador notifica a los diferentes agentes, como pueden ser las personas o los programas informáticos, que hay que llevar a cabo una tarea y realiza un seguimiento de su finalización. Resulta más sencillo realizar un seguimiento del estado del flujo de trabajo con el enfoque centralizado que con el enfoque completamente distribuido.

El planificador debe garantizar que termine el flujo de trabajo en uno de los estados aceptables de terminación especificados. Idealmente, antes de intentar ejecutar un flujo de trabajo, el planificador debe examinarlo para comprobar si puede terminar en un estado no aceptable. Si el planificador no puede garantizar que el flujo de trabajo termine en un estado aceptable, debe rechazar esas especificaciones sin intentar ejecutar el flujo de trabajo. Por ejemplo, considérese un flujo de trabajo consistente en dos tareas representadas por las subtransacciones S_1 y S_2, con los requisitos de atomicidad ante fallos que indican que se deben comprometer las dos subtransacciones o ninguna de ellas. Si S_1 y S_2 no proporcionan estados preparados para comprometerse (para un compromiso de dos fases) y, además, no tienen transacciones compensadoras, es posible alcanzar un estado en que se comprometa una subtransacción y se aborte la otra, y no haya manera de llevar a las dos al mismo estado. Por tanto, esa especificación del flujo de trabajo es **insegura**, y debe rechazarse.

Los controles de seguridad como el que se acaba de describir pueden ser imposibles o poco prácticos de implementar en el planificador; pasa a ser, entonces, responsabilidad de la persona que diseña la especificación del flujo de trabajo asegurarse de que el flujo de trabajo sea seguro.

25.2.4 Recuperación en los flujos de trabajo

El objetivo de la **recuperación en los flujos de trabajo** es hacer que se cumpla la atomicidad ante fallos de los flujos de trabajo. Los procedimientos de recuperación deben asegurarse de que, si se produce un fallo en cualquiera de los componentes de procesamiento del flujo de trabajo (incluido el planificador), éste acabe alcanzando un estado aceptable de terminación (sea abortado o comprometido). Por ejemplo, el planificador puede continuar procesando tras el fallo y la recuperación, como si no hubiera pasado nada, lo que proporciona recuperabilidad hacia delante. En caso contrario, el planificador puede abortar todo el flujo de trabajo (es decir, alcanzar uno de los estados globales abortados). De cualquier forma, puede que haga falta comprometer algunas subtransacciones o incluso remitirlas para su ejecución (por ejemplo, las subtransacciones compensadoras).

Se da por supuesto que las entidades de procesamiento implicadas en el flujo de trabajo tienen sus propios sistemas locales de recuperación y tratan sus fallos locales. Para recuperar el contexto del entorno de ejecución las rutinas de recuperación de los fallos deben restaurar la información de estado del planificador en el momento del fallo, incluida la información sobre el estado de ejecución de cada tarea. Por tanto, la información de estado correspondiente debe registrarse en almacenamiento estable.

También hay que considerar el contenido de las colas de mensajes. Cuando un agente transfiere una tarea a otro, la transferencia debe ejecutarse exactamente una vez: si la transferencia tiene lugar dos veces, puede que se ejecute dos veces una tarea; si no se produce la transferencia, puede que se pierda la tarea. La mensajería persistente (Apartado 22.4.3) proporciona exactamente las características para asegurar una transferencia positiva y única.

25.2.5 Sistemas gestores de flujos de trabajo

Los flujos de trabajo suelen codificarse a mano como parte de los sistemas de aplicaciones. Por ejemplo, los sistemas de planificación de los recursos de las empresas (Enterprise Resource Planning, ERP), que ayudan a coordinar las actividades en toda la empresa, tienen incorporados numerosos flujos de trabajo.

El objetivo de los sistemas gestores de flujos de trabajo es simplificar la construcción de flujos de trabajo y hacerlos más dignos de confianza, permitiéndoles que se especifiquen en un modo de nivel elevado y se ejecuten de acuerdo con la especificación. Hay gran número de sistemas comerciales de gestión de flujos de datos; algunos, como FlowMark de IBM, son sistemas gestores de flujos de trabajo de propósito general, mientras que otros son específicos de flujos de trabajo concretos, como los sistemas de procesamiento de comandos o los sistemas de comunicación de fallos.

En el mundo actual de organizaciones interconectadas, no es suficiente gestionar los flujos de trabajo exclusivamente en el interior de una organización. Los flujos de trabajo que atraviesan las fronteras organizativas se están volviendo cada vez más frecuentes. Por ejemplo, considérese un pedido realizado por una organización y comunicado a otra organización que lo atiende. En cada organización puede que haya un flujo de trabajo asociado con el pedido, y es importante que los flujos de trabajo puedan operar entre sí con objeto de minimizar la intervención humana.

La Coalición de gestión de flujos de trabajo (Workflow Management Coalition) ha desarrollado estándares para la interoperatividad entre sistemas de flujos de trabajo. Los esfuerzos actuales de normalización usan XML como lenguaje subyacente para comunicar la información sobre el flujo de trabajo. Véanse las notas bibliográficas para obtener más información.

25.3 Comercio electrónico

El término comercio electrónico hace referencia al proceso de llevar a cabo varias actividades relacionadas con el comercio por medios electrónicos, principalmente por Internet. Entre los tipos de actividades figuran:

- Actividades previas a la venta, necesarias para informar al posible comprador del producto o servicio que se desea vender.
- El proceso de venta, que incluye las negociaciones sobre el precio y la calidad del servicio, y otros asuntos contractuales.
- El mercado: cuando hay varios vendedores y varios compradores para un mismo producto, un mercado, como la bolsa, ayuda a negociar el precio que se va a pagar por el producto. Las subastas se usan cuando hay un único vendedor y varios compradores, y las subastas inversas se usan cuando hay un solo comprador y varios vendedores.
- El pago de la compra.
- Las actividades relacionadas con la entrega del producto o servicio. Algunos productos y servicios pueden entregarse por Internet; para otros, Internet sólo se usa para facilitar información sobre el envío y para realizar un seguimiento de los envíos de los productos.
- Atención al cliente y servicio postventa.

Las bases de datos se usan ampliamente para soportar estas actividades. Para algunas de las actividades el uso de las bases de datos es directo, pero hay aspectos interesantes de desarrollo de aplicaciones para las demás actividades.

25.3.1 Catálogos electrónicos

Cualquier sitio de comercio electrónico proporciona a los usuarios un catálogo de los productos y servicios que ofrece. Los servicios facilitados por los catálogos electrónicos pueden variar considerablemente.

Como mínimo, un catálogo electrónico debe proporcionar servicios de exploración y de búsqueda para ayudar a los clientes a hallar el producto que buscan. Para ayudar en la exploración conviene que los productos estén organizados en una jerarquía intuitiva, de modo que unas pocas pulsaciones en los hipervínculos puedan llevar a los clientes a los productos en los que estén interesados. Las palabras clave facilitadas por el cliente (por ejemplo, "cámara digital" o "computadora") deben acelerar el proceso de búsqueda de los productos solicitados. Los catálogos electrónicos también deben proporcionar un medio para que los clientes comparen con facilidad las alternativas para elegir entre los diferentes productos.

Los catálogos electrónicos pueden personalizarse para los clientes. Por ejemplo, puede que un vendedor al por menor tenga un acuerdo con una gran empresa para venderle algunos productos con descuento. Un empleado de la empresa, al examinar el catálogo para adquirir productos para la empresa, debería ver los precios con el descuento acordado, en vez de con los precios normales. Debido a las restricciones legales sobre las ventas de algunos tipos de artículos, no se les deberían mostrar a los clientes menores de edad, o de ciertos estados o países, los artículos que no se les pueden vender legalmente.

Los catálogos también pueden personalizarse para usuarios individuales, de acuerdo con su historial de compras. Por ejemplo, se pueden ofrecer a los clientes frecuentes descuentos especiales en algunos productos.

El soporte de esa personalización necesita que la información de los clientes y la información sobre precios o descuentos especiales y sobre restricciones a las ventas se guarde en una base de datos. También hay desafíos en el soporte de tasas de transacciones muy elevadas, que suelen abordarse guardando en la caché los resultados de las consultas o las páginas Web generadas.

25.3.2 Mercados

Cuando hay varios vendedores o varios compradores (o ambas cosas) para un producto, los mercados ayudan a negociar el precio que debe pagarse por el producto. Hay varios tipos diferentes de mercados:

- En los sistemas de **subastas inversas** los compradores manifiestan sus necesidades y los vendedores pujan por proporcionar el artículo. El proveedor que ofrece el precio más bajo gana la subasta. En los sistemas de puja cerrada las pujas no se hacen públicas, mientras que en los sistemas de puja abierta las pujas sí se hacen públicas.
- En las **subastas** hay varios compradores y un solo vendedor. Por simplificar, supóngase que sólo se vende un ejemplar de cada artículo. Los compradores pujan por los artículos que se venden y el que puja más alto por un artículo consigue comprarlo por el precio de la puja.

  Cuando hay varios ejemplares de un artículo las cosas se vuelven más complicadas: supóngase que hay cuatro artículos y puede que un comprador desee tres ejemplares a 10 € cada uno, mientras que otro desea dos copias por 13 € cada una. No es posible satisfacer ambas pujas. Si los artículos carecen de valor si no se venden (por ejemplo, billetes de avión, que deben venderse antes de que despegue el avión), el vendedor simplemente selecciona un conjunto de pujas que maximice los ingresos. En caso contrario, la decisión es más complicada.
- En las **bolsas**, como puede ser una bolsa de valores, hay varios vendedores y varios compradores. Los compradores pueden especificar el precio máximo que desean pagar, mientras que los vendedores especifican el precio mínimo que desean. Suele haber un *creador de mercado* que casa las ofertas de compra y de venta y decide el precio de cada intercambio (por ejemplo, al precio de la oferta de venta).

Existen otros tipos de mercados más complejos.

Entre los aspectos de las bases de datos relacionados con el manejo de los mercados figuran:

- Hay que autentificar a los que realizan las pujas antes de permitirles pujar.
- Las pujas (de compra o de venta) deben registrarse de modo seguro en una base de datos. Hay que comunicar rápidamente las pujas al resto de las personas implicadas en el mercado (como pueden ser todos los compradores o todos los vendedores), que puede que sean numerosas.
- Los retrasos en la difusión de las pujas pueden llevar a pérdidas financieras para algunos participantes.
- Los volúmenes de los intercambios pueden resultar tremendamente grandes en tiempos de volatilidad de las bolsas, o hacia el final de las subastas. Por tanto, se usan para estos sistemas bases de datos de muy alto rendimiento con elevados grados de paralelismo.

25.3.3 Liquidación de pedidos

Una vez seleccionados los artículos (quizás mediante un catálogo electrónico) y determinado el precio (acaso mediante un mercado electrónico), hay que liquidar el pedido. La liquidación implica el pago y la entrega de las mercancías.

Una manera sencilla pero poco segura de pagar electrónicamente consiste en enviar el número de una tarjeta de crédito. Hay dos problemas principales. En primer lugar, es posible el fraude con las tarjetas de crédito. Cuando un comprador paga mercancías físicas las empresas pueden asegurarse de que la

dirección de entrega coincide con la del titular de la tarjeta, de modo que nadie más reciba la mercancía, pero para las mercancías entregadas de manera electrónica no es posible realizar esa comprobación. En segundo lugar, hay que confiar en que el vendedor sólo facture el artículo acordado y en que no pase el número de la tarjeta de crédito a personas no autorizadas que lo puedan usar de manera no adecuada.

Se dispone de varios protocolos para los pagos seguros que evitan los dos problemas mencionados. Además, proporcionan una mayor intimidad, ya que no hay que dar al vendedor datos innecesarios del comprador y no se le facilitan a la compañía de la tarjeta de crédito información innecesaria de los artículos comprados. Toda la información transmitida debe cifrarse de modo que nadie que intercepte los datos por la red pueda averiguar su contenido. El cifrado con claves pública y privada se usa mucho para esta tarea.

Los protocolos también deben evitar los **ataques de personas intermedias**, en los que alguien puede suplantar al banco o a la compañía de la tarjeta de crédito, o incluso al vendedor o al comprador y sustraer información secreta. La suplantación puede realizarse pasando una clave falsa como si fuera la clave pública de otra persona (el banco, la compañía de la tarjeta de crédito, el comerciante o el comprador). La suplantación se evita mediante un sistema de **certificados digitales**, en el que las claves públicas vienen firmadas por una agencia de certificación cuya clave pública es bien conocida (o que, a su vez, tiene su clave pública certificada por otra agencia de certificación, y así hasta llegar a una clave que sea bien conocida). A partir de la clave pública bien conocida el sistema puede autentificar las otras claves comprobando los certificados en una secuencia inversa. Los certificados digitales se describieron anteriormente en el Apartado 8.8.3.3.

El protocolo **transacciones electrónicas seguras** (Secure Electronic Transaction, SET) es uno de los protocolos de pago seguro. El protocolo necesita varias rondas de comunicación entre el comprador, el vendedor y el banco para garantizar la seguridad de la transacción.

También hay sistemas que ofrecen más anonimato, parecido al proporcionado por el dinero físico en efectivo. El sistema de pagos **DigiCash** es uno de esos sistemas. Cuando se realiza un pago en estos sistemas no es posible identificar al comprador. Sin embargo, la identificación del comprador resulta muy sencilla con las tarjetas de crédito e, incluso en el caso de SET, es posible identificar al comprador con la colaboración de la compañía de la tarjeta de crédito o del banco.

25.4 Bases de datos en memoria principal

Para permitir una velocidad elevada de procesamiento de transacciones (centenares o millares de transacciones por segundo) hay que usar hardware de alto rendimiento y aprovechar el paralelismo. Estas técnicas, por sí solas, no obstante, resultan insuficientes para obtener tiempos de respuesta muy bajos, ya que las operaciones de E/S de disco siguen constituyendo un cuello de botella—se necesitan alrededor de diez milisegundos para cada operación de E/S y esta cifra no se ha reducido a una velocidad comparable con el aumento en la velocidad de los procesadores. Las operaciones de E/S suele ser el cuello de botella de las operaciones de lectura y de los compromisos de las transacciones. La elevada latencia de los discos (alrededor de diez milisegundos de promedio) no sólo aumenta el tiempo necesario para tener acceso a un elemento de datos, sino que también limita el número de accesos por segundo.

Se puede hacer un sistema de bases de datos menos ligado a los discos aumentando el tamaño de la memoria intermedia de la base de datos. Los avances en la tecnología de la memoria principal permiten construir memorias principales de gran tamaño con un coste relativamente bajo. Hoy en día los sistemas comerciales de sesenta y cuatro bits pueden soportar memorias principales de decenas de gigabytes.

Para algunas aplicaciones, como el control en tiempo real, es necesario almacenar los datos en la memoria principal para cumplir los requisitos de rendimiento. El tamaño de memoria exigido para la mayoría de estos sistemas no resulta excepcionalmente grande, aunque hay unas cuantas aplicaciones que exigen que sean residentes en la memoria varios gigabytes de datos. Dado que el tamaño de la memoria ha estado creciendo con una velocidad muy elevada, se puede suponer que los datos de cada vez más aplicaciones puedan caber en la memoria principal.

Las memorias principales de gran tamaño permiten el procesamiento más rápido de las transacciones, ya que los datos están residentes en la memoria. No obstante, sigue habiendo limitaciones relacionadas con los discos:

- Hay que guardar en almacenamiento estable los registros del registro histórico antes de comprometer una transacción. El rendimiento mejorado que hace posible la memoria principal de gran tamaño puede hacer que el proceso de registro se convierta en un cuello de botella. Se puede reducir el tiempo de compromiso creando un búfer de registro estable en la memoria principal, usando RAM no volátil (implementada, por ejemplo, mediante memoria sustentada por baterías). La sobrecarga impuesta por el registro también puede reducirse mediante la técnica de *compromiso en grupo* estudiada más adelante en este apartado. La productividad (el número de transacciones por segundo) sigue estando limitada por la velocidad de transferencia de datos del disco de registro.
- Sigue habiendo que escribir los bloques de la memoria intermedia marcados como modificados por las transacciones comprometidas para que se reduzca la cantidad de registro histórico que hay que volver a ejecutar en el momento de la recuperación. Si la velocidad de actualización es extremadamente elevada, la velocidad de transferencia de los datos al disco puede convertirse en un cuello de botella.
- Si el sistema falla, se pierde toda la memoria principal. En la recuperación el sistema tiene la memoria intermedia de la base de datos vacía y hay que introducir desde el disco los elementos de datos cuando se tenga acceso a ellos. Por tanto, incluso una vez que esté completa la recuperación hace falta algo de tiempo antes de que se cargue completamente la base de datos en memoria principal y se pueda reanudar el procesamiento de transacciones de alta velocidad.

Por otro lado, las bases de datos en memoria principal ofrecen oportunidades para la optimización:

- Como la memoria resulta más costosa que el espacio de disco, hay que diseñar las estructuras internas de los datos de la memoria principal para reducir los requisitos de espacio. No obstante, las estructuras de datos pueden tener punteros que atraviesen varias páginas a diferencia de los de las bases de datos en disco, en las que el coste de que la operación de E/S atraviese varias páginas resultaría excesivamente elevado. Por ejemplo, las estructuras arbóreas de las bases de datos en memoria principal pueden ser relativamente profundas, a diferencia de los árboles B^+, pero deben minimizar los requisitos de espacio.
- No hace falta clavar en la memoria las páginas de la memoria intermedia antes de que se tenga acceso a los datos, ya que las páginas de la memoria intermedia no se sustituyen nunca.
- Las técnicas de procesamiento de consultas deben diseñarse para minimizar la sobrecarga de espacio, de modo que no se superen los límites de la memoria mientras se evalúa una consulta; esa situación daría lugar a que se paginara el área de intercambio y ralentizaría el procesamiento de la consulta.
- Una vez eliminado el cuello de botella de las operaciones de E/S del disco, pueden convertirse en cuellos de botella operaciones como los bloqueos y los pestillos. Hay que eliminar estos cuellos de botella mediante mejoras en la implementación de estas operaciones.
- Los algoritmos de recuperación pueden optimizarse, ya que rara vez hace falta borrar las páginas para hacer sitio a otras páginas.

TimesTen y DataBlitz son dos productos de bases de datos en memoria principal que aprovechan varias de estas optimizaciones, mientras que la base de datos de Oracle ha añadido características especiales para soportar memorias principales de tamaño muy grande. Se da información adicional sobre las bases de datos en memoria principal en las referencias de las notas bibliográficas.

El proceso de comprometer una transacción T exige que estos registros se escriban en almacenamiento estable:

- Todos los registros del registro histórico asociados con T que no se hayan remitidos al almacenamiento estable
- El registro <**comprometer** T> del registro histórico

Estas operaciones de salida suelen exigir la salida de bloques que sólo se hallan parcialmente llenos. Para asegurarse de que se saquen bloques casi llenos se usa la técnica de **compromiso en grupo**. En lugar de intentar comprometer T cuando se complete T, el sistema espera hasta que se hayan completado varias transacciones, o hasta que haya pasado un determinado periodo de tiempo desde que se completó la ejecución de una transacción. Luego compromete el grupo de transacciones que están esperando, todas juntas. Los bloques escritos en el registro histórico en almacenamiento estable contienen registros de varias transacciones. Mediante una cuidadosa selección del tamaño del grupo y del tiempo máximo de espera, el sistema puede asegurarse de que los bloques estén llenos cuando se escriben en el almacenamiento estable sin hacer que las transacciones esperen demasiado. Esta técnica da como resultado, en promedio, menos operaciones de salida por cada transacción comprometida.

Aunque el compromiso en grupo reduce la sobrecarga impuesta por el registro histórico, da lugar a un ligero retraso en el compromiso de las transacciones que llevan a cabo actualizaciones. El retraso puede hacerse bastante pequeño (del orden de diez milisegundos), lo que resulta aceptable para muchas aplicaciones. Estos retrasos pueden eliminarse si los discos o los controladores de disco soportan los búferes de RAM no volátil para las operaciones de escritura. Las transacciones pueden comprometerse en cuanto la operación de escritura se lleva a cabo en la memoria intermedia de RAM no volátil. En este caso, no hay necesidad de compromiso en grupo.

Obsérvese que el compromiso en grupo resulta útil incluso en bases de datos con datos residentes en disco.

25.5 Sistemas de transacciones de tiempo real

Las restricciones de integridad que se han considerado hasta ahora corresponden a los valores almacenados en la base de datos. En determinadas aplicaciones las restricciones incluyen **tiempos límite** en los que se tiene que haber completado una tarea. Entre estas aplicaciones están la gestión de factorías, el control del tráfico y la planificación. Cuando se incluyen tiempos límite, la corrección de la ejecución ya no es exclusivamente un problema de consistencia de la base de datos. Por el contrario, hay que preocuparse por el número de tiempos límite sobrepasados y por el tiempo que hace que se sobrepasaron. Los tiempos límite se caracterizan de la manera siguiente:

- **Tiempo límite estricto**. Pueden producirse problemas graves, como fallos del sistema, si no se completa una tarea antes de su tiempo límite.
- **Tiempo límite firme**. La tarea no tiene ningún valor si se completa después del tiempo límite.
- **Tiempo límite flexible**. La tarea tiene un valor decreciente si se completa tras el tiempo límite, y el valor se aproxima a cero a medida que aumenta el retraso.

Los sistemas con tiempos límite se denominan **sistemas de tiempo real**.

La gestión de transacciones en los sistemas de tiempo real debe tener en cuenta los tiempos límite. Si el protocolo de control de concurrencia determina que la transacción T_i debe esperar, puede hacer que T_i supere el tiempo límite. En esos casos, puede que resulte preferible adelantar la transacción que mantiene el bloqueo y permitir que T_i siga adelante. El adelantamiento debe usarse con cuidado, no obstante, ya que el tiempo perdido por la transacción adelantada (debido al retroceso y al renicio) puede hacer que la transacción supere su tiempo límite. Por desgracia, es difícil determinar si es preferible retroceder o esperar en una situación dada.

Una de las principales dificultades para soportar las restricciones de tiempo real surge de la variabilidad en el tiempo de ejecución de las transacciones. En el caso más favorable, todos los accesos a los datos hacen referencia a datos de la memoria intermedia de la base de datos. En el peor de los casos, cada acceso hace que se escriba una página de la memoria intermedia en el disco (precedida de los registros del registro histórico necesario), seguido de la lectura desde el disco de la página que contiene los datos a los que hay que tener acceso. Como los dos o más accesos al disco necesarios en el peor de los casos tardan varios órdenes de magnitud más que las referencias a la memoria principal necesarias en el caso más favorable, el tiempo de ejecución de las transacciones puede estimarse con muy poca precisión si

los datos están residentes en el disco. Por tanto, se suelen usar las bases de datos en memoria principal si hay que cumplir restricciones de tiempo real.

Sin embargo, aunque los datos estén residentes en la memoria principal, la variabilidad del tiempo de ejecución surge de las esperas de los bloqueos, de los abortos de las transacciones, etc. Los investigadores han dedicado esfuerzos considerables al control de concurrencia para las bases de datos de tiempo real. Han ampliado los protocolos de bloqueo para conceder una prioridad más elevada a las transacciones con tiempos límite más próximos. Han determinado que los protocolos de concurrencia optimistas tienen un buen comportamiento en las bases de datos de tiempo real; es decir, estos protocolos dan lugar a menos tiempos límite sobrepasados incluso que los protocolos de bloqueo ampliados. Las notas bibliográficas proporcionan referencias para la investigación en el área de las bases de datos de tiempo real.

En los sistemas de tiempo real, los tiempos límite, y no la velocidad absoluta, son el aspecto más importante. El diseño de sistemas de tiempo real implica asegurarse de que hay suficiente capacidad de procesamiento como para respetar las tiempos límite sin necesitar excesivos recursos de hardware. La consecución de este objetivo, pese a la variabilidad de los tiempos de ejecución resultante de la gestión de las transacciones, sigue constituyendo un problema sin resolver.

25.6 Transacciones de larga duración

El concepto de transacción se desarrolló inicialmente en el contexto de las aplicaciones de procesamiento de datos, en el que la mayor parte de las transacciones son de corta duración y no interactivas. Aunque las técnicas presentadas aquí y, anteriormente, en los Capítulos 15, 16 y 17 funcionan bien en esas aplicaciones, surgen problemas graves cuando se aplica este concepto a sistemas de bases de datos que implican la interacción con personas. Esas transacciones tienen las siguientes propiedades principales:

- **Larga duración**. Una vez que una persona interactúa con una transacción activa esa transacción se transforma en una **transacción de larga duración** desde la perspectiva de la computadora, ya que el tiempo de respuesta de las personas es lento en comparación con la velocidad de las computadoras. Además, en las aplicaciones de diseño, la actividad humana puede suponer horas, días o periodos incluso más prolongados. Por tanto, las transacciones pueden ser de larga duración en términos humanos, además de serlo en términos de la máquina.
- **Exposición de datos no comprometidos**. Los datos generados y mostrados a los usuarios por las transacciones de larga duración no están comprometidos, ya que la transacción puede abortarse. Por tanto, los usuarios—y, en consecuencia, las demás transacciones—pueden verse forzados a leer datos no comprometidos. Si varios usuarios están colaborando en un proyecto puede que las transacciones de los usuarios necesiten intercambiar datos antes de comprometer las transacciones.
- **Subtareas**. Cada transacción interactiva puede consistir en un conjunto de subtareas iniciadas por el usuario. Puede que el usuario desee abortar una subtarea sin hacer necesariamente que aborte toda la transacción.
- **Recuperabilidad**. Resulta inaceptable abortar una transacción interactiva de larga duración debido a un fallo del sistema. La transacción activa debe recuperarse hasta un estado que existiera poco antes del fallo para que se pierda una cantidad de trabajo humano relativamente pequeña.
- **Rendimiento**. El buen rendimiento de los sistemas interactivos de transacciones se define como tiempo de respuesta rápido. Esta definición difiere de la de los sistemas no interactivos, en los que el objetivo es una productividad (número de transacciones por segundo) elevada. Los sistemas con productividad elevada hacen un uso eficiente de los recursos del sistema. Sin embargo, en el caso de las transacciones interactivas, el recurso más costoso es el usuario. Si hay que optimizar la eficiencia y la satisfacción del usuario, el tiempo de respuesta debe ser rápido (desde el punto de vista humano). En los casos en los que una tarea tarde mucho tiempo, el tiempo de respuesta debe ser predecible (es decir, la variabilidad de los tiempos de respuesta debe ser baja), de modo que los usuarios puedan administrar bien su tiempo.

En los Apartados 25.6.1 a 25.6.5 se verá el motivo de que estas cinco propiedades sean incompatibles con las técnicas presentadas hasta ahora, y se estudiará el modo en que se pueden modificar dichas técnicas para acomodar las transacciones interactivas de larga duración.

25.6.1 Ejecuciones no secuenciables

Las propiedades que se han estudiado hacen poco práctico obligar a que se cumpla el requisito empleado en los capítulos anteriores de que sólo se permitan las planificaciones secuenciables. Cada uno de los protocolos de control de concurrencia del Capítulo 16 tiene efectos negativos sobre las transacciones de larga duración:

- **Bloqueo de dos fases**. Cuando no se puede conceder un bloqueo, la transacción que lo ha solicitado se ve obligada a esperar a que se desbloquee el elemento de datos en cuestión. La duración de la espera es proporcional a la duración de la transacción que sostiene el bloqueo. Si el elemento de datos está bloqueado por una transacción de corta duración, se espera que el tiempo de espera sea breve (excepto en el caso de interbloqueos o de carga extraordinaria del sistema). Sin embargo, si el elemento de datos está bloqueado por una transacción de larga duración, la espera será prolongada. Los tiempos de espera elevados provocan tiempos de respuesta mayores y una mayor posibilidad de interbloqueos.

- **Protocolos basados en grafos**. Los protocolos basados en grafos permiten que se liberen los bloqueos antes que con los protocolos de bloqueo de dos fases, y evitan los interbloqueos. Sin embargo, imponen una ordenación de los elementos de datos. Las transacciones deben bloquear los elementos de datos de manera consistente con esta ordenación. En consecuencia, puede que una transacción tenga que bloquear más datos de los que necesita. Además, la transacción debe mantener el bloqueo hasta que no haya posibilidades de que se vuelva a necesitar. Por tanto, es probable que se produzcan esperas por bloqueos de larga duración.

- **Protocolos basados en marcas temporales**. Los protocolos de marcas temporales nunca necesitan que las transacciones esperen. Sin embargo, exigen que las transacciones se aborten bajo ciertas circunstancias. Si se aborta una transacción de larga duración, se pierde una cantidad sustancial de trabajo. Para las transacciones no interactivas este trabajo perdido supone un problema de rendimiento. Para las transacciones interactivas el problema también es de satisfacción de los usuarios. Resulta muy poco deseable que el usuario descubra que se han deshecho varias horas de trabajo.

- **Protocolos de validación**. Al igual que los protocolos basados en las marcas temporales, los protocolos de validación hacen que se cumpla la secuencialidad mediante el aborto de transacciones.

Por tanto, parece que el cumplimiento de la secuencialidad provoca esperas de larga duración, el aborto de transacciones de larga duración o ambas cosas. Hay resultados teóricos, citados en las notas bibliográficas, que sustentan esta conclusión.

Cuando se consideran los problemas de las recuperaciones surgen dificultades adicionales con el cumplimiento de la secuencialidad. Ya se ha estudiado el problema de los retrocesos en cascada, en los que el aborto de una transacción puede conducir al aborto de otras transacciones. Este fenómeno no es deseable, especialmente para las transacciones de larga duración. Si se usa el bloqueo, se deben mantener bloqueos exclusivos hasta el final de la transacción, si hay que evitar el retroceso en cascada. Este mantenimiento de bloqueos exclusivos, no obstante, aumenta la duración del tiempo de espera de las transacciones.

Por tanto, parece que el cumplimiento de la atomicidad de las transacciones debe llevar a una mayor probabilidad de las esperas de larga duración o a crear la posibilidad de retrocesos en cascada.

Estas consideraciones son la base de los conceptos alternativos de corrección de las ejecuciones concurrentes y de la recuperación de transacciones que se considerarán en el resto de este apartado.

25.6.2 Control de concurrencia

El objetivo fundamental del control de concurrencia de las bases de datos es asegurarse de que la ejecución concurrente de las transacciones no da lugar a una pérdida de la consistencia de la base de datos. El concepto de secuencialidad puede usarse para conseguir este objetivo, ya que todas las planificaciones secuenciables conservan la consistencia de las bases de datos. No obstante, no todas las planificaciones que conservan la consistencia de las bases de datos son secuenciables. Por ejemplo, considérese nuevamente una base de datos bancaria que consista en dos cuentas, A y B, con el requisito de consistencia de que se conserve la suma $A + B$. Aunque la planificación de la Figura 25.5 no es secuenciable para conflictos, pese a todo, conserva la suma de $A + B$. También ilustra dos aspectos importantes del concepto de corrección sin secuencialidad.

- La corrección depende de las restricciones de consistencia concretas de la base de datos.
- La corrección depende de las propiedades de las operaciones llevadas a cabo por cada transacción.

En general, no es posible llevar a cabo un análisis automático de las operaciones de bajo nivel de las transacciones y comprobar su efecto en las restricciones de consistencia de la base de datos. Sin embargo, hay técnicas más sencillas. Una de ellas es el uso de las restricciones de consistencia de la base de datos como base de una división de la base de datos en subbases de datos en las que se puede administrar por separado la concurrencia. Otra es el intento de tratar algunas operaciones aparte de **leer** y de **escribir** como operaciones fundamentales de bajo nivel y ampliar el control de concurrencia para trabajar con ellas.

Las notas bibliográficas hacen referencia a otras técnicas para asegurar la consistencia sin exigir secuencialidad. Muchas de estas técnicas aprovechan variedades del control de concurrencia multiversión (véase el Apartado 17.5). A las aplicaciones de procesamiento de datos más antiguas que sólo necesitan una versión los protocolos multiversión les imponen una elevada sobrecarga de espacio para almacenar las versiones adicionales. Dado que muchas de las nuevas aplicaciones de bases de datos exigen el mantenimiento de las versiones de los datos, las técnicas de control de concurrencia que aprovechan varias versiones resultan prácticas.

25.6.3 Transacciones anidadas y multinivel

Las transacciones de larga duración pueden considerarse como conjuntos de subtareas relacionadas o subtransacciones. Al estructurar cada transacción como un conjunto de subtransacciones, se puede mejorar el paralelismo, ya que puede que sea posible ejecutar en paralelo varias subtransacciones. Además, es posible trabajar con los fallos de las subtransacciones (debidos a abortos, fallos del sistema, etc.) sin tener que hacer retroceder toda la transacción de larga duración.

T_1	T_2
leer(A)	
$A := A - 50$	
escribir(A)	
	leer(B)
	$B := B - 10$
	escribir(B)
leer(B)	
$B := B + 50$	
escribir(B)	
	leer(A)
	$A := A + 10$
	escribir(A)

Figura 25.5 Planificación no secuenciable en cuanto a conflictos.

Una transacción anidada o multinivel T consiste en un conjunto $T = \{t_1, t_2, \ldots, t_n\}$ de subtransacciones y en un orden parcial P sobre T. Cada subtransacción t_i de T puede abortar sin obligar a que T aborte. En lugar de eso, puede que T reinicie t_i o simplemente escoja no ejecutar t_i. Si se compromete t_i, esa acción no hace que t_i sea permanente (a diferencia de la situación del Capítulo 17). En vez de eso, t_i *se compromete con* T, y puede que todavía aborte (o exija compensación—véase el Apartado 25.6.4) si T aborta. La ejecución de T no debe violar el orden parcial P. Es decir, si un trazo $t_i \rightarrow t_j$ aparece en el grafo de precedencia, $t_i \rightarrow t_j$ no debe estar en el cierre transitivo de P.

El anidamiento puede tener varios niveles de profundidad, representando la subdivisión de una transacción en subtareas, subsubtareas, etc. En el nivel inferior de anidamiento se tienen las operaciones estándar de bases de datos **leer** y **escribir** que se han usado anteriormente.

Si se permite a una subtransacción de T liberar los bloqueos al completarse, T se denomina **transacción multinivel**. Cuando una transacción multinivel representa una actividad de larga duración, a veces se denomina **saga**. De manera alternativa, si los bloqueos mantenidos por la subtransacción t_i de T se asignan de manera automática a T al concluir t_i, T se denomina **transacción anidada**.

Aunque el principal valor práctico de las transacciones multinivel surja en las transacciones complejas de larga duración, se usará el sencillo ejemplo de la Figura 25.5 para mostrar el modo en que el anidamiento puede crear operaciones de nivel superior que pueden mejorar la concurrencia. Se reescribe la transacción T_1 mediante las subtransacciones $T_{1,1}$ y $T_{1,2}$, que llevan a cabo operaciones de suma o de resta:

- T_1 consiste en
  - $T_{1,1}$, que resta 50 de A
  - $T_{1,2}$, que suma 50 a B

De manera parecida, se reescribe la transacción T_2 mediante las subtransacciones $T_{2,1}$ y $T_{2,2}$, que también llevan a cabo operaciones de suma o de resta:

- T_2 consiste en
  - $T_{2,1}$, que resta 10 de B
  - $T_{2,2}$, que suma 10 a A

No se especifica ninguna ordenación para $T_{1,1}$, $T_{1,2}$, $T_{2,1}$ ni $T_{2,2}$. Cualquier ejecución de estas subtransacciones generará un resultado correcto. La planificación de la Figura 25.5 corresponde a la planificación $< T_{1,1},\ T_{2,1},\ T_{1,2},\ T_{2,2} >$.

25.6.4 Transacciones compensadoras

Para reducir la frecuencia de las esperas de larga duración se dispone que las actualizaciones no comprometidas se muestren otras transacciones que se ejecuten de manera concurrente. En realidad, las transacciones multinivel pueden permitir esta exposición. No obstante, la exposición de datos no comprometidos crea la posibilidad de retrocesos en cascada. El concepto de **transacciones compensadoras** ayuda a tratar este problema.

Divídase la transacción T en varias subtransacciones $t_1, t_2, \ldots, t_n$. Una vez comprometida la subtransacción t_i, libera sus bloqueos. Ahora, si hay que abortar la transacción del nivel externo T, hay que deshacer el efecto de sus subtransacciones. Supóngase que las subtransacciones $t_1, \ldots, t_k$ se han comprometido y que t_{k+1} se estaba ejecutando cuando se tomó la decisión de abortar. Se pueden deshacer los efectos de t_{k+1} abortando esa subtransacción. Sin embargo, no es posible abortar las subtransacciones $t_1, \ldots, t_k$, puesto que ya se han comprometido.

En lugar de eso, se ejecuta una nueva subtransacción tc_i, denominada *transacción compensadora*, para deshacer el efecto de cada subtransacción t_i. Es necesario que cada subtransacción t_i tenga su transacción compensadora tc_i. Las transacciones compensadoras deben ejecutarse en el orden inverso $tc_k, \ldots, tc_1$. A continuación se ofrecen varios ejemplos de compensaciones:

- Considérese la planificación de la Figura 25.5, que se ha demostrado que es correcta, aunque no secuenciable en cuanto a conflictos. Cada subtransacción libera sus bloqueos una vez se completa. Supóngase que T_2 falla justo antes de su terminación, una vez que $T_{2,2}$ ha liberado sus

bloqueos. Se ejecuta una transacción compensadora para $T_{2,2}$ que resta 10 de A y una transacción compensadora para $T_{2,1}$ que suma 10 a B.

- Considérese una inserción en la base de datos por la transacción T_i que, como efecto lateral, provoca que se actualice el índice del árbol B$^+$. La operación de inserción puede haber modificado varios nodos del índice del árbol B$^+$. Otras transacciones pueden haber leído estos nodos al tener acceso a datos diferentes del registro insertado por T_i. Como en el Apartado 17.8, se puede deshacer la inserción eliminando el registro insertado por T_i. El resultado es un árbol B$^+$ correcto y consistente, pero no necesariamente uno con exactamente la misma estructura que la que se tenía antes de que se iniciara T_i. Por tanto, la eliminación es una acción compensadora para la inserción.

- Considérese una transacción de larga duración T_i que represente una reserva de viaje. La transacción T tiene tres subtransacciones: $T_{i,1}$, que hace las reservas de billetes de avión; $T_{i,2}$, que reserva los coches de alquiler; y $T_{i,3}$, que reserva las habitaciones de hotel. Supóngase que el hotel cancela la reserva. En lugar de deshacer todas las T_i, se compensa el fallo de $T_{i,3}$ eliminando la reserva de hotel antigua y realizando una nueva.

Si el sistema falla en medio de la ejecución de una transacción del nivel externo hay que hacer retroceder sus subtransacciones cuando se recupere. Las técnicas descritas en el Apartado 17.8 pueden usarse con este fin.

La compensación del fallo de una transacción exige que se utilice la semántica de la transacción que ha fallado. Para determinadas operaciones, como el incremento o la inserción en un árbol B$^+$, la compensación correspondiente se define con facilidad. Para transacciones más complejas puede que los planificadores de la aplicación tengan que definir la forma correcta de compensación en el momento en que se codifique la transacción. Para las transacciones interactivas complejas puede que sea necesario que el sistema interactúe con el usuario para determinar la forma adecuada de compensación.

25.6.5 Problemas de implementación

Los conceptos sobre las transacciones estudiados en este apartado crean serias dificultades para su implementación. Aquí se presentan unos cuantos y se estudia el modo de abordar esos problemas.

Las transacciones de larga duración deben sobrevivir a los fallos del sistema. Se puede asegurar que lo harán llevando a cabo una operación **rehacer** con las subtransacciones comprometidas, y llevando a cabo una operación **deshacer** o una compensación para cualquier subtransacción de corta duración que estuviera activa en el momento del fallo. Sin embargo, estas acciones sólo resuelven parte del problema. En los sistemas típicos de bases de datos los datos internos del sistema como las tablas de bloqueos y las marcas temporales de las transacciones se conservan en almacenamiento volátil. Para que se pueda reanudar una transacción de larga duración tras un fallo hay que restaurar esos datos. Por tanto, es necesario registrar no sólo las modificaciones de la base de datos, sino también las modificaciones de los datos internos del sistema correspondientes a las transacciones de larga duración.

El registro histórico de las actualizaciones se hace más complicado cuando hay en la base de datos ciertos tipos de elementos de datos. Un elemento de datos puede ser un diseño CAD, el texto de un documento u otra forma de diseño compuesto. Estos elementos de datos son de gran tamaño físico. Por tanto, guardar los valores antiguos y nuevos del elemento de datos en un registro del registro histórico no resulta deseable.

Hay dos enfoques para reducir la sobrecarga de asegurar la recuperabilidad de elementos de datos de gran tamaño:

- **Registro histórico de operaciones**. Sólo se guardan en el registro histórico la operación llevada a cabo en el elemento de datos y el nombre del elemento de datos. El registro histórico de operaciones también se denomina **registro histórico lógico**. Para cada operación debe haber una operación inversa. Se lleva a cabo la operación **deshacer** usando la operación inversa y la operación **rehacer** usando la misma operación. La recuperación mediante el registro histórico de operaciones resulta más difícil, ya que **rehacer** y **deshacer** no son idempotentes. Además, el uso del registro lógico para una operación que actualice varias páginas resulta muy complicado debi-

do al hecho de que algunas, pero no todas, las páginas actualizadas pueden haberse escrito en el disco, por lo que resulta difícil aplicar tanto **rehacer** como **deshacer** a la operación en la imagen del disco durante la recuperación.

El uso del registro histórico físico de rehacer y registro histórico lógico de deshacer tal y como se describe en el Apartado 17.8 proporciona las ventajas de concurrencia del registro histórico lógico y evita los inconvenientes mencionados.

- **Registro histórico y paginación en la sombra**. El registro se usa para las modificaciones de elementos de datos de pequeño tamaño, pero los elementos de datos de gran tamaño a menudo se hacen recuperables mediante una técnica de paginación en la sombra (véase el Apartado 15.3). Cuando se usa esta técnica es posible reducir la sobrecarga manteniendo sólo las páginas que realmente se modifiquen.

Independientemente de la técnica usada, las complejidades introducidas por las transacciones de larga duración y los elementos de datos de gran tamaño complican el proceso de recuperación. Por tanto, es deseable permitir que algunos datos no esenciales queden exentos del registro, y confiar en las copias de seguridad fuera de línea y en la intervención de las personas.

25.7 Gestión de transacciones en varias bases de datos

Hay que recordar del Apartado 22.8 que un sistema con múltiples bases de datos crea la ilusión de una integración lógica de las bases de datos, en un sistema de bases de datos heterogéneo en el que los sistemas locales de bases de datos pueden usar diferentes modelos lógicos de datos y lenguajes de definición y de manipulación diferentes, y pueden diferenciarse en sus mecanismo de control de concurrencia y de gestión de las transacciones.

Los sistemas con múltiples bases de datos soportan dos tipos de transacciones:

1. **Transacciones locales**. Estas transacciones las ejecuta cada sistema local de base de datos fuera del control del sistema con múltiples bases de datos.
2. **Transacciones globales**. Estas transacciones se ejecutan bajo el control del sistema con múltiples bases de datos.

El sistema con múltiples bases de datos es consciente del hecho de que pueden ejecutarse transacciones locales en los sitios locales, pero no de las transacciones concretas que se ejecutan, ni de los datos a los que tienen acceso.

Asegurar la autonomía local de cada sistema de bases de datos exige que no se realice ningún cambio en su software. Por tanto, el sistema de bases de datos de un sitio no puede comunicarse directamente con los de otros sitios para sincronizar la ejecución de transacciones globales activas en varios sitios.

Dado que el sistema con múltiples bases de datos no tiene ningún control sobre la ejecución de las transacciones globales, cada sistema local debe usar un esquema de control de concurrencia (por ejemplo, el compromiso de dos fases o las marcas temporales) para asegurarse de que su planificación sea secuenciable. Además, en caso de bloqueo, cada sistema local debe poder protegerse contra la posibilidad de interbloqueos locales.

La garantía de la secuencialidad local no es suficiente para asegurar la secuencialidad global. Como ejemplo, considérense dos transacciones globales T_1 y T_2, cada una de las cuales tiene acceso y actualiza a dos elementos de datos, A y B, ubicados en los sitios S_1 y S_2, respectivamente. Supóngase que las planificaciones locales son secuenciables. Sigue siendo posible que haya una situación en la que, en el sitio S_1, T_2 siga a T_1, mientras que, en S_2, T_1 siga a T_2, dando como resultado una planificación global no secuenciable. En realidad, aunque no haya concurrencia entre las transacciones globales (es decir, cada transacción global sólo se remite una vez que la anterior se compromete o aborta), la secuencialidad local no resulta suficiente para asegurar la secuencialidad global (véase el Ejercicio práctico 25.7).

En función de la implementación de los sistemas locales de bases de datos puede que una transacción global no pueda controlar el comportamiento exacto de los bloqueos de sus subtransacciones. Por tanto, incluso si todos los sistemas locales de bases de datos siguen el bloqueo de dos fases, puede que sólo sea posible asegurar que cada transacción local siga las reglas del protocolo. Por ejemplo, puede que

un sistema local de bases de datos comprometa su subtransacción y libere sus bloqueos mientras que la subtransacción de otro sistema local se sigue ejecutando. Si los sistemas locales permiten el control del comportamiento de los bloqueos y todos los sistemas siguen el bloqueo de dos fases, el sistema con múltiples bases de datos puede asegurar que las transacciones globales se bloqueen en la modalidad de dos fases y que los puntos de bloqueo de las transacciones en conflicto definan su orden global de secuencia. Aunque diferentes sistemas locales siguen diferentes mecanismos de control de concurrencia, sin embargo, esta forma directa de control global no funciona.

Hay muchos protocolos para asegurar la consistencia pese a la ejecución concurrente de las transacciones globales y locales en los sistemas con múltiples bases de datos. Algunos se basan en imponer las condiciones suficientes para asegurar la secuencialidad global. Otros sólo aseguran una forma de consistencia más débil que la secuencialidad, pero la consiguen por medios menos restrictivos. Se considerará uno de estos últimos esquemas: la *secuencialidad de dos niveles*. El Apartado 25.6 describe más enfoques de la consistencia sin secuencialidad; se citan otros enfoques en las notas bibliográficas.

Un problema relacionado en los sistemas de bases de datos es el del compromiso atómico global. Si todos los sistemas locales siguen el protocolo de compromiso de dos fases, puede usarse este protocolo para conseguir la atomicidad global. No obstante, puede que los sistemas locales no diseñados para ser parte de un sistema distribuido no puedan participar en este protocolo. Aunque un sistema local pueda soportar el compromiso de dos fases, la organización propietaria del sistema puede que no desee permitir las esperas en los casos en los que se producen bloqueos. En esos casos, pueden alcanzarse compromisos que permitan la falta de atomicidad en determinadas modalidades de fallo. En la bibliografía proporcionada hay un tratamiento más extenso de estos asuntos (véanse las notas bibliográficas).

25.7.1 Secuencialidad de dos niveles

La **secuencialidad de dos niveles** (**S2N**) asegura la secuencialidad en dos niveles del sistema:

- Cada sistema local de bases de datos asegura la secuencialidad local entre sus transacciones locales, incluidas las que forman parte de las transacciones globales.
- El sistema con múltiples bases de datos asegura sólo la secuencialidad entre las transacciones globales–*ignorando las ordenaciones inducidas por las transacciones locales*.

Resulta sencillo hacer que se cumpla cada uno de estos niveles de secuencialidad. Los sistemas locales ya ofrecen garantías de secuencialidad; por tanto, el primer requisito es sencillo de lograr. El segundo requisito sólo se aplica a una proyección de la planificación global en la que las transacciones locales no aparecen. Por tanto, el sistema con múltiples bases de datos puede asegurar el segundo requisito usando técnicas estándar de control de concurrencia (no importa la elección concreta de la técnica).

Los dos requisitos de S2N no resultan suficientes para asegurar la secuencialidad global. Sin embargo, con el enfoque S2N, se adopta un requisito más débil que la secuencialidad, denominado **corrección fuerte**:

1. La conservación de la consistencia tal y como especifica un conjunto de restricciones de consistencia.
2. La garantía de que el conjunto de elementos de datos leído por cada transacción sea consistente.

Puede probarse que determinadas restricciones al comportamiento de las transacciones, combinadas con S2N, son suficientes para asegurar la corrección fuerte (aunque no necesariamente para asegurar la secuencialidad). Se citarán varias de estas restricciones.

En cada uno de los protocolos se distingue entre los **datos locales** y los **datos globales**. Los elementos locales de datos pertenecen a un sitio concreto y se hallan bajo el control exclusivo de ese sitio. Obsérvese que no puede haber restricciones de consistencia entre los elementos locales de datos de sitios distintos. Los elementos globales de datos pertenecen al sistema con múltiples bases de datos y, aunque puede que se almacenen en un sitio local, se hallan bajo el control del sistema con múltiples bases de datos.

El **protocolo globales–lectura** permite que las transacciones globales lean, pero no actualicen, los elementos locales de datos, mientras que impide el acceso a los datos globales por parte de las transacciones locales. El protocolo globales–lectura asegura la corrección fuerte si se cumplen las reglas siguientes:

1. Las transacciones locales sólo tienen acceso a los elementos locales de datos.
2. Las transacciones globales pueden tener acceso a los elementos globales de datos, y pueden leer los elementos locales de datos (aunque no deben escribirlos).
3. No hay restricciones de consistencia entre los elementos de datos locales y los globales.

El **protocolo locales–lectura** concede a las transacciones locales acceso de lectura a los datos globales, pero impide el acceso a los datos locales por las transacciones globales. En este protocolo hay que introducir el concepto de **dependencia del valor**. Cada transacción tiene una dependencia del valor si el valor que escribe en el elemento de datos de un sitio depende del valor que ha leído para el elemento de datos de otro sitio.

El protocolo locales–lectura asegura la corrección fuerte si se cumplen las condiciones siguientes:

1. Las transacciones locales pueden tener acceso a los elementos locales de datos y pueden leer los elementos globales de datos almacenados en ese sitio (aunque no deban escribir elementos globales de datos).
2. Las transacciones globales sólo tienen acceso a los elementos globales de datos.
3. Ninguna transacción puede tener dependencia de ningún valor.

El **protocolo globales–lectura–escritura/locales–lectura** es el más generoso en términos de acceso a los datos de los protocolos que se han considerado. Permite que las transacciones globales lean y escriban los datos locales y que las transacciones locales lean los datos globales. Sin embargo, impone tanto la condición de dependencia del valor del protocolo locales–lectura como la condición del protocolo globales–lectura de que no haya restricciones de consistencia entre los datos locales y los globales.

El protocolo globales–lectura–escritura/locales–lectura asegura la corrección fuerte si se cumplen las condiciones siguientes:

1. Las transacciones locales pueden tener acceso a los elementos locales de datos y pueden leer los elementos globales de datos almacenados en ese sitio (aunque no deben escribir los elementos globales de datos).
2. Las transacciones globales pueden tener acceso a los elementos globales de datos y a los elementos locales de datos (es decir, pueden leer y escribir todos los datos).
3. No hay restricciones de consistencia entre los elementos locales de datos y los globales.
4. Ninguna transacción puede tener dependencia de ningún valor.

25.7.2 Aseguramiento de la secuencialidad global

Los primeros sistemas con múltiples bases de datos restringían las transacciones globales a ser sólo de lectura. Así evitaban la posibilidad de que las transacciones globales introdujeran inconsistencia en los datos, pero no eran lo bastante restrictivas como para asegurar la secuencialidad global. Resulta posible realmente obtener planificaciones globales de este tipo y desarrollar un esquema para asegurar la secuencialidad global, y se pide al lector que haga las dos cosas en el Ejercicio práctico 25.8.

Hay varios esquemas generales para asegurar la secuencialidad global en entornos en los que se pueden ejecutar actualizaciones y transacciones sólo de lectura. Varios de estos esquemas se basan en la idea del **billete**. Se crea un elemento de datos especial denominado billete en cada sistema local de bases de datos. Cada transacción global que tenga acceso a los datos de un sitio debe escribir en el billete de ese sitio. Este requisito asegura que las transacciones globales entren en conflicto directamente en cada sitio que visiten. Además, el gestor de las transacciones globales puede controlar el orden en el que se

secuencian las transacciones globales controlando el orden en el que tienen acceso a los billetes. Las referencias a estos esquemas aparecen en las notas bibliográficas.

Si se desea asegurar la secuencialidad global en entornos en los que no se generan en cada sitio conflictos locales directos, hay que realizar algunas suposiciones sobre las planificaciones autorizadas por el sistema local de bases de datos. Por ejemplo, si las planificaciones locales son tales que el orden de compromiso y el de secuenciación son siempre idénticos, se puede asegurar la secuencialidad controlando únicamente el orden en que se comprometen las transacciones.

El problema de los esquemas que aseguran la secuencialidad global es que pueden restringir la concurrencia de manera inadecuada. Resultan especialmente propensos a hacerlo porque la mayor parte de las transacciones remiten al sistema de bases de datos subyacente las sentencias SQL, en lugar de remitir cada uno de los pasos de **lectura**, **escritura**, **compromiso** y **aborto**. Aunque sigue siendo posible asegurar la secuencialidad global con esta suposición, el nivel de concurrencia puede ser tal que otros esquemas, como la técnica de secuencialidad de dos niveles estudiada en el Apartado 25.7.1, resulten alternativas atractivas.

25.8 Resumen

- Los flujos de trabajo son actividades que implican la ejecución coordinada de varias tareas llevadas a cabo por diferentes entidades de proceso. No sólo existen en las aplicaciones informáticas, sino también en casi todas las actividades de una organización. Con el auge de las redes y la existencia de numerosos sistemas autónomos de bases de datos, los flujos de trabajo ofrecen una manera adecuada de llevar a cabo las tareas que implican a varios sistemas.

- Aunque los requisitos transaccionales ACID habituales resultan demasiado estrictos o no pueden implementarse para estas aplicaciones de flujo de trabajo, los flujos de trabajo deben satisfacer un conjunto limitado de propiedades transaccionales que garantiza que no se dejen los procesos en estados inconsistentes.

- Los monitores de procesamiento de transacciones se desarrollaron inicialmente como servidores con varias hebras que podían atender a un gran número de terminales desde un único proceso. Desde entonces han evolucionado y hoy en día ofrecen la infraestructura para la creación y gestión de sistemas complejos de procesamiento de transacciones que tienen gran número de clientes y varios servidores. Proporcionan servicios como colas duraderas de las solicitudes de los clientes y de las respuestas de los servidores, el encaminamiento de los mensajes de los clientes a los servidores, la mensajería persistente, el equilibrio de carga y la coordinación del compromiso de dos fases cuando las transacciones tienen acceso a varios servidores.

- Los sistemas de comercio electrónico se han convertido en una parte principal del comercio. Hay varios aspectos de las bases de datos en los sistemas de comercio electrónico. La administración de los catálogos, especialmente su personalización, se realiza con bases de datos. Los mercados electrónicos ayudan a fijar el precio de los productos mediante subastas, subastas inversas o bolsas. Se necesitan sistemas de bases de datos de alto rendimiento para manejar este intercambio. Los pedidos se liquidan mediante sistemas de pago electrónico, que también necesitan sistemas de bases de datos de alto rendimiento para manejar tasas de transacciones muy elevadas.

- Las memorias principales de gran tamaño se aprovechan en determinados sistemas para conseguir una gran productividad del sistema. En esos sistemas el registro histórico constituye un cuello de botella. Bajo el concepto de compromiso en grupo se puede reducir el número de salidas hacia el almacenamiento estable, lo que libera este cuello de botella.

- La gestión eficiente de las transacciones interactivas de larga duración resulta más compleja debido a las esperas de larga duración y a la posibilidad de los abortos. Dado que las técnicas de control de concurrencia usadas en el Capítulo 16 emplean las esperas, los abortos o las dos cosas, hay que considerar técnicas alternativas. Estas técnicas deben asegurar la corrección sin exigir la secuencialidad.

- Las transacciones de larga duración se representan como transacciones atómicas con operaciones atómicas de la base de datos anidadas en el nivel inferior. Si una transacción falla, sólo se abortan las transacciones activas de corta duración. Las transacciones activas de larga duración se reanudan una vez que se han recuperado todas las transacciones de corta duración. Se necesita una transacción compensadora para deshacer las actualizaciones de las transacciones anidadas que se hayan comprometido, si falla la transacción del nivel exterior.
- En los sistemas con restricciones de tiempo real la corrección de la ejecución no sólo implica la consistencia de la base de datos, sino también el cumplimiento de los tiempos límite. La amplia variabilidad de los tiempos de ejecución de las operaciones de lectura y de escritura complica el problema de la gestión de las transacciones en sistemas con restricciones temporales.
- Los sistemas con múltiples bases de datos proporcionan un entorno en el que las nuevas aplicaciones de bases de datos pueden tener acceso a los datos desde gran variedad de bases de datos existentes previamente ubicadas en varios entornos heterogéneos de hardware y de software.

  Los sistemas locales de bases de datos pueden usar diferentes modelos lógicos y lenguajes diferentes de definición y de manipulación de datos, y puede que se diferencien en sus mecanismos de control de concurrencia y de gestión de las transacciones. Los sistemas con múltiples bases de datos crean la ilusión de la integración lógica de las bases de datos sin exigir su integración física.

Términos de repaso

- Monitor TP.
- Arquitecturas de los monitores TP.
  - Proceso por cliente.
  - Servidor único.
  - Varios servidores, un encaminador.
  - Varios servidores, varios encaminadores.
- Multitarea.
- Cambio de contexto.
- Servidor con varias hebras.
- Gestor de la cola.
- Coordinación de aplicaciones.
  - Gestor de recursos.
  - Llamada a procedimiento remoto (Remote Procedure Call, RPC).
- Flujos de trabajo transaccionales.
  - Tarea.
  - Entidad de procesamiento.
  - Especificación del flujo de trabajo.
  - Ejecución del flujo de trabajo.
- Estado del flujo de trabajo.
  - Estados de ejecución.
  - Valores de salida.
  - Variables externas.
- Atomicidad ante fallos del flujo de trabajo.
- Estados de terminación del flujo de trabajo:
  - Aceptable.
  - No aceptable.
  - Comprometido.
  - Abortado.
- Recuperación en los flujos de trabajo.
- Sistema gestor de flujos de trabajo.
- Arquitecturas de los sistemas gestores del flujo de trabajo:
  - Centralizadas.
  - Parcialmente distribuidas.
  - Completamente distribuidas.
- Comercio electrónico.
- Catálogos electrónicos.
- Mercados.
  - Subastas.
  - Subastas inversas.
  - Bolsas.
- Liquidación de pedidos.
- Certificados digitales.
- Bases de datos en memoria principal.
- Compromiso en grupo.
- Sistemas de tiempo real.
- Tiempo límite:
  - Estricto.
  - Firme.
  - Flexible.
- Bases de datos de tiempo real.
- Transacciones de larga duración.
- Exposición de datos no comprometidos.
- Ejecuciones no secuenciables.
- Transacciones anidadas.

- Transacciones multinivel.
- Saga.
- Transacciones compensadoras.
- Registro histórico lógico.
- Sistemas con múltiples bases de datos.
- Autonomía.
- Transacciones locales.
- Transacciones globales.
- Secuencialidad de dos niveles (S2N).
- Corrección fuerte.
- Datos locales.
- Datos globales.
- Protocolos:
  - ☐ Globales–lectura.
  - ☐ Locales–lectura.
  - ☐ De dependencia del valor.
  - ☐ Globales–lectura–escritura/locales–lectura.
- Aseguramiento de la secuencialidad global.
- Billete.

Ejercicios prácticos

25.1 Al igual que los sistemas de bases de datos, los sistemas de flujo de trabajo también necesitan la gestión de la concurrencia y de la recuperación. Indíquense tres motivos por los que no se puede aplicar simplemente un sistema relacional de bases de datos usando bloqueo de dos fases, registro histórico de operaciones físicas de deshacer y compromiso de dos fases.

25.2 Considérese un sistema de bases de datos en memoria principal que se recupera de un fallo del sistema. Explíquense las ventajas relativas de:

a. Volver a cargar toda la base de datos en memoria principal antes de reanudar el procesamiento de las transacciones.
b. Cargar los datos a medida que los soliciten las transacciones.

25.3 Indíquese si un sistema de transacciones de alto rendimiento es necesariamente un sistema de tiempo real. Explíquese el motivo.

25.4 Explíquese el motivo por el que puede que no resulte práctico exigir la secuencialidad para las transacciones de larga duración.

25.5 Considérese un proceso con varias hebras que entrega mensajes desde una cola duradera de mensajes persistentes. Pueden ejecutarse de manera concurrente diferentes hebras, que intentan entregar mensajes diferentes. En caso de fallo en la entrega el mensaje debe restaurarse en la cola. Modélense las acciones que lleva a cabo cada hebra como una transacción multinivel, de manera que no haga falta mantener los bloqueos en la cola hasta que se entregue cada mensaje.

25.6 Descríbanse las modificaciones que hay que hacer en cada uno de los esquemas de recuperación tratados en el Capítulo 17 si se permiten las transacciones anidadas. Explíquense también las diferencias que se producen si se permiten las transacciones multinivel.

25.7 Considérese un sistema con múltiples bases de datos en el que se garantice que, como máximo, está activa una transacción global en un momento dado y que cada sistema local asegura la secuencialidad local.

a. Sugiéranse maneras de que el sistema con múltiples bases de datos pueda asegurar que haya como máximo una transacción global activa en cualquier momento dado.
b. Demuéstrese mediante un ejemplo que resulta posible que se produzca una planificación global no secuenciable pese a estas suposiciones.

25.8 Considérese un sistema con múltiples bases de datos en el que cada sitio local asegura la secuencialidad local y todas las transacciones globales son sólo de lectura.

a. Demuéstrese mediante un ejemplo que pueden producirse ejecuciones no secuenciables en este sistema.
b. Muéstrese la manera en que se podría usar un esquema de billete para asegurar la secuencialidad global.

Ejercicios

25.9 Explíquese el modo en que los monitores TP administran los recursos de la memoria y del procesador de manera más efectiva que los sistemas operativos habituales.

25.10 Compárense las características de los monitores TP con las proporcionadas por los servidores Web que soportan servlets (estos servidores se han denominado *TP-lite*—TP ligeros).

25.11 Considérese el proceso de admisión de nuevos alumnos en la universidad (o de nuevos empleados en la organización).

- **a.** Dese una imagen de alto nivel del flujo de trabajo comenzando por el procedimiento de matrícula de los estudiantes.
- **b.** Indíquense los estados de terminación aceptables y los pasos que implican intervención de personas.
- **c.** Indíquense los posibles errores (incluido el vencimiento del tiempo límite) y el modo en que se tratan.
- **d.** Estúdiese la cantidad de flujo de trabajo que se ha automatizado en la universidad.

25.12 Contéstense las preguntas siguientes con respecto a sistemas electrónicos de pago.

- **a.** Explíquese por qué las transacciones electrónicas realizadas usando números de la tarjeta de crédito son inseguras.
- **b.** Un alternativa es tener una pasarela para el pago electrónico mantenida por la compañía de la tarjeta de crédito, y que el sitio que reciba el pago redirija a los clientes a la pasarela para hacer el pago.
  - I. Explíquense las ventajas que ofrece este sistema si la entrada no autentica al usuario.
  - II. Explíquense las ventajas si la pasarela tiene un mecanismo para autenticar al usuario.
- **c.** Algunas compañías de tarjetas de crédito ofrecen un número de la tarjeta de crédito de un solo uso como método más seguro de pago electrónico. Los clientes se conectan con el sitio Web de las compañías de tarjetas de crédito para conseguir un número de un solo uso. Explíquese la ventaja que ofrece este sistema con respecto a usar números de tarjeta de crédito normales. Explíquense también sus ventajas e inconvenientes con respecto a las pasarelas de pago con autentificación.
- **d.** ¿Alguno de estos sistemas garantiza la misma privacidad disponible cuando los pagos se hacen en efectivo? Explíquese la respuesta.

25.13 Si toda la base de datos cabe en la memoria principal, indíquese si sigue haciendo falta un sistema de bases de datos para administrar los datos. Explíquese la respuesta.

25.14 En la técnica de compromiso en grupo indicar el número de transacciones que deben formar parte de un grupo. Explíquese la respuesta.

25.15 En un sistema de bases de datos que utilice el registro histórico de escritura adelantada indíquese el número de accesos a disco necesarios para leer un elemento de datos de una página de disco especificada en el peor caso posible. Explíquese el motivo por el que esto supone un problema para los diseñadores de sistemas de bases de datos de tiempo real.

25.16 Indíquese la finalidad de las transacciones compensadoras. Preséntense dos ejemplos de su uso.

25.17 Explíquense las semejanzas entre un flujo de trabajo y una transacción de larga duración.

Notas bibliográficas

Gray y Edwards [1995] proporcionan una introducción a las arquitecturas de los monitores TP; en el libro Gray y Reuter [1993] se ofrece una descripción detallada (y excelente) de los sistemas de procesamiento de transacciones, incluidos capítulos sobre los monitores TP. X/Open [1991] define la interfaz X/Open XA. El procesamiento de las transacciones en Tuxedo se describe en Wipfler [1987] es uno de los textos sobre el desarrollo de aplicaciones mediante CICS.

Fischer [2001] es un manual sobre los sistemas de flujos de trabajo. Un modelo de referencia para los flujos de trabajo, propuesto por la Coalición para la gestión de flujos de trabajo (Workflow Management Coalition), se presenta en Hollingsworth [1994] En Hollingsworth [2004], se presenta una revisión del modelo, incluyendo sus conexiones con el *modelo de procesos de negocios*, como parte del manual sobre flujos de trabajo (Fischer [2004]). El sitio Web de la coalición es www.wfmc.org. La descripción de flujos de trabajo que se ha dado sigue el modelo de Rusinkiewicz y Sheth [1995].

Loeb [1998] proporciona una descripción detallada de las transacciones electrónicas seguras.

Garcia-Molina y Salem [1992] ofrecen una introducción a las bases de datos en memoria principal. Jagadish et al. [1993] describen un algoritmo de recuperación diseñado para las bases de datos en memoria principal. En Jagadish et al. [1994] se describe un gestor de almacenamiento para las bases de datos en memoria principal.

Las bases de datos de tiempo real se estudian en Lam y Kuo [2001]. Entre las aplicaciones de procesamiento de datos en tiempo real se encuentra TelegraphCQ (Chandrasekaran et al. [2003]). El control de concurrencia y las planificaciones en las bases de datos de tiempo real se estudian en Haritsa et al. [1990], Hong et al. [1993] y en Pang et al. [1995]. Ozsoyoglu y Snodgrass [1995] es una reseña de la investigación en las bases de datos de tiempo real y en las bases de datos temporales.

Las transacciones anidadas y las transacciones multinivel se presentan en Lynch [1983], Moss [1985], Lynch y Merritt [1986], Fekete et al. [1990], Weikum [1991], Korth y Speegle [1994] y en Pu et al. [1988]. Los aspectos teóricos de las transacciones multinivel se presentan en Lynch et al. [1988] y en Weihl y Liskov [1990].

Se han definido varios modelos de transacciones ampliadas, incluidos Sagas (Garcia-Molina y Salem [1987]), el modelo ConTract (Wachter y Reuter [1992]) y ARIES (Mohan et al. [1992] y Rothermel y Mohan [1989]). La división de las transacciones para conseguir un rendimiento mayor se aborda en Shasha et al. [1995]. La recuperación en los sistemas de transacciones anidadas se estudia en Moss [1987], Haerder y Rothermel [1987] y Rothermel y Mohan [1989].

El procesamiento de las transacciones de larga duración se considera en Weikum y Schek [1984], Haerder y Rothermel [1987], Weikum et al. [1990] y en Korth et al. [1990]. Salem et al. [1994] presentan una extensión del bloqueo de dos fases para las transacciones de larga duración al permitir la liberación precoz de los bloqueos en ciertas circunstancias.

El procesamiento de las transacciones en los sistemas con múltiples bases de datos se estudia en Mehrotra et al. [2001]. El esquema del billete se presenta en Georgakopoulos et al. [1994]. S2N se introduce en Mehrotra et al. [1991].

PARTE 9

Estudio de casos

Esta parte describe la forma en que los distintos sistemas de bases de datos integran los conceptos descritos anteriormente en el libro. Comienza en el Capítulo 26 con el estudio de un sistema de bases de datos de código abierto ampliamente usado: PostgreSQL. En los Capítulos 27, 28 y 29 se estudian tres sistemas comerciales de bases de datos muy usados: DB2 de IBM, Oracle y SQL Server de Microsoft. Representan tres de los sistemas comerciales de bases de datos más usados.

Cada uno de estos capítulos muestra características únicas de cada sistema de bases de datos: herramientas, variaciones y extensiones de SQL, y la arquitectura del sistema, incluyendo organización del almacenamiento, procesamiento de consultas, control de concurrencia, recuperación y réplicas.

Los capítulos tratan solamente los aspectos clave de los productos de bases de datos que describen y por tanto no se deberían usar como una descripción completa. Además, puesto que los productos se mejoran periódicamente, sus detalles pueden cambiar. Cuando se usa una versión particular del producto hay que asegurarse de consultar en los manuales de usuario los detalles específicos.

Se debe tener presente que en los capítulos de esta parte se usa a menudo terminología industrial en lugar de académica. Por ejemplo, se usa *tabla* en lugar de *relación*, *fila* en lugar de *tupla* y *columna* en lugar de *atributo*.

CAPÍTULO 26

PostgreSQL

Anastassia Ailamaki, Sailesh Krishnamurthy, Spiros Papadimitriou, Bianca Schroeder
CMU

PostgreSQL es un sistema gestor de bases de datos relacionales de objetos de código abierto. Es un descendiente de uno de los primeros sistemas de este tipo, el sistema Postgres desarrollado bajo la dirección del profesor Michael Stonebraker en la Universidad de California en Berkeley. El nombre "Postgres" proviene del nombre de un sistema pionero de bases de datos relacionales, Ingres, también desarrollado bajo la dirección de Stonebraker en Berkeley. Actualmente, PostgreSQL soporta SQL92 y SQL:1999 y ofrece características como las consultas complejas, las claves externas, los disparadores, las vistas, la integridad transaccional y el control de concurrencia de varias versiones. Además, los usuarios pueden ampliar PostgreSQL con tipos de datos, funciones, operadores y métodos de indexación nuevos. PostgreSQL trabaja con gran variedad de lenguajes de programación (incluidos C, C++, Java, Perl, TcI y Python). Quizás, el punto fuerte de PostgreSQL sea que, junto con MySQL, son los dos sistemas de bases de datos relacionales de código abierto más utilizados. La licencia de PostgreSQL es la licencia BSD, que concede libre de cargo permiso para el uso, modificación y distribución del código y de la documentación de PostgreSQL con cualquier propósito.

26.1 Introducción

A lo largo de más de una década, PostgreSQL ha tenido varias versiones importantes. El primer sistema prototipo, con el nombre de Postgres, se presentó en la conferencia SIGMOD de la ACM en 1988. La versión 1 se distribuyó a los usuarios en 1989. Después de que las versiones posteriores añadieran un nuevo sistema de reglas, soporte para varios gestores de almacenamiento y un ejecutor de consultas mejorado, los desarrolladores del sistema se centraron en la portabilidad y en el rendimiento hasta 1994, cuando se añadió un intérprete del lenguaje SQL. Con un nuevo nombre, Postgres95, el sistema se distribuyó por Web y, posteriormente, fue comercializado por Illustra Information Technologies (que posteriormente se fusionó con Informix, que ahora es propiedad de IBM). Hacia 1996 el nombre Postgres95 fue sustituido por PostgreSQL, para reflejar la relación entre el Postgres original y las versiones más recientes con capacidades de SQL.

PostgreSQL se puede ejecutar bajo prácticamente todos los sistemas operativos tipo Unix, incluidos Linux y OS X para Apple de Macintosh. PostgreSQL también puede ejecutarse bajo Microsoft Windows en el entorno Cygwin, el cual proporciona emulación de Linux bajo Windows. La versión más reciente, la 8.0, publicada en enero de 2005, ofrece soporte nativo de Microsoft Windows.

Hoy en día PostgreSQL se utiliza para implementar varias aplicaciones de investigación y de producción diferentes (como el proyecto informático científico Sequoia 2000) y como herramienta educativa en

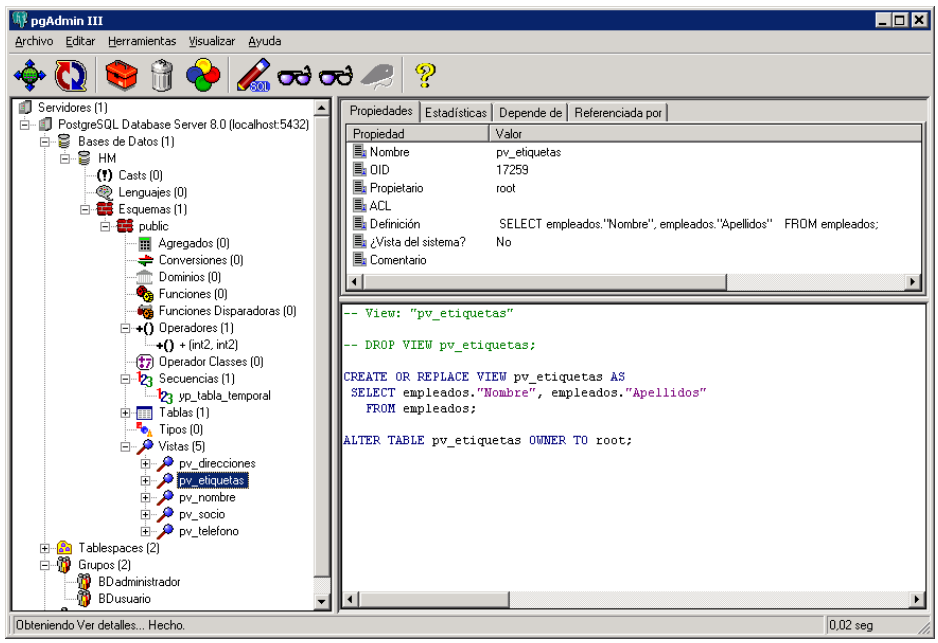


Figura 26.1 pgAdmin III: una interfaz gráfica de usuario de código abierto para la administración de bases de datos.

varias universidades. El sistema sigue evolucionando gracias a las contribuciones de una comunidad de alrededor de 1.000 desarrolladores. En este capítulo se explicará el funcionamiento de PostgreSQL, comenzando por las interfaces y los lenguajes de usuario y siguiendo por el corazón del sistema (las estructuras de datos y los mecanismos de control de concurrencia).

26.2 Interfaces de usuario

La distribución estándar de PostgreSQL incluye herramientas de línea de comandos para la administración de las bases de datos. Sin embargo, existe un gran número de herramientas de administración y de diseño comerciales y de código abierto que soportan PostgreSQL. PostgreSQL ofrece un amplio conjunto de interfaces de usuario.

26.2.1 Interfaces de terminal interactivas

Como la mayor parte de los sistemas de bases de datos, PostgreSQL ofrece herramientas de línea de comandos para la administración de bases de datos. El principal cliente de terminal interactiva es psql, que se modeló a partir del intérprete de comandos de Unix y permite la ejecución de comandos de SQL en el servidor, así como otras operaciones (como la copia en el lado del cliente). Algunas de sus características son:

- **Variables.** psql ofrece características de sustitución de variables, parecidas a los intérpretes de comandos Unix habituales.
- **Interpolación SQL**. El usuario puede sustituir ("interpolar") las variables de psql en instrucciones SQL regulares colocando un punto y coma delante del nombre de la variable.
- **Edición de la línea de comandos**. psql utiliza la biblioteca Readline de GNU para hacer más cómoda la edición de líneas, que permite autocompletar las entradas mediante tabuladores.

PostgreSQL también ofrece pgtksh y pgtclsh, que son versiones de los intérpretes de comandos tipo Tk y Tcl (wish) los cuales incluyen adicionalmente enlaces con PostgreSQL. Tcl/Tk es un lenguaje de guiones flexible, empleado generalmente para la creación rápida de prototipos.

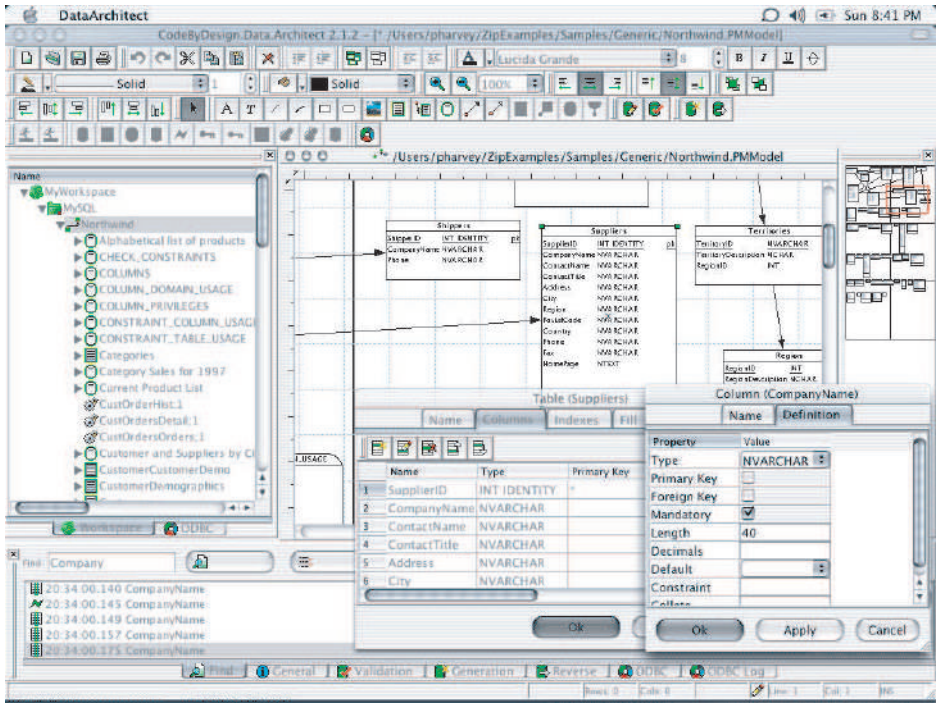


Figura 26.2 Data Architect: una interfaz gráfica de usuario multiplataforma para el diseño de bases de datos.

26.2.2 Interfaces gráficas

La distribución estándar de PostgreSQL no contiene ninguna herramienta gráfica. No obstante, existen varias herramientas con interfaz gráfica de usuario, de forma que se pueden escoger entre las posibilidades comerciales y las de código abierto. Muchas de ellas experimentan ciclos de publicación rápidos; la lista siguiente refleja la situación en el momento de escribir este libro.

Se encuentran disponibles herramientas gráficas para la administración, incluidas pgAccess y pgAdmin, la última de las cuales puede verse en la Figura 26.1. Entre las herramientas para el diseño de bases de datos están TORA y Data Architect, la última de las cuales puede verse en la Figura 26.2.

PostgreSQL trabaja con varias herramientas comerciales para el diseño de formularios y la generación de informes. Entre las posibilidades de código abierto se encuentran Rekall (que puede verse en las Figuras 26.3 y 26.4), GNU Report Generator y un conjunto de herramientas más general, GNU Enterprise.

26.2.3 Interfaces para lenguajes de programación

PostgreSQL ofrece interfaces nativas para ODBC y para JDBC, así como enlaces con la mayor parte de los lenguajes de programación, incluidos C, C++, PHP, Perl, Tcl/Tk, ECPG, Python y Ruby.

La interfaz de programación para aplicaciones en C de PostgreSQL es libpq, que también es el motor subyacente de la mayor parte de los enlaces de lenguajes de programación (por tanto, todas sus características también están disponibles en los demás lenguajes soportados). La biblioteca libpq soporta tanto la ejecución síncrona de los comandos de SQL y de las instrucciones preparadas como la asíncrona. Es reentrante y aporta seguridad a las hebras (*threads*). Utiliza variables de entorno para ciertos parámetros y un archivo de contraseñas opcional para las conexiones que exigen autentificación.

26.3 Variaciones y extensiones de SQL

PostgreSQL cumple la norma SQL de ANSI. Soporta casi todas las características del nivel básico de SQL92 (que es lo que la mayor parte de los fabricantes de sistemas de bases de datos relacionales entienden por conformidad con SQL92) y muchas de las de los niveles intermedio y completo. Finalmente, soporta

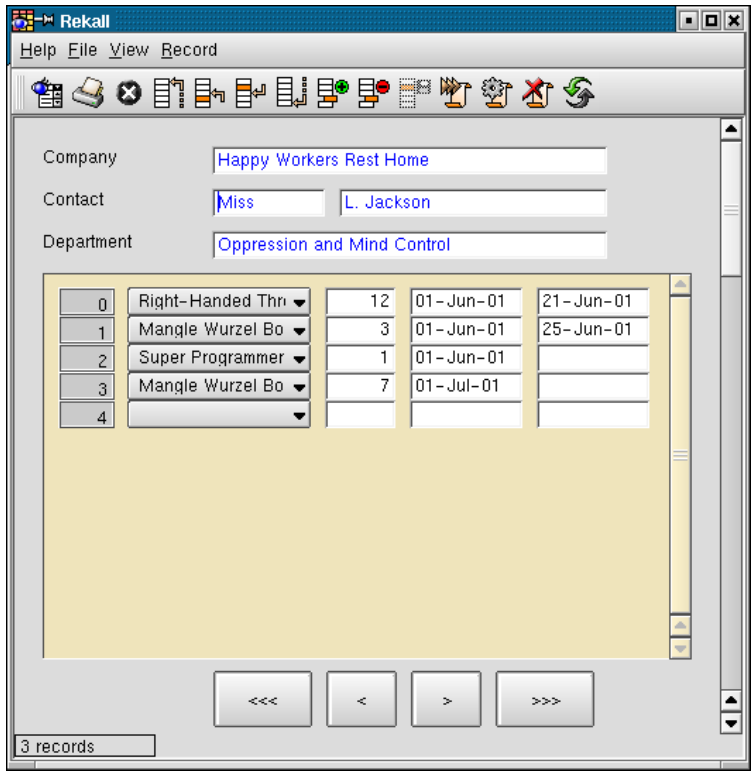


Figura 26.3 Rekall: interfaz gráfica de usuario para diseño de formularios.

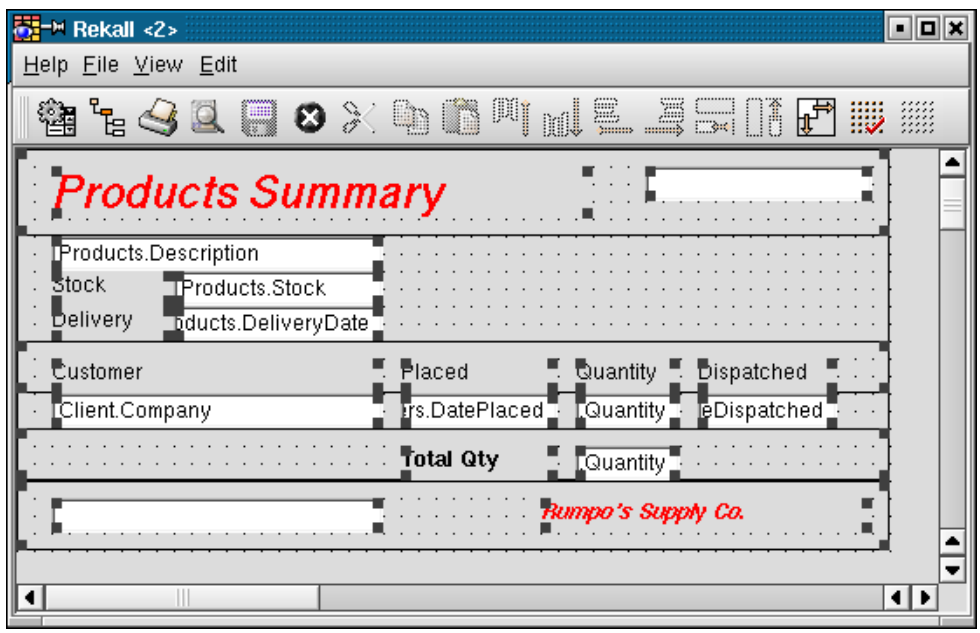


Figura 26.4 Rekall: interfaz gráfica de usuario para diseño de informes.

varias de las características de SQL:1999 (incluida la mayor parte de las características relacionales de objetos que se describen en el Capítulo 9); de hecho, algunas de ellas fueron introducidas por PostgreSQL o por sus antecesores. Carece de características OLAP (principalmente, **cube** y **rollup**), pero los datos de PostgreSQL pueden cargarse con facilidad en servidores OLAP externos de código abierto (como Mondrian), así como en productos comerciales.

26.3.1 Tipos de PostgreSQL

PostgreSQL incluye soporte para varios tipos no normalizados, que resultan útiles para dominios de aplicaciones concretas. Además, los usuarios pueden definir nuevos tipos con el comando **create type**. Esto incluye nuevos tipos básicos de bajo nivel, generalmente escritos en C (véase el Apartado 26.3.3.1)

26.3.1.1 El sistema de tipos de PostgreSQL

Los tipos de PostgreSQL pueden clasificarse en las categorías siguientes:

- **Tipos básicos.** Los tipos básicos también se conocen como **tipos abstractos de datos**; es decir, módulos que encapsulan al mismo tiempo un estado y un conjunto de operaciones. Se implementan por debajo del nivel SQL, generalmente en un lenguaje como C (véase el Apartado 26.3.3.1). Entre los ejemplos se encuentran **int4** (que ya está incluido en PostgreSQL) y **complex** (que se incluye como tipo de extensión opcional). Cada tipo básico de PostgreSQL va acompañado automáticamente por un tipo de *array* que pueda almacenar *arrays* de longitud variable de ese tipo básico concreto.
- **Tipos compuestos.** Se corresponden con las filas de las tablas; es decir, son una lista de nombres de campos y de sus tipos básicos respectivos. Siempre que se crea una tabla se crea implícitamente un tipo compuesto, aunque los usuarios también pueden crearlos de manera explícita.
- **Dominios.** Son muy parecidos a los tipos básicos (y a menudo ambos tipos son intercambiables), sin embargo pueden tener restricciones respecto de los valores que les están permitidos.
- **Pseudotipos.** Actualmente, PostgreSQL soporta los pseudotipos siguientes: *any*, *anyarray*, *anyelement*, *cstring*, *internal*, *language handler*, *record*, *trigger* y *void*. No pueden utilizarse en los tipos compuestos (y, por tanto, no pueden utilizarse para las columnas de las tablas), pero pueden emplearse como argumentos y tipos de retorno de funciones definidas por los usuarios.
- **Tipos polimórficos.** Los dos pseudotipos *anyelement* y *anyarray* se conocen como **polimórficos**. Las funciones con argumentos de estos tipos (denominadas, a su vez, **funciones polimórficas**) pueden operar sobre cualquier tipo. PostgreSQL tiene un esquema sencillo de resolución de tipos que exige que: (1) en cada invocación de una función polimórfica todas las ocurrencias de los tipos polimórficos estén vinculadas al mismo tipo (es decir, una función definida como *f*(*fanyelement*, *fanyelement)* sólo puede operar sobre parejas del mismo tipo) y (2) si el tipo devuelto es polimórfico, al menos uno de los argumentos debe ser del mismo tipo polimórfico.

26.3.1.2 Tipos no estándar

Los tipos descritos en este apartado se incluyen en la distribución estándar. Además, gracias a la naturaleza abierta de PostgreSQL, se han aportado varios tipos adicionales, como los números complejos y el ISBN/ISSN (véase el Apartado 26.3.3).

Los tipos de datos geométricos (*point*, *line*, *lseg*, *box*, *polygon*, *path*, *circle*) se utilizan en los sistemas de información geográfica para representar objetos espaciales bidimensionales como los puntos, los segmentos de línea, los polígonos, los caminos y los círculos. En PostgreSQL se dispone de numerosas funciones y operadores para llevar a cabo diferentes operaciones geométricas como el cambio de escala, las traslaciones, las rotaciones y la determinación de las intersecciones. Además, PostgreSQL soporta el indexado de estos tipos mediante árboles R (Apartados 24.3.5.3 y 26.5.2.1).

PostgreSQL ofrece tipos de datos para el almacenamiento de direcciones de red. Estos tipos de datos permiten que las aplicaciones de administración de redes utilicen bases de datos de PostgreSQL como al-

macenes de datos. Para aquéllos familiarizados con las redes de computadoras, se ofrece a continuación un breve resumen de esta característica. Hay tipos diferentes para direcciones IPv4, direcciones IPv6 direcciones de Control de Acceso a Medios (Media Access Control, MAC) (*cidr*, *inet* y *macaddr*, respectivamente). Tanto el tipo *inet* como el tipo *cidr* pueden almacenar direcciones IPv4 y direcciones IPv6, con máscaras de subred opcionales. Su principal diferencia estriba en el formato de entrada y de salida, así como en la restricción de que las direcciones de enrutamiento de dominios de Internet sin clase (Classless Internet Domain Routing, CIDR) no aceptan valores con bits diferentes de cero a la derecha de la máscara de subred. El tipo *macaddr* se utiliza para almacenar direcciones MAC (generalmente, direcciones de hardware de tarjetas Ethernet). PostgreSQL soporta el indexado y la ordenación de estos tipos, así como un conjunto de operaciones (incluidas la comprobación de subredes y la asignación de direcciones MAC a nombres de fabricantes de hardware). Además, estos tipos ofrecen comprobación de errores de entrada. Por tanto, son preferibles a los campos de texto sencillo.

El tipo *bit* de PostgreSQL puede almacenar cadenas de unos y ceros tanto de longitud fija como de longitud variable. PostgreSQL soporta los operadores lógicos de bits y las funciones de manipulación de cadenas de caracteres.

26.3.2 Las reglas y otras características de las bases de datos activas

PostgreSQL soporta las restricciones y los disparadores de SQL (y los procedimientos almacenados; véase el Apartado 26.3.3). Además, ofrece reglas de reescritura de consultas que pueden declararse en el servidor.

PostgreSQL permite comprobar las restricciones, las restricciones de existencia y las restricciones de clave principal y de clave externa (con borrados restrictivos y en cascada).

Al igual que otros muchos sistemas de bases de datos relacionales, PostgreSQL soporta los disparadores, que resultan útiles para las restricciones no triviales y para comprobar o hacer cumplir la consistencia. Las funciones disparadoras pueden escribirse en lenguajes procedimentales como PL/pgSQL(véase el Apartado 26.3.3.4) o en C, pero no en mero SQL. Los disparadores pueden ejecutarse antes o después de las operaciones **insert**, **update** o **delete** y tanto una sola vez por fila modificada como una sola vez por instrucción SQL.

El sistema de reglas de PostgreSQL permite que los usuarios definan las reglas de reescritura de consultas en el servidor de bases de datos. A diferencia de los procedimientos almacenados y de los disparadores, el sistema de reglas interviene entre el analizador de consultas y el planificador, y modifica las consultas de acuerdo con el conjunto de reglas. Una vez transformado el árbol de la consulta original en uno o más árboles, éstos se pasan al planificador de consultas. Por tanto, el planificador tiene toda la información necesaria (las tablas que hay que explorar, las relaciones entre ellas, las calificaciones, la información conjunta, etc.) y puede presentar un plan de ejecución eficiente, aunque estén implicadas reglas complejas.

La sintaxis general para la declaración de las reglas es:

create rule *nombre_regla* **as**
    on { **select** | **insert** | **update** | **delete** }
    to *tabla* [**where** *calificación_regla*]
    do [**instead**] { **nothing** | *comando* | (*comando* ; *comando* ...) }

El resto de este apartado ofrece ejemplos que ilustran las posibilidades del sistema de reglas. En la documentación de PostgreSQL se pueden encontrar más detalles sobre el modo en que las reglas se adaptan a los árboles de consultas y la manera en que estos últimos se transforman posteriormente (véanse las notas bibliográficas). El sistema de reglas se implementa en la fase de reescritura del procesamiento de las consultas y se explica en el Apartado 26.6.1.

En primer lugar, PostgreSQL utiliza las reglas para implementar las vistas. La definición de una vista como:

create view *mivista* **as select** * **from** *mitabla*;

se transforma en la siguiente definición de regla:

```
create table mivista (la misma lista de columnas que en mitabla);
create rule return as on select to mivista do instead
    select * from mitabla;
```

Las consultas a *mivista* se transforman antes de su ejecución en consultas a la tabla subyacente *mitabla*. La sintaxis **create view** se considera una forma de programación mejor en este caso, ya que es más concisa y también evita la creación de vistas que hagan referencia unas a otras (lo que es posible si las reglas se declaran de manera poco cuidadosa, y puede dar lugar a errores de tiempo de ejecución que lleven a la confusión). No obstante, se pueden utilizar reglas para definir de manera explícita las acciones de actualización de las vistas (las instrucciones **create view** no permiten hacer esto).

Como ejemplo adicional, considérese el caso de que el usuario desee examinar las actualizaciones de las tablas. Algo así podría lograrse mediante una regla como:

```
create rule control_sueldos as on update to empleado
    where new.sueldo <> old.sueldo
    do insert into control_sueldos
    values (current_timestamp, current_user,
                    new.nombre_emp, old.sueldo, new.sueldo);
```

Finalmente, se ofrece una regla de inserción y actualización ligeramente más complicada. Supóngase que se almacenan los aumentos salariales pendientes en la tabla *aumentos_salariales(nombre_emp, aumento)*. Se puede declarar la tabla adicional *aumentos_aprobados* con los mismos campos y definir después la regla siguiente:

```
create rule aumentos_aprobados_insertar
    as on insert to aumentos_aprobados
    do instead
    update empleado
            set sueldo = sueldo + new.aumento
            where nombre_emp = new.nombre_emp;
```

Entonces la consulta siguiente:

```
insert into aumentos_aprobados select * from aumentos_salariales;
```

actualizará a la vez todos los sueldos de la tabla *empleado*.

El sistema de reglas de PostgreSQL puede utilizarse para implementar la mayor parte de los disparadores. Algunos tipos de restricciones (especialmente las claves externas) no se pueden implementar mediante reglas. Además, si la violación de una restricción genera un mensaje de error (en vez de descartar silenciosamente los valores no válidos, mediante la declaración de una regla "... **do instead nothing**"), entonces hay que utilizar disparadores. Los disparadores no pueden usarse para las acciones **update** ni **delete** sobre vistas. Dado que no hay ningún dato real en las relaciones de vistas, nunca se llamaría al disparador.

Finalmente, los disparadores se activan una vez por cada fila afectada. Las reglas, por su parte, manipulan el árbol de consultas antes de planificar la consulta. Por tanto, si una instrucción afecta a muchas filas, las reglas son más eficientes que los disparadores.

La implementación de los disparadores y de las restricciones en PostgreSQL se esboza brevemente en el Apartado 26.6.4.

26.3.3 Extensibilidad

Al igual que la mayor parte de los sistemas de bases de datos relacionales, PostgreSQL almacena la información sobre las bases de datos, las tablas, las columnas, etc., en lo que suele denominarse como **catálogos del sistema**, que se presentan ante el usuario como tablas normales. Otros sistemas de bases de datos relacionales se suelen ampliar modificando procedimientos incluidos en el código fuente o mediante la carga de módulos especiales de extensión escritos por el fabricante.

Sin embargo, a diferencia de la mayor parte de los sistemas de bases de datos relacionales, PostgreSQL va un paso más allá y almacena mucha más información en los catálogos: no sólo la información sobre las tablas y las columnas, sino también la información sobre los tipos de datos, las funciones, los métodos de acceso, etc. Por tanto, resulta sencillo para los usuarios ampliar PostgreSQL y esto facilita la creación rápida de prototipos de nuevas aplicaciones y de estructuras de almacenamiento. PostgreSQL también puede incorporar en el servidor código escrito por los usuarios, mediante la carga dinámica de los objetos compartidos. Esto ofrece un enfoque alternativo a la escritura de extensiones que puede utilizarse cuando las extensiones basadas en los catálogos no resultan suficientes.

Además, el módulo contrib de la distribución de PostgreSQL incluye numerosas funciones de usuario (por ejemplo, iteradores de *arrays*, comparación difusa de cadenas de caracteres, funciones criptográficas), tipos básicos por ejemplo, contraseñas cifradas, ISBN/ISSN, cubos n—dimensionales) y extensiones de los índices (por ejemplo, árboles RD, indexado de texto completo). Gracias a la naturaleza abierta de PostgreSQL hay una gran comunidad de profesionales y entusiastas de PostgreSQL que también amplían PostgreSQL prácticamente a diario. Los tipos de las extensiones son idénticos en funcionalidad a los tipos predefinidos (véase también el Apartado 26.3.1.2); los últimos simplemente se hallan ya enlazados en el servidor y están registrados previamente en el catálogo del sistema. De manera parecida, ésta es la única diferencia entre las funciones intrínsecas y las ampliadas.

26.3.3.1 Tipos

PostgreSQL permite que los usuarios definan tipos compuestos, así como que amplíen los tipos básicos disponibles.

Las definiciones de los tipos compuestos son parecidas a las definiciones de las tablas (de hecho, la última lleva a cabo la primera de manera implícita). Los tipos compuestos independientes suelen resultar útiles para los argumentos de las funciones. Por ejemplo, la definición

create type *t_ciudad* **as** (*nombre* **varchar**(80), *provincia* **char**(2))

permite que las funciones acepten y devuelvan las tuplas *t_ciudad*, aunque no haya ninguna tabla que contenga de manera explícita filas de este tipo.

La adición de tipos básicos a PostgreSQL es directa; se puede encontrar un ejemplo en complex.sql y complex.c en los tutoriales de la distribución de PostgreSQL. Los tipos básicos pueden declararse en C, por ejemplo:

```
typedef struct Complejo {
    double x;
    double y;
} Complejo;
```

A continuación el usuario tiene que definir las funciones para leer y escribir los valores del nuevo tipo en formato de texto (véase el Apartado 26.3.3.2). Posteriormente, el nuevo tipo puede registrarse empleando la instrucción:

```
create type complejo {
    internallength = 16,
    input = complejo_entrada,
    output = complejo_salida,
    alignment = double
};
```

suponiendo que las funciones de E/S de texto se hayan registrado como *complejo_entrada* y *complejo_salida*.

El usuario también tiene la posibilidad de definir funciones de E/S binarias (para un volcado de datos más eficiente). Los tipos ampliados pueden utilizarse como los tipos básicos de PostgreSQL ya existentes. De hecho, la única diferencia es que los tipos ampliados se cargan y enlazan dinámicamente

en el servidor. Además, los índices pueden ampliarse fácilmente para que manejen los nuevos tipos básicos; véase el Apartado 26.3.3.3.

26.3.3.2 Funciones

PostgreSQL permite que los usuarios definan funciones que se almacenen y ejecuten en el servidor. También soporta la sobrecarga de funciones (es decir, se pueden declarar funciones que utilicen el mismo nombre pero con argumentos de tipos diferentes). Las funciones pueden escribirse como instrucciones de mero SQL. Además, se soportan varios lenguajes procedimentales (se tratan en el Apartado 26.3.3.4). Finalmente, PostgreSQL tiene una interfaz para programación de aplicaciones para añadir funciones escritas en C (que se explica en este apartado).

Las funciones definidas por los usuarios pueden escribirse en C (o en un lenguaje con convenios de llamada compatibles, como C++). Los convenios de codificación reales son básicamente los mismos para las funciones definidas por los usuarios que se cargan de manera dinámica y para las funciones internas (que se enlazan en el servidor de manera estática). Por tanto, la biblioteca de funciones interna estándar es un fuente abundante de ejemplos de codificación para las funciones de C definidas por los usuarios. Una vez que se haya creado la biblioteca compartida que contiene la función, una declaración como la siguiente la registra en el servidor:

```
create function complejo_salida(complejo)
    returns cstring
    as 'nombre_archivo_objeto_compartido'
    language C immutable strict;
```

Se da por supuesto que el punto de entrada al archivo del objeto compartido es el nombre de la función de SQL (aquí, *complejo_salida*), a menos que se especifique lo contrario.

El ejemplo siguiente continúa el del Apartado 26.3.3.1. La interfaz de programación de aplicaciones oculta la mayor parte de los detalles internos de PostgreSQL. Por tanto, el código C para la función de salida de texto anterior de valores *complejo* es bastante sencillo:

```
pg_function_info_v1(complejo_salida);
Datum complejo_salida(pg_function_args) {
    Complejo *complejo = (Complejo *) pg_getarg_pointer(0);
    char *resultado;
    resultado = (char *) palloc(100);
    snprintf(resultado, 100, "(%g,%g)", complejo->x, complejo->y);
    pg_return_cstring(resultado);
}
```

Las funciones agregadas de PostgreSQL operan actualizando los **valores de estado** mediante funciones de **transición de estado** que se llaman para cada valor de la tupla del grupo de agregación. Por ejemplo, el estado del operador **avg** consta de la suma y recuento de los valores. A medida que llega cada tupla, la función de transición simplemente debe añadir su valor a la suma y aumentar en uno el recuento. De manera opcional, se puede llamar a una función *final* para que calcule el valor de retorno de acuerdo con la información de estado. Por ejemplo, la función final para **avg** simplemente dividiría la suma ejecutada entre el recuento y devolvería el resultado.

Por tanto, la definición de un operador nuevo es tan sencilla como la definición de esas dos funciones. Para el ejemplo del tipo *complejo*, si *complejo_suma* es una función definida por los usuarios que toma dos argumentos complejos y devuelve su suma, entonces el operador de agregación **sum** puede ampliarse a los números complejos mediante la declaración siguiente:

```
create aggregate sum (
    sfunc = suma_complejo,
    basetype = complejo,
    stype = complejo,
    initcond = '(0,0)'
);
```

Obsérvese el empleo de la sobrecarga de funciones: PostgreSQL llamará a la función de agregación *sum* adecuada en función del tipo real del argumento que se invoque. *Basetype* es el tipo de argumento y *stype* es el tipo del valor de estado. En este caso, no hace falta ninguna función final, ya que el valor devuelto es el propio valor de estado (es decir, la suma total en ambos casos).

Las funciones definidas por los usuarios también pueden invocarse mediante la sintaxis de los operadores. Más allá del mero "azúcar sintáctico" para la invocación de funciones, las declaraciones de los operadores también pueden ofrecer sugerencias al optimizador de consultas para que optimice el rendimiento. Estas sugerencias pueden incluir información sobre la conmutatividad, la restricción y la estimación de la selectividad de las uniones, así como otras propiedades relacionadas con los algoritmos de unión.

26.3.3.3 Extensiones de los índices

PostgreSQL soporta los índices habituales de árbol B y asociativos, así como los índices de árbol R (para los objetos espaciales bidimensionales) y los índices GiST genéricos (que son exclusivos de PostgreSQL y se explican en el Apartado 26.5.2.1). Todos ellos pueden extenderse fácilmente para incluir nuevos tipos básicos.

La adición de extensiones de los índices para un tipo dado exige la definición de una **clase de operador** que encapsule lo siguiente:

- **Estrategias para métodos de indexado.** Se trata de un conjunto de operadores que pueden utilizarse como calificadores en las cláusulas **where**. El conjunto concreto depende del tipo de índice. Por ejemplo, los índices de árbol B pueden recuperar rangos de objetos, por lo que el conjunto consiste en cinco operadores ($<$, $<=$, $=$, $>=$ y $>$), todos los cuales pueden aparecer en las cláusulas **where** que afecten a índices de árbol B. Los índices asociativos sólo permiten comprobar las igualdades y los índices de árbol R permiten varias relaciones espaciales (por ejemplo, contenido, a la izquierda, etc.).
- **Rutinas de soporte de los métodos de indexado.** El conjunto de operadores anterior no suele ser suficiente para la operación del índice. Por ejemplo, los índices asociativos necesitan una función que calcule el valor asociativo de cada objeto. Los índices de árbol R necesitan poder calcular las intersecciones y las uniones, y estimar el tamaño de los objetos indexados.

Por ejemplo, si se definen las funciones y operadores siguientes para comparar la magnitud de los números *complejo* (véase el Apartado 26.3.3.1), entonces se puede hacer que esos objetos sean indexables mediante la declaración siguiente:

```
create operator class ops_abs_complejo
    default for type complejo using btree as
            operator 1 <(complejo, complejo),
            operator 2 <= (complejo, complejo),
            operator 3 = (complejo, complejo),
            operator 4 >= (complejo, complejo),
            operator 5 > (complejo, complejo),
            function 1 cmp_abs_complejo(complejo, complejo);
```

Las instrucciones **operator** definen los métodos estratégicos y las instrucciones **function** definen los de soporte.

26.3.3.4 Lenguajes procedimentales

Las funciones y los procedimientos almacenados pueden escribirse en varios lenguajes procedimentales. Además, PostgreSQL define una interfaz para programación de aplicaciones con objeto de aprovechar cualquier lenguaje de programación con esta finalidad. Los lenguajes de programación pueden registrarse a petición de los usuarios y pueden ser **dignos de confianza (trusted)** o **no dignos de confianza (untrusted)**. Estos últimos permiten un acceso ilimitado al SGBD y al sistema de archivos, y escribir en ellos las funciones almacenadas exige privilegios de superusuario.

- **PL/pqSQL**. Se trata de un lenguaje digno de confianza que añade a SQL posibilidades de programación procedimental (por ejemplo, variables y control de flujo). Es muy parecido a PL/pqSQL de Oracle. Aunque no se puede transferir el código tal cual entre uno y otro, la portabilidad suele resultar sencilla.
- **PL/Tcl**, **PL/Perl** y **PL/Python**. Estos lenguajes aprovechan la potencia de TcI, Perl y Python para escribir en el servidor funciones y procedimientos almacenados. Los dos primeros están disponibles tanto en versión digna de confianza como en versión no digna de confianza (PL/Tcl, PL/Perl y PL/TclU, PL/PerlU, respectivamente), mientras que PL/Python no es digno de confianza en el momento de escribir este libro. Cada uno de ellos tiene enlaces que permiten el acceso al sistema de bases de datos mediante una interfaz específica de ese lenguaje.

26.3.3.5 Interfaz de programación para servidores

La interfaz de programación para servidores (Server Programming Interface, SPI) es una interfaz para programadores de aplicaciones que permite que las funciones de C definidas por los usuarios (véase el Apartado 26.3.3.2) ejecuten comandos SQL arbitrarios dentro de las funciones. Esto ofrece a los escritores de funciones definidas por los usuarios la posibilidad de implementar en C sólo las partes fundamentales y de aprovechar con facilidad toda la potencia del motor de bases de datos relacionales para hacer la mayor parte del trabajo.

26.4 Gestión de transacciones en PostgreSQL

El control de concurrencia de PostgreSQL implementa tanto el control de concurrencia de varias versiones (Multiversion Concurrency Control, MVCC) como el bloqueo de dos fases. El protocolo empleado depende del tipo de instrucción en ejecución. Para las instrucciones LMD[1] se emplean esquemas MVCC parecidos al que se presenta en el Apartado 16.5.1. El control de concurrencia para las instrucciones LDD, por su parte, se basa en el bloqueo de dos fases estándar.

26.4.1 Control de concurrencia en PostgreSQL

Dado que los detalles del protocolo MVCC de PostgreSQL dependen del *nivel de aislamiento* solicitado por la aplicación, se comenzará con una visión general de los niveles de aislamiento que ofrece PostgreSQL. Luego se describirán las ideas principales subyacentes al esquema MVCC de PostgreSQL, a lo que seguirá la discusión de su implementación en el MVCC de PostgreSQL y algunas implicaciones en relación con la gestión del almacenamiento por PostgreSQL, el diseño de las aplicaciones de usuario para PostgreSQL y el rendimiento de las bases de datos de PostgreSQL. Este apartado concluirá con una visión general de los bloqueos para las instrucciones LDD y una discusión del control de concurrencia para los índices.

26.4.1.1 Niveles de aislamiento en PostgreSQL

La norma de SQL define tres niveles de consistencia débiles, además del nivel de consistencia secuenciable, en el que se basa la mayor parte del estudio en este libro. La finalidad de ofrecer los niveles de consistencia débil es permitir un mayor grado de concurrencia para las aplicaciones que no necesiten

1. Las instrucciones LMD son instrucciones que actualizan o leen los datos de una tabla, es decir, **select**, **insert**, **update**, **fetch** y **copy**. Las instrucciones LDD afectan a toda la tabla, pueden eliminarla o cambiar su esquema, por ejemplo. Las instrucciones DDL y algunas otras instrucciones exclusivas de PostgreSQL se estudiarán más adelante en este apartado.

Nivel de aislamiento	*Lectura sucia*	*Lectura irrepetible*	*Fantasma*
Lectura no comprometida	Posible	Posible	Posible
Lectura comprometida	Posible	Posible	Posible
Lectura repetida	Posible	Posible	Posible
Secuenciable	No	No	No

Figura 26.5 Definición de los cuatro niveles de aislamiento estándar de SQL.

las sólidas garantías que ofrece la secuencialidad. Entre los ejemplos de estas aplicaciones están las transacciones duraderas que recopilan estadísticas de la base de datos y cuyos resultados no tienen por qué ser muy precisos.

La norma SQL define los diferentes niveles de aislamiento en términos de tres fenómenos que violan la secuencialidad. Esos tres fenómenos se denominan *lectura irrepetible*, *lectura sucia* y *lectura fantasma*, y se definen de la manera siguiente:

- **Lectura irrepetible.** Una transacción lee dos veces el mismo objeto durante la ejecución y encuentra un valor diferente en la segunda lectura, aunque la transacción no haya cambiado ese valor mientras tanto.
- **Lectura sucia.** La transacción lee valores escritos por otra transacción que todavía no se ha comprometido.
- **Lectura fantasma.** Una transacción vuelve a ejecutar una consulta que devuelve un conjunto de filas que satisfacen una condición de búsqueda y descubre que el conjunto de filas que satisfacen la condición ha cambiado como consecuencia de otra transacción comprometida recientemente (se puede hallar una explicación más detallada de este fenómeno en el Apartado 16.7.3).

Debería resultar evidente que cada uno de los fenómenos anteriores viola el aislamiento de las transacciones y, por tanto, violan la secuencialidad. La Figura 26.5 muestra la definición de los cuatro niveles de aislamiento de SQL especificados en la norma SQL: lectura no comprometida, lectura comprometida, lectura repetible y lectura secuenciable—en relación con estos fenómenos. PostgreSQL soporta dos de los cuatro niveles de aislamiento, la lectura comprometida y la lectura secuenciable.

26.4.1.2 Control de concurrencia para los comandos LMD

La idea principal que subyace a MVCC es conservar diferentes versiones de cada fila, que corresponden a diferentes instancias de esa fila en diferentes momentos. El protocolo MVCC se asegura de que cada transacción sólo vea las versiones de los datos que sean consistentes con la vista de la base de datos que tiene la transacción. Cada transacción ve una instantánea de los datos, que consta únicamente de los datos que se comprometieron en el momento en que se inició la transacción[2]. Esta instantánea no es necesariamente igual al estado actual de los datos.

La razón de emplear MVCC es que los lectores nunca bloquean a los escritores, y viceversa. Los lectores tienen acceso a la versión más reciente de las filas que forman parte de la instantánea de la transacción. Los escritores crean su propia copia independiente de la fila que se va a actualizar. El Apartado 26.4.1.3 muestra que el único conflicto que hace que se bloquee una transacción surge si dos escritores intentan actualizar la misma fila. Por el contrario, con el enfoque estándar de bloqueo de dos fases, puede que se bloqueen tanto los lectores como los escritores, ya que sólo hay una versión de cada objeto de datos y tanto las operaciones de lectura como las de escritura deben obtener un bloqueo antes de tener acceso a los datos.

MVCC de PostgreSQL es muy parecido en espíritu al esquema de ordenación de marcas temporales multiversión que se describe en el Apartado 16.5.1. PostgreSQL no emplea nunca bloqueos para los comandos LMD de manera explícita y, por tanto, no es necesaria ninguna interacción con el administrador de bloqueos. Esto se diferencia del esquema del MVCC empleado por Oracle, el único sistema

2. Además, las transacciones con varias consultas también ven los datos de consultas anteriores de la misma transacción.

comercial de bases de datos que utiliza MVCC en lugar del bloqueo de dos fases. El esquema del MVCC de Oracle es, básicamente, el protocolo de bloqueo de dos fases multiversión que se describe en el Apartado 16.5.2.

26.4.1.3 Implementación de MVCC en PostgreSQL

En el corazón de MVCC de PostgreSQL se halla el concepto de *visibilidad de tuplas*. Las tuplas de PostgreSQL hacen referencia a versiones de las filas. La visibilidad de tuplas define la versión válida de las potencialmente numerosas versiones de cada fila de una tabla en el contexto de una instrucción o transacción dada.

Una tupla es visible para una transacción T_A si se cumplen las dos condiciones siguientes:

1. La tupla fue creada por una transacción T_B que comenzó a ejecutarse y se comprometió antes de que la transacción T_A comenzara a ejecutarse.
2. Las actualizaciones de la tupla (si es que se ha llevado a cabo alguna) las ejecutó una transacción T_C que
   - se abortó *o*
   - comenzó a ejecutarse después de la transacción T_A *o*
   - estaba en proceso al comienzo de T_A.

El objetivo de las condiciones anteriores es asegurarse de que cada transacción sólo tenga acceso a los datos que se comprometieron en el momento en que la transacción comenzó a ejecutarse. PostgreSQL mantiene las siguientes estructuras de datos para comprobar eficientemente estas condiciones:

- Se asigna un *identificador de transacción*, que sirve simultáneamente como marca de tiempo, en el momento del comienzo de la transacción. PostgreSQL utiliza un contador lógico (como se describe en el Apartado 16.2.1) para la asignación de los identificadores de transacciones.
- Un archivo de registro denominado *pg_clog* contiene el estado actual de cada transacción. El estado puede ser en progreso, comprometida o abortada.
- Cada tupla de una tabla tiene una cabecera con tres campos: *xmin*, que contiene el identificador de la transacción que ha creado la tupla y que, por tanto, se denomina también *identificador de transacción creadora*; *xmax*, que contiene el identificador de transacción de la transacción de sustitución o eliminación (o *null*, si no se elimina o sustituye) y que también se conoce como *identificador de transacción expiradora*; y un enlace adelantado a nuevas versiones de la misma fila lógica, si es que hay alguna.
- Se crea una estructura de datos *SnapshotData* bien en el momento inicial de la transacción, bien en el momento inicial de la consulta, según el nivel de aislamiento (que se describe con más detalle más adelante). La estructura de datos *SnapshotData* contiene, entre otras cosas, una lista de todas las transacciones activas en el momento en que se toma la instantánea.

La Figura 26.6 ilustra estas estructuras de datos mediante un ejemplo sencillo que implica a una base de datos con una sola tabla, la tabla *sucursal* de la Figura 2.3. La tabla *sucursal* contiene tres atributos, el nombre de la sucursal, la ciudad en la que está ubicada y los activos de la sucursal. La Figura 26.6 muestra un fragmento de la tabla *sucursal* que sólo contiene (las versiones de) la fila de la sucursal de "Galapagar". Las cabeceras de las tuplas indican que la fila la creó originalmente la transacción 100, y que posteriormente la actualizaron las transacciones 102 y 106. La Figura 26.6 muestra también un fragmento del archivo de registro *pg_log*. Según el archivo *pg_log*, las transacciones 100 y 102 quedan comprometidas, mientras que las transacciones 104 y 106 se hallan procesándose.

Dadas las estructuras de datos anteriores, las dos condiciones que hay que satisfacer para que una tupla resulte visible pueden reescribirse de la manera siguiente:

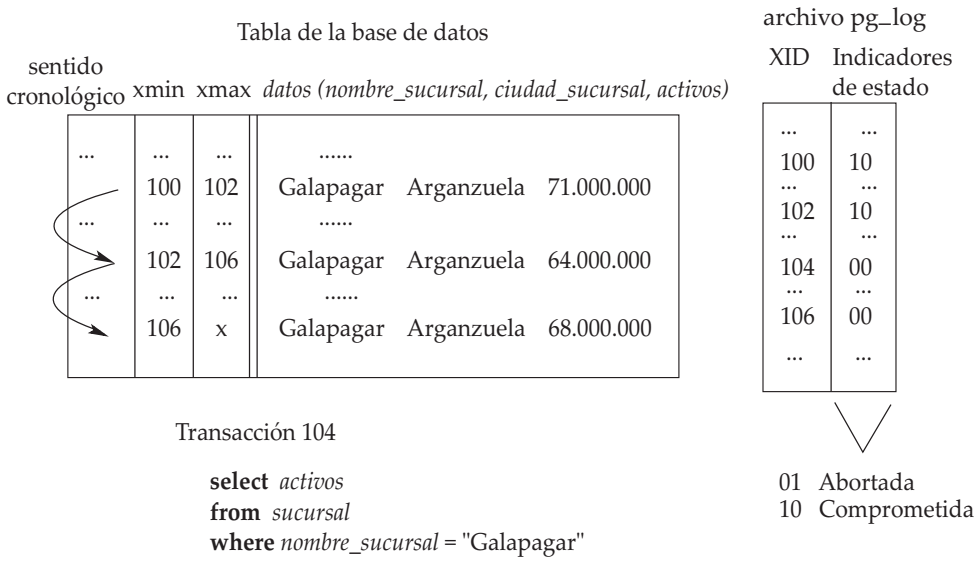


Figura 26.6 Las estructuras de datos de PostgreSQL empleadas para MVCC

1. El identificador de transacción creadora de la cabecera de la tupla
   - es una transacción comprometida de acuerdo con el archivo *pg_log y*
   - es menor que el contador de transacciones almacenado al comienzo de la consulta en *SnapshotData y*
   - no estaba procesándose al comienzo de la consulta según *SnapshotData*.
2. El identificador de transacción expiradora
   - está en blanco o abortado *o*
   - es mayor que el contador de transacciones almacenado al comienzo de la consulta en *SnapshotData o*
   - estaba procesándose al comienzo de la consulta según *SnapshotData*.

Por ejemplo, la única versión de la fila correspondiente a la sucursal de "Galapagar" que es visible para la transacción 104 de la Figura 26.6 es la segunda versión de la tabla, creada por la transacción 102. La primera versión, creada por la transacción 100 no es visible, dado que viola la condición 2: el identificador de transacción expiradora de esta tupla es 102, que corresponde a una transacción que no está abortada y que tiene un identificador de transacción menor que la transacción 104. La tercera versión de la sucursal "Galapagar" no es visible, dado que la creó la transacción 106, que tiene un identificador de transacción mayor que la transacción 104, lo que implica que esta versión no se había comprometido en el momento en que la transacción 104 comenzó a ejecutarse. Además, la transacción 106 sigue procesándose, lo que viola otra de las condiciones. La segunda versión de la fila cumple todas las condiciones de visibilidad de las tuplas.

Los detalles del modo en que el MVCC de PostgreSQL interactúa con la ejecución de las instrucciones de SQL dependen de si la instrucción es **insert**, **select, update** o **delete**. El caso más sencillo es el de la instrucción **insert**. A diferencia del caso de los bloqueos de dos fases, para las instrucciones **insert** no hay esencialmente ninguna interacción con el protocolo de control de concurrencia durante la ejecución de la instrucción: las instrucciones **insert** se limitan a crear una nueva tupla basada en los datos de la instrucción, inicializar la cabecera de la tupla (el identificador de creación) e insertar la nueva tupla en la tabla.

Cuando el sistema ejecuta una instrucción **select**, **update** o **delete**, la interacción con el protocolo MVCC depende del nivel de aislamiento especificado por la aplicación. Si el nivel de aislamiento es

lectura comprometida, el procesamiento de una nueva instrucción comienza por la creación de una nueva estructura de datos *SnapshotData* (independientemente de si la instrucción comienza una nueva transacción o forma parte de otra ya existente). A continuación, el sistema identifica las tuplas *objetivo*; es decir, las tuplas que son visibles con respecto a *SnapshotData* y cumplen los criterios de búsqueda de la instrucción. En el caso de las instrucciones **select**, el conjunto de tuplas objetivo constituye el resultado de la consulta.

En el caso de las instrucciones **update** o **delete**, se necesita un paso adicional tras identificar las tuplas objetivo, antes de que pueda tener lugar la verdadera operación de actualización o de eliminación. El motivo es que la visibilidad de las tuplas sólo garantiza que la tupla haya sido creada por una transacción comprometida antes del inicio de la instrucción **update** / **delete**. No obstante, es posible que, desde el inicio de la consulta, la tupla haya sido actualizada o eliminada por otra transacción que se esté ejecutando de manera concurrente. Esto puede detectarse examinando el identificador de transacción expiradora de la tupla. Si el identificador de transacción expiradora corresponde a una transacción que todavía esté ejecutándose, es necesario esperar a que se complete esa transacción. Si la transacción se aborta, la instrucción **update** o **delete** puede seguir adelante y llevar a cabo la verdadera operación de actualización o de eliminación. Si la transacción se compromete, el criterio de búsqueda de la instrucción **update** / **delete** debe volver a evaluarse y, sólo si la tupla sigue cumpliendo los criterios, puede llevarse a cabo la actualización o la eliminación. La realización de la operación de actualización o de eliminación incluye la creación de una nueva tupla (con su correspondiente cabecera que contendrá el identificador de creación) y también la actualización de la información del cabecera de la tupla vieja (es decir, el identificador de transacción expiradora).

Volviendo al ejemplo de la Figura 26.6, la transacción 104, que consiste sólo en una instrucción **select**, identifica la segunda versión de la fila Galapagar como tupla objetivo y la devuelve de manera inmediata. Si, en vez de eso, la transacción 104 fuera una instrucción de actualización, por ejemplo, que intentara incrementar los activos de la sucursal de Galapagar en algún importe, tendría que esperar a que la transacción 106 se completara. Luego volvería a evaluar la condición de búsqueda y, sólo si siguiera cumpliéndose, seguiría adelante con la actualización.

Emplear el protocolo descrito anteriormente para las instrucciones **update** y **delete** sólo proporciona el nivel de aislamiento de lectura comprometida. La secuencialidad puede violarse de varias maneras. En primer lugar, son posibles lecturas no repetibles. Dado que cada consulta de una transacción comienza con una nueva instantánea, una consulta de la transacción podría ver el efecto de las transacciones completadas entretanto que no fueran visibles a consultas anteriores de la misma transacción. Siguiendo la misma línea de pensamiento, son posibles las lecturas fantasmas. Además, una consulta **update** puede ver los efectos de las actualizaciones concurrentes llevadas a cabo por otras consultas en la misma fila, pero no ve el efecto de esas consultas concurrentes en otras filas de la base de datos.

Con objeto de proporcionar el nivel de aislamiento secuenciable de PostgreSQL, MVCC de PostgreSQL elimina las violaciones de la secuencialidad de dos maneras: en primer lugar, cuando determina la visibilidad de las tuplas, todas las consultas de una misma transacción utilizan una instantánea del comienzo de la transacción, en lugar de usarla del comienzo de cada consulta. Así, las sucesivas consultas de una misma transacción siempre ven los mismos datos.

En segundo lugar, el modo en que se procesan las actualizaciones y las eliminaciones es diferente en el modo secuenciable y en el modo de lectura comprometida. Al igual que en el modo de lectura comprometida, las transacciones esperan tras identificar una fila objetivo visible que cumple la condición de búsqueda y está siendo actualizada o eliminada por otra transacción concurrente. Si la transacción concurrente que está ejecutando la actualización o la eliminación se aborta, la transacción que se halla en espera puede seguir adelante con su propia actualización. No obstante, si la transacción concurrente se compromete, no hay manera de que PostgreSQL garantice la secuencialidad de la transacción en espera. Por tanto, la transacción en espera retrocede y devuelve el mensaje de error "Can't serialize access due to concurrent update" (No se puede secuenciar el acceso debido a una actualización concurrente).

Depende de cada aplicación el manejo correcto de los mensajes de error como el anterior, abortando la transacción correspondiente y reiniciando toda la transacción desde el principio. Obsérvese que los retrocesos debidos a problemas de secuencialidad sólo son posibles para las instrucciones de actualización. Sigue cumpliéndose que las transacciones sólo de lectura no entran nunca en conflicto con otras transacciones.

26.4.1.4 Repercusiones del empleo de MVCC

El empleo del esquema MVCC de PostgreSQL tiene repercusiones en tres áreas diferentes: (1) Se le asigna carga extra al gestor de almacenamiento, ya que necesita mantener diferentes versiones de las tuplas; (2) hace falta extremar el cuidado al desarrollar aplicaciones concurrentes, ya que el MVCC de PostgreSQL puede generar sutiles, pero importantes, diferencias en el modo en que se comportan las transacciones concurrentes respecto a los sistemas en que se utiliza el bloqueo de dos fases estándar; (3) el rendimiento de PostgreSQL depende de las características de la carga de trabajo que se ejecute en él. Las repercusiones del MVCC de PostgreSQL se describen a continuación con más detalle.

La creación y almacenamiento de varias versiones de cada fila puede llevar a un consumo excesivo de almacenamiento. Para aliviar este problema, PostgreSQL libera periódicamente espacio mediante la identificación y la eliminaciones de versiones de las filas que ya no resultan necesarias. Esta funcionalidad se implementa mediante el comando **vacuum**. El comando **vacuum** se ejecuta en segundo plano como si fuera un demonio, pero también pueden invocarlo directamente los usuarios.

El comando **vacuum** ofrece diferentes modos de operación: el comando **vacuum** sencillo simplemente reclama el espacio ocupado por las filas que ya no se utilizan y deja ese espacio libre para reutilizarlo. Esta forma del comando puede operar en paralelo con la lectura y la escritura normales de las tablas. El comando **vacuum full** lleva a cabo un procesamiento más amplio, que incluye el traslado de tuplas de unos bloques a otros para intentar compactar las tablas en un número mínimo de bloques de disco. Esta modalidad es mucho más lenta y exige el bloqueo exclusivo de cada tabla mientras se procesa. Cuando se invoca con el parámetro opcional **analyze**, **vacuum** reúne estadísticas del contenido de las tablas que está vaciando. El resultado se utiliza luego para actualizar la tabla de sistema *pg_statistic*, lo que permite que el planificador de consultas de PostgreSQL realice mejores elecciones al planificar las consultas.

Debido al empleo del control de concurrencia multiversión en PostgreSQL, el traslado de aplicaciones desde otros entornos a PostgreSQL puede exigir algunas precauciones adicionales para garantizar la consistencia de los datos. Por ejemplo, considérese una transacción T_A que ejecuta una instrucción **select**. Dado que los lectores de PostgreSQL no bloquean los datos, los datos leídos y seleccionados por T_A pueden ser sobrescritos por otra transacción concurrente, T_B, mientras que T_A todavía se está ejecutando. En consecuencia, puede que parte de los datos que T_A devuelve ya no estén actualizados en el momento de completarse T_A. Puede que T_A devuelva filas que otras transacciones hayan modificado o eliminado mientras tanto. Para garantizar la validez de las filas en cada momento y protegerla de las transacciones concurrentes, las aplicaciones deben utilizar **select for update** o adquirir de manera explícita un bloqueo con el comando **lock table** correspondiente.

El enfoque que PostgreSQL da al control de concurrencia proporciona un rendimiento óptimo para las cargas de trabajo que contienen muchas más lecturas que actualizaciones ya que, en ese caso, las posibilidades de que dos actualizaciones entren en conflicto y den lugar al retroceso de una transacción son muy bajas.

26.4.1.5 Control de la concurrencia LDD

Los mecanismos de MVCC descritos en el apartado anterior no protegen a las transacciones de las operaciones que afectan a tablas completas como, por ejemplo, las transacciones que descartan una tabla o modifican su esquema. Para ello, PostgreSQL ofrece bloqueos explícitos que los comandos LDD se ven obligados a adquirir antes de comenzar su ejecución. Estos bloqueos siempre se basan en las tablas (en vez de basarse en las filas) y se adquieren y se liberan de acuerdo con el estricto protocolo de bloqueo de dos fases.

La Figura 26.7 muestra una lista de todos los tipos de bloqueos ofrecidos por PostgreSQL, los comandos que los utilizan y la compatibilidad con otros modos de bloqueo. Los nombres de los tipos de bloqueo suelen tener un origen histórico y no reflejan necesariamente el uso que se les da. Por ejemplo, todos los bloqueos son bloqueos del nivel de tabla, aunque algunos tengan en el nombre la palabra "fila". Los comandos LMD sólo adquieren bloqueos de los tipos 1, 2 y 3. Estos tres tipos de bloqueo son compatibles entre sí, ya que MVCC se preocupa de proteger esas operaciones unas de otras. Los comandos LDD sólo adquieren esos bloqueos para protegerse de comandos LDD.

Nombre del bloqueo	*Entra en conflicto con*	*Adquirido por*
De acceso compartido	De acceso exclusivo	consulta **select**
De fila compartido	Exclusivo	consulta **select for update**
De fila exclusivo	De actualización compartida exclusivo	consultas **update** **delete** **insert**
De actualización compartida exclusivo	De actualización compartida exclusivo De fila compartida exclusivo Exclusivo De acceso exclusivo	**vacuum**
Compartido	De fila exclusivo De actualización compartida exclusivo De fila compartida exclusivo De acceso exclusivo	**create index**
De fila compartida exclusivo	De fila exclusivo De actualización compartida exclusivo Compartido De fila compartida exclusivo De acceso exclusivo	—
Exclusivo	De fila compartido De fila exclusivo De actualización compartida exclusivo Compartido De fila compartida exclusivo De acceso exclusivo	—
De acceso exclusivo	Entra en conflicto con los bloqueos de todos los modos	**drop table** **alter table** **vacuum full**

Figura 26.7 Modos de bloqueo en el nivel de tabla.

Aunque el principal objetivo es proporcionar control interno de la concurrencia a PostgreSQL para los comandos LDD, las aplicaciones de PostgreSQL también pueden adquirir de manera explícita todos los bloqueos de la Figura 26.7 mediante el comando **lock table**.

Los bloqueos se registran en una tabla de bloqueos que se implementa como una tabla asociativa en memoria compartida que utiliza como claves el tipo y el identificador del objeto que se bloquea. Si una transacción desea adquirir un bloqueo sobre un objeto que otra transacción retiene en un modo no compatible, tiene que esperar hasta que se libere el bloqueo. Las esperas de bloqueo se implementan mediante semáforos. Cada transacción tiene asociado un semáforo. Cuando espera debido a un bloqueo, realmente es debido al semáforo asociado con la transacción que tiene el bloqueo. Una vez el titular del bloqueo lo libera, se lo comunica a la transacción que está esperando mediante el semáforo. Al implementar las esperas por bloqueo titular de bloqueo, en vez de hacerlo por objeto bloqueado, PostgreSQL necesita al menos un semáforo por cada transacción concurrente, en lugar de uno por cada objeto bloqueable.

La detección de interbloqueos en PostgreSQL se basa en tiempos límites. De manera predeterminada, la detección de interbloqueos se activa si una transacción ha estado esperando debido a un bloqueo más de un segundo. El algoritmo de detección de interbloqueos crea un gráfico de esperas en términos de la información de la tabla de bloqueos y busca en ese gráfico dependencias circulares. Si encuentra alguna, lo que significa que se ha detectado un interbloqueo, la transacción que desencadenó el interbloqueo se aborta y devuelve un error al usuario. Si no se detecta ningún ciclo, la transacción sigue esperando debido al bloqueo. A diferencia de algunos sistemas comerciales, PostgreSQL no ajusta el parámetro de tiempo límite de bloqueo, pero permite que el administrador lo ajuste manualmente. Lo ideal sería escoger este parámetro del orden del tiempo de vida de las transacciones, con objeto de optimizar el equilibrio entre el tiempo que se tarda en detectar un interbloqueo y el trabajo perdido al ejecutar el algoritmo de detección de interbloqueos cuando no se ha producido ninguno.

26.4.1.6 Bloqueos e índices

Los bloqueos realizados para el acceso a índices dependen del tipo de índice. Para GiST y para los árboles R se utilizan bloqueos sencillos del nivel de índices que se mantienen durante todo la duración del comando. Los accesos a índices asociados permiten más concurrencia mediante el empleo de bloqueos del nivel de páginas que se liberan una vez procesada la página. No obstante, los bloqueos del nivel de páginas para los índices asociados no están libres de interbloqueos. El tipo de índice recomendado para las aplicaciones con un grado elevado de concurrencia son los índices B, ya que ofrecen bloqueos de grano fino sin condiciones de interbloqueo: el acceso a los índices B se protege mediante bloqueos compartidos o exclusivos de corta duración del nivel de páginas. Los bloqueos se liberan en cuanto se captura o inserta cada tupla del índice.

26.4.2 Recuperación

Históricamente, PostgreSQL no usaba un registro histórico de escritura anticipada (Write–Ahead Logging, WAL) para la recuperación y, por tanto, no podía garantizar la consistencia en caso de caída. Las caídas pueden acabar dando lugar a estructuras de índice inconsistentes o, peor aún, a tablas con sus contenidos completamente corruptos, debido a las páginas de datos escritas parcialmente. En consecuencia, a partir de la versión 7.1, PostgreSQL emplea la recuperación estándar basada en registro histórico de escritura anticipada con una fase rehacer (redo) y una fase deshacer (undo) parecidas a las de ARIES. El Apartado 17.6.1 ofrece una introducción a estas fases y al concepto de registro histórico de escritura anticipada y el Apartado 17.8.6 ofrece una visión general de ARIES. La principal diferencia en la implementación de la recuperación en PostgreSQL es que el MVCC de PostgreSQL permite algunas simplificaciones en comparación con la recuperación estándar basada en el registro histórico.

En primer lugar, en PostgreSQL la recuperación no necesita deshacer los efectos de las transacciones abortadas. Las transacciones en proceso de aborto realizan una entrada en el archivo pg_clog que registra el hecho de que está en proceso de aborto. En consecuencia, todas las versiones de las filas que deja atrás no serán nunca visibles para las demás transacciones. El único caso en que este enfoque puede acabar dando lugar a problemas se produce cuando una transacción aborta debido a una caída del proceso PostgreSQL correspondiente y ese proceso no tiene la posibilidad de crear la entrada *pg_clog* antes de la caída. PostgreSQL maneja esto de la manera siguiente: siempre que una transacción comprueba el estado de otra en el archivo pg_clog y descubre que el estado es "en proceso", comprueba si la transacción se está ejecutando realmente en alguno de los procesos de PostgreSQL. Si ningún proceso de PostgreSQL está ejecutando la transacción, se deduce que el proceso correspondiente se ha caído y la entrada de pg _clog de la transacción se actualiza a "abortada".

En segundo lugar, la recuperación basada en registro histórico de escritura anticipada se simplifica debido a que MVCC de PostgreSQL ya realiza un seguimiento de parte de la información necesaria para el registro WAL. Más concretamente, no hace falta registrar el inicio, el compromiso y el aborto de las transacciones, ya que MVCC registra el estado de cada transacción en *pg_clog*.

26.5 Almacenamiento e índices

Al igual que con el resto de PostgreSQL, la filosofía de diseño de la disposición y el almacenamiento de los datos tiene dos objetivos gemelos de (1) una implementación sencilla y limpia y (2) la facilidad de administración. Las bases de datos gestionadas por los servidores de PostgreSQL las divide el administrador en *agrupaciones de bases de datos*, donde todos los datos y los metadatos asociados con cada agrupación se almacenan en el mismo directorio del sistema de archivos. A diferencia de los sistemas comerciales, PostgreSQL no soporta "espacios de tablas"[3] que permitirían que el administrador de la base de datos tuviera un control preciso de la ubicación de almacenamiento de cada objeto físico concreto. Además, PostgreSQL sólo soporta "sistemas de archivos cocinados", lo que excluye el empleo de particiones de disco sin formato.

Al igual que con el resto de PostgreSQL, la filosofía de diseño de la disposición y el almacenamiento de los datos tiene dos objetivos gemelos de (1) una implementación sencilla y limpia y (2) la facilidad

3. El desarrollo de los espacios de tablas se está llevando a cabo activamente y formará parte de versiones futuras.

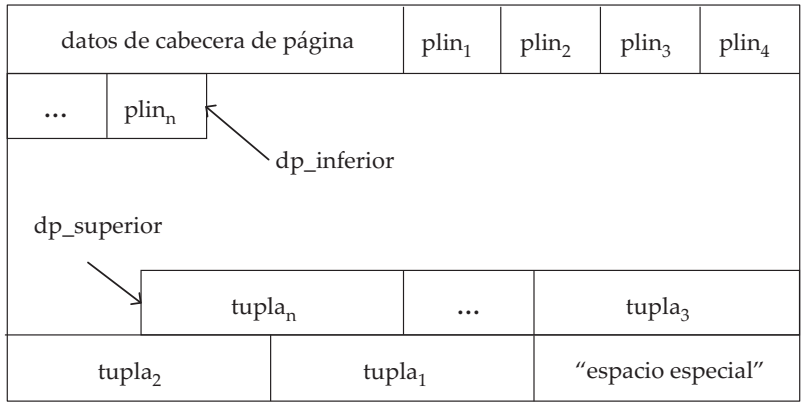


Figura 26.8 Formato de página con ranuras para las tablas de PostgreSQL.

de administración. Las bases de datos gestionadas por los servidores de PostgreSQL las divide el administrador en *agrupaciones de bases de datos*, donde todos los datos y los metadatos asociados con cada agrupación se almacenan en el mismo directorio del sistema de archivos. A diferencia de los sistemas comerciales, PostgreSQL no soporta "espacios de tablas"[4] que permitirían que el administrador de la base de datos tuviera un control preciso de la ubicación de almacenamiento de cada objeto físico concreto. Además, PostgreSQL sólo soporta "sistemas de archivos cocinados", lo que excluye el empleo de particiones de disco sin formato.

La simplicidad del diseño del sistema de almacenamiento de PostgreSQL puede crear algunas limitaciones en el rendimiento. La falta de soporte para los espacios de tablas limita las posibilidades de utilizar con eficiencia los recursos de almacenamiento disponibles, en especial varios discos que operen en paralelo. Además, el tamaño de bloque de todos los objetos de la base de datos está predeterminado (el valor predeterminado es de 8 KB y puede modificarse recompilando el código), lo que puede dar lugar a rendimientos por debajo de lo normal al tratar con medios de almacenamiento que manejen bloques de datos de mayor tamaño. El empleo de sistemas de archivos cocinados da lugar al empleo de memorias intermedias dobles, en las que cada bloque se captura primero del disco a la memoria intermedia del sistema de archivos (en el espacio del núcleo) antes de copiarlo al grupo de memorias intermedias de PostgreSQL.

Por otro lado, las empresas modernas utilizan cada vez más sistemas de almacenamiento como los medios de almacenamiento conectados en red y las redes de áreas de almacenamiento, en lugar de los discos conectados a los servidores. La filosofía, en este caso, es que el almacenamiento es un servicio que se administra y se ajusta con facilidad de manera independiente. El empleo de RAID para conseguir tanto paralelismo como almacenamiento redundante se explica en el Apartado 11.3.

Por tanto, la sensación de muchos desarrolladores de PostgreSQL es que, para una amplia mayoría de aplicaciones, y, en realidad, para el público de PostgreSQL, las limitaciones de rendimiento son mínimas y están justificadas por su facilidad de administración y de gestión, así como por la simplicidad de su implementación.

26.5.1 Tablas

La unidad principal de almacenamiento de PostgreSQL es la tabla. En PostgreSQL las tablas se almacenan en "archivos en montículo". Estos archivos utilizan una forma del formato estándar de "página con ranuras" que se describe en el Apartado 11.6.2.1. El formato PostgreSQL se muestra en la Figura 26.8. En cada página, a la cabecera le sigue un *array* de "punteros de línea". Cada puntero de línea guarda la posición (en relación con el comienzo de la página) y la longitud de una tupla concreta de la página. Las tuplas reales se almacenan en orden inverso de los punteros de línea desde el final de la página.

Cada registro de un archivo en montículo queda identificado por un **identificador de tupla** (tuple ID, TID). El TID consiste en un identificador de bloque de 4 bytes que especifica la página del archivo

4. El desarrollo de los espacios de tablas se está llevando a cabo activamente y formará parte de versiones futuras.

que contiene la tupla y un identificador de ranura de 2 bytes. El identificador de ranura es un índice del *array* de punteros de línea que, a su vez, se utiliza para tener acceso a la tupla.

Aunque esta infraestructura permite que las tuplas de una página se eliminen o se actualicen, en el enfoque de MVCC de PostgreSQL ninguna de esas operaciones se lleva a cabo realmente de modo físico durante el procesamiento normal. Cuando se *vacía* una página, sin embargo, las tuplas caducadas se eliminan físicamente, lo que hace que se formen agujeros en la página. El método indirecto de acceso a las tuplas mediante el *array* de punteros de línea permite la reutilización de dichos agujeros.

La longitud de cada tupla viene limitada normalmente por la de las páginas de datos. Esto hace difícil almacenar tuplas muy largas. Cuando PostgreSQL encuentra una tupla tan grande, intenta **comprimir** (*toast*) las columnas de gran tamaño. Los datos de las columnas **comprimidas** son sustituidos por un puntero que localiza una versión comprimida de los datos que se almacenan fuera de la página.

26.5.2 Índices

PostgreSQL soporta los diferentes tipos de índices, incluyendo los basados en métodos de acceso extensible del usuario. Aunque los métodos de acceso pueden usar un formato de página diferente, todos los índices disponibles en PostgreSQL usan el mismo formato de página con ranuras descrito en el Apartado 26.5.1.

26.5.2.1 Tipos de índices

PostgreSQL soporta los siguientes tipos de índices:

- **De árbol B.** El tipo de índice predeterminado es un método de árbol B que es una implementación de los árboles de alta concurrencia de Lehman–Yao (esto se explica con detalle en el Apartado 16.9). Estos índices resultan útiles para las consultas de igualdad y para las de rango sobre datos ordenables, y también para algunas operaciones de coincidencia de patrones como la expresión **like**.
- **De asociación.** Los índices asociativos de PostgreSQL son una implementación de la asociación lineal (para más información sobre los índices de asociación, véase el Apartado 12.6.3). Este tipo de índices resulta útil para las operaciones de igualdad sencillas. Se puede crear un índice asociativo con la siguiente instrucción LDD:

  create index *nombreind* **on** *nombretab* **using hash** (*nombrecol*)

  Se ha demostrado que los índices asociativos de PostgreSQL tienen un rendimiento de búsqueda no mejor que el de los árboles B, pero tienen un tamaño y unos costes de mantenimiento considerablemente mayores. Por tanto, casi siempre es preferible utilizar índices de árbol B a índices de asociación.
- **De árbol R.** Para operaciones como el *superposición* con los tipos de datos espaciales predefinidos (caja, círculo, punto, etc.), PostgreSQL ofrece índices de árbol R. Estos índices implementan el algoritmo de división cuadrática (Apartado 24.3.5.3). Los índices de árbol R pueden crearse con la siguiente instrucción LDD:

  create index *nombreind* **on** *nombretab* **using rtree** (*nombrecol*)

  La gestión de los datos espaciales se estudió en el Apartado 24.3, con información sobre los árboles R y otras técnicas de indexación.
- **GiST.** Finalmente, PostgreSQL soporta un cuarto tipo de índices basado en árboles extensibles denominado GiST, o árboles de búsqueda generalizados (Generalized Search Trees). GiST es un método de acceso equilibrado estructurado en árboles que puede utilizarse para implementar una familia completa de índices diferentes; por ejemplo, los índices de árbol B predeterminados y los índices de árbol R pueden implementarse mediante GiST. Los índices GiST facilitan a los

expertos en dominios que estén versados en un tipo de datos concreto (como los datos de imágenes) el desarrollo de índices que mejoren el rendimiento sin tener que tratar con los detalles internos del sistema de bases de datos. Entre los ejemplos de índices creados empleando GiST están la indexación de cubos multidimensionales y la indexación de texto completo para las consultas de recuperación de información. Véanse las notas bibliográficas para obtener referencias de más información sobre los índices GiST.

26.5.2.2 Otras variaciones de los índices

Para algunos de los tipos de índices que se han descrito PostgreSQL soporta más variaciones complejas como:

- **Índices de varias columnas.** Resultan útiles para conjuntos de predicados que abarcan varias columnas de una tabla. Los índices multicolumna sólo se soportan para índices de árbol B y para índices GiST.
- **Índices únicos.** Las restricciones de clave única y de clave principal pueden aplicarse mediante el empleo de índices únicos en PostgreSQL. Sólo se pueden definir como únicos los índices de árbol B.
- **Índices sobre expresiones.** En PostgreSQL es posible crear índices sobre expresiones escalares arbitrarias de las columnas de las tablas, y no sólo sobre columnas concretas. Esto resulta especialmente útiles cuando las expresiones en cuestión son "caras"; es decir, suponen un cálculo complicado definido por los usuarios. Un ejemplo es el soporte de comparaciones que no distinguen entre mayúsculas y minúsculas mediante la definición de un índice sobre la expresión *lower(columna)* y el empleo del predicado *lower(columna)* = 'valor' en las consultas. Un inconveniente es que los costes de mantenimiento de los índices sobre expresiones es elevado.
- **Clases de operadores.** Las funciones de comparación concretas empleadas para crear, mantener y utilizar los índices sobre columnas están ligadas al tipo de datos de cada columna. Cada tipo de datos tiene asociado una "clase de operador" predeterminada (que se describe en el Apartado 26.3.3.3) que identifica los operadores reales que se utilizarían normalmente. Aunque este operador predeterminado suele resultar suficiente para la mayor parte de los usos, puede que algunos tipos de datos posean varias clases "significativas". Por ejemplo, al tratar con los números complejos, puede que resulte preferible indexar el componente real o el imaginario. PostgreSQL ofrece algunas clases de operador para operaciones de comparación de patrones (como **like**) sobre datos de texto que no utilicen las reglas de comparación locales concretas.
- **Índices parciales.** Se trata de índices creados sobre un subconjunto de una tabla definido por un predicado. El índice sólo contiene entradas para tuplas que satisfagan el predicado. Los índices parciales resultan adecuados para casos en que una columna pueda contener gran número de apariciones de un número muy pequeño de valores. En esos casos, sin un índice parcial, las consultas del tipo "aguja en el pajar" que buscan un valor infrecuente acabarían examinando todo el índice. Los índices parciales que excluyen los valores frecuentes son pequeños y suponen menos operaciones E/S. Los índices parciales resultan menos costosos de mantener, ya que una fracción importante de las inserciones no participa en ellos.

26.6 Procesamiento y optimización de consultas

Cuando PostgreSQL recibe una consulta, ésta se divide primero en una representación interna que sufre una serie de transformaciones que dan lugar a un plan de consulta que el **ejecutor** utiliza para procesar la consulta.

26.6.1 Reescritura de consultas

La primera etapa de la transformación de una consulta es la **reescritura** (rewrite) y, es esta etapa la responsable del sistema de **reglas** (rules) del sistema de PostgreSQL. Como se explicó en el Aparta-

do 26.3.2, en PostgreSQL los usuarios pueden crear **reglas** que se activan ante diferentes eventos, como instrucciones **update**, **delete**, **insert** y **select**. El sistema implementa una vista mediante la conversión de la definición de una vista en una regla **select**. Cuando se recibe una consulta que implica una instrucción **select** sobre la vista, se desencadena la regla **select** para la vista y se reescribe la consulta utilizando la definición de la vista.

Las reglas se registran en el sistema utilizando el comando **create rule**, momento en el cual la información sobre la regla se almacena en el catálogo. Este catálogo se utiliza luego durante la reescritura de la consulta para descubrir todas las reglas candidatas para una consulta dada.

La fase de reescritura trabaja primero con todas las instrucciones **update, delete** e **insert** desencadenando todas las reglas adecuadas. Obsérvese que puede que tales instrucciones sean complejas y contengan cláusulas **select**. En consecuencia, se desencadenan todas las demás reglas que sólo contienen instrucciones **select**. Dado que el desencadenamiento de una regla puede hacer que la consulta se reescriba de una manera que exija que se desencadene otra regla, las reglas se comprueban repetidamente en cada forma de la consulta reescrita hasta que se alcanza un punto fijo y no hace falta desencadenar más reglas.

En PostgreSQL no hay reglas predeterminadas, sólo las definidas de manera explícita por los usuarios y de forma implícita por las definiciones de las vistas.

26.6.2 Planificación y optimización de consultas

Una vez reescrita la consulta, se somete a la fase de planificación y optimización. En esta fase cada bloque de la consulta se trata de manera aislada y se genera un plan específico para él. Esta planificación comienza de abajo arriba desde la subconsulta más interna de la consulta reescrita, hasta su bloque de consulta más externo.

El optimizador de PostgreSQL está, en su mayor parte, basado en costes. Lo ideal es generar un plan de acceso cuyo coste estimado sea mínimo. El modelo de costes incluye como parámetros tanto el coste de E/S de la captura no secuencial de páginas como los costes de CPU del procesamiento de tuplas de montículo, tuplas de índices y predicados sencillos.

El proceso real de optimización se basa en una de las dos formas siguientes:

- **Planificador estándar.** Se trata de un enfoque de optimización tradicional que se utilizaba en System R, un sistema relacional pionero desarrollado por los centros de investigación de IBM en los años 70. Es un algoritmo de programación dinámica en el que, para cada bloque, se enumeran y planifican todas las posibilidades de reunión de dos sentidos y se estima su coste de acceso. Luego se enumeran y estiman todas las posibilidades de reunión de tres sentidos, utilizando las mejores estimaciones de reunión de dos sentidos. Este proceso continúa hasta que se genera un "buen" plan para el bloque de consulta. En el Capítulo 14 se ofrecen más detalles sobre este enfoque.
- **Optimizador genético de consultas.** Cuando el número de tablas en un bloque de consulta es muy grande, el algoritmo de programación dinámica del Sistema R se vuelve muy costoso. A diferencia de otros sistemas comerciales que pasan de manera predeterminada a técnicas avaras o basadas en reglas, PostgreSQL utiliza un enfoque más radical: un algoritmo genético que inicialmente se desarrolló para resolver problemas de rutas de viajantes. Hay pruebas anecdóticas del empleo con éxito de la optimización genética de consultas en sistemas de producción para consultas con alrededor de 45 tablas.

Dado que el planificador opera de abajo arriba, puede llevar a cabo ciertas transformaciones en el plan de la consulta a medida que éste se genera. Un ejemplo es la frecuente transformación de subconsulta a reunión que aparece en muchos sistemas comerciales (generalmente implementada en la fase de reescritura). Cuando PostgreSQL encuentra una subconsulta no correlacionada (como puede ser la causada por una consulta sobre una vista), suele ser posible "elevar" la subconsulta planificada y mezclarla en el bloque de consulta del nivel superior. No obstante, las transformaciones que empujan las eliminaciones de duplicados hacia bloques de consulta de nivel inferior no suelen ser posibles en PostgreSQL.

La fase de optimización de consultas da lugar a un plan de consultas que es un árbol de operadores relacionales. Cada operador representa una operación concreta sobre uno o varios conjuntos de tuplas. Los operadores pueden ser unarios (por ejemplo, la reunión de bucle anidado) o n-arios (por ejemplo, la unión de conjuntos).

Resulta crucial para el modelo de costes una estimación precisa del número total de tuplas que se procesarán en cada operador del plan. Esto lo deduce el optimizador en función de las estadísticas que se tienen de cada relación del sistema. Estas estadísticas indican el número total de tuplas para cada relación e información específica sobre cada columna de una relación, como puede ser la cardinalidad de esa columna, una lista de los valores más frecuentes de la tabla y su número de apariciones, y un histograma que divida los valores de la columna en grupos de igual población. Además, PostgreSQL tiene también una correlación estadística entre las ordenaciones física y lógica en las filas de los valores de las columnas; esto indica el coste de las exploraciones de índices para recuperar tuplas que pasen predicados a la columna. El ABD debe garantizar que estas estadísticas estén al día mediante la ejecución periódica del comando **analyze**.

26.6.3 Ejecutor de consultas

El módulo ejecutor es responsable del procesamiento de los planes de consultas producidos por el optimizador. El ejecutor sigue el modelo **iterador** (iterator) con un conjunto de cuatro funciones implementadas para cada operador (open, next, rescan y close). Los iteradores también se estudian como parte de los cauces bajo demanda en el Apartado 13.7.2.1. Los iteradores de PostgreSQL tienen una función adicional, rescan, que se utiliza para reiniciar subplanes (por ejemplo, para el bucle interno de una reunión) con parámetros como pueden ser los rangos de las claves del índice.

Algunos de los operadores importantes del ejecutor pueden catalogarse de la siguiente manera:

1. **Métodos de acceso.** Los métodos de acceso reales que se utilizan en PostgreSQL para recuperar datos de objetos que se hallan en el disco son las exploraciones secuenciales desde el montículo y las exploraciones de índices.
   - **Exploraciones secuenciales.** Las tuplas de cada relación se exploran secuencialmente desde el primer bloque del archivo hasta el último. Cada tupla se devuelve al invocador sólo si es "visible" de acuerdo con las reglas de aislamiento de transacciones del Apartado 26.4.1.3.
   - **Exploraciones de índices.** Dado un rango de índices (o una clave concreta, en el caso de un índice de montículo), este método de acceso devuelve un conjunto de tuplas coincidentes del archivo de montículo asociado. No hay nada previsto para separar el acceso a los identificadores de las tuplas desde el índice y la captura de las tuplas reales. Esto evita la ordenación de los TIDs y garantiza que se tenga acceso al archivo de montículo de manera secuencial, lo que minimiza el número de operaciones de captura de páginas.

2. **Métodos de reunión.** PostgreSQL soporta tres métodos de reunión: reuniones por mezcla ordenadas, reuniones de bucle anidado (incluidas las variantes de índice de bucle anidado para el interior) y una reunión híbrida asociativa.

3. **Ordenación.** La ordenación externa se implementa en PostgreSQL mediante algoritmos que se explican en el Apartado 13.4. Los datos de entrada se dividen en secuencias ordenadas que se mezclan en una mezcla polifásica. Aunque las secuencias iniciales se formen mediante selección por sustitución, se utiliza un árbol de prioridades en lugar de una estructura de datos para fijar el número de registros que se guardan en la memoria. Esto se debe a que PostgreSQL puede trabajar con tuplas que varían considerablemente en tamaño e intenta garantizar la utilización completa del espacio de memoria configurado para la ordenación.

4. **Agregación.** La agregación agrupada en PostgreSQL puede basarse en la ordenación o en la asociación. Cuando el número estimado de grupos diferentes es muy grande, se utiliza la primera y, en caso contrario, se prefiere el enfoque basado en la asociación.

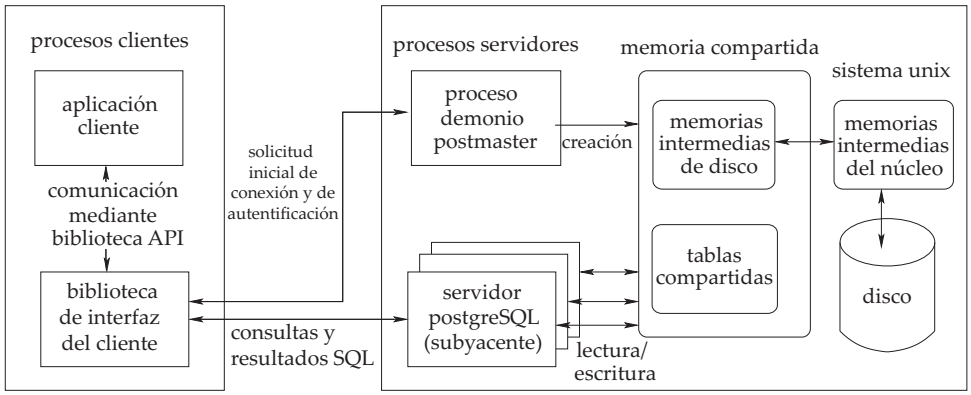


Figura 26.9 La arquitectura del sistema PostgreSQL.

26.6.4 Disparadores y restricciones

En PostgreSQL (a diferencia de algunos sistemas comerciales) las características de las bases de datos activas como los disparadores y las restricciones no se implementan en la fase de reescritura. Por el contrario, se implementan como parte del ejecutor de consultas. Cuando el usuario registra los disparadores y las restricciones, los detalles se asocian con la información del catálogo para cada relación e índice adecuados. El ejecutor procesa las instrucciones **update**, **delete** e **insert** generando repetidamente modificaciones en las tuplas para la relación. Por cada una de esas modificaciones (una operación de actualización, eliminación o inserción) el ejecutor comprueba de manera explícita los disparadores y las restricciones candidatos, y los desencadena o aplica, antes o después de la modificación, según se requiera.

26.7 Arquitectura del sistema

La arquitectura del sistema PostgreSQL sigue el modelo de proceso por transacción. Los sitios PostgreSQL en funcionamiento son gestionados por un proceso coordinador central, denominado **postmaster**. El proceso postmaster es responsable de inicializar y cerrar el servidor y también de manejar las solicitudes de conexión de nuevos clientes . El postmaster asigna cada nuevo cliente que se conecta a un proceso servidor en segundo plano que es responsable de ejecutar las consultas en nombre del cliente y de devolver los resultados al cliente. Esta arquitectura se describe en la Figura 26.9.

Las aplicaciones clientes pueden conectarse con el servidor de PostgreSQL y remitir consultas mediante alguna de las muchas interfaces de programación de aplicaciones de la base de datos soportadas por PostgreSQL (libpq, JDBC, ODBC, Perl DBD) que se proporcionan como bibliotecas para la parte cliente. Un ejemplo de aplicación cliente es el programa de línea de comandos psql, que se incluye en la distribución PostgreSQL estándar. El postmaster es responsable de manejar las conexiones iniciales de los clientes. Para ello, escucha constantemente si hay nuevas conexiones en un puerto conocido. Tras llevar a cabo los pasos de inicialización, como la autenticación del usuario, el postmaster genera un nuevo proceso de servidor en segundo plano para manejar el nuevo cliente. Tras esta conexión inicial, el cliente sólo interactúa con el proceso servidor en segundo plano, remitiendo consultas y recibiendo resultados de consultas. Ésta es la esencia del modelo de proceso por conexión adoptado por PostgreSQL.

El proceso servidor en segundo plano es responsable de ejecutar las consultas remitidas por el cliente mediante la realización de los pasos de ejecución de consultas necesarios, incluidos el análisis, la optimización y la ejecución. Cada proceso de servidor en segundo plano sólo puede manejar una única consulta a la vez. Para ejecutar más de una consulta en paralelo la aplicación debe establecer varias conexiones con el servidor.

En cualquier momento puede haber varios clientes conectados al sistema y, por tanto, varios procesos de servidor en segundo plano pueden estar ejecutándose de manera concurrente. Los procesos de servidor en segundo plano tienen acceso a los datos de la base de datos mediante el grupo de memorias intermedias de la memoria principal, que se ubica en la memoria compartida, por lo que todos los pro-

cesos tienen la misma vista de los datos. La memoria compartida también se utiliza para implementar otras formas de sincronización entre los procesos de servidor como, por ejemplo, el bloqueo de elementos de datos. El empleo de la memoria compartida como medio de comunicación exige que el servidor de PostgreSQL se ejecute en una sola máquina; los sitios con un solo servidor no pueden extenderse por varias máquinas.

Notas bibliográficas

Se puede encontrar una extensa documentación en línea de PostgreSQL en www.postgresql.org. Este sitio Web es la fuente autorizada de información sobre las nuevas versiones de PostgreSQL, que tienen lugar con frecuencia. Hasta PostgreSQL versión 8, la única manera de ejecutar PostgreSQL bajo Microsoft Windows era utilizar Cygwin. Cygwin es un entorno similar a Linux que permite la reconstrucción de aplicaciones Linux a partir de su código para ejecutarlas bajo Windows. Más detalles en www.cygwin.com.

Entre los libros sobre PostgreSQL están Douglas y Douglas [2003], y Stinson [2002]. Las reglas, tal y como se utilizan en PostgreSQL, se presentaron en Stonebraker et al. [1990]. La estructura GiST se describe en Hellerstein et al. [1995].

Las herramientas de administración de PostgreSQL, pgAccess y pgAdmin se describen en el sitio Web www.pgaccess.org y www.pgadmin.org, respectivamente.

Las herramientas de diseño de bases de datos de PostgreSQL, TORA y Data Architect, se describen en www.globecom.se/tora y www.thekompany.com/products/dataarchitect, respectivamente.

Las herramientas de generación de informes GNU Report Generator y GNU Enterprise se describen en www.gnu.org/software/grg y www.gnuenterprise.org, respectivamente.

El servidor OLAP de Mondrian se describe mondrian.sourceforge.net. Es de código abierto, como PostgreSQL.

Una alternativa de código abierto a PostgreSQL es MySQL, que está disponible para usos no comerciales bajo la Licencia Pública General GNU, pero que exige el pago para usos comerciales. Las comparaciones entre las versiones más recientes de los dos sistemas se pueden conseguir fácilmente en Web.

CAPÍTULO 27

Oracle

Hakan Jakobsson
Oracle Corporation

Cuando Oracle se fundó en 1977 como Software Development Laboratories por Larry Ellison, Bob Miner y Ed Oates no había productos de bases de datos relacionales comerciales. La compañía, cuyo nombre cambió posteriormente a Oracle, se estableció para construir un sistema de gestión de bases de datos como producto comercial y fue la primera en lanzarlo al mercado. Desde entonces Oracle ha mantenido una posición líder en el mercado de bases de datos relacionales, pero con el paso de los años su producto y servicios ofrecidos han crecido más allá del servicio de bases de datos relacionales. Además de las herramientas directamente relacionadas con el desarrollo y gestión de bases de datos Oracle ofrece herramientas de inteligencia de negocio, incluyendo herramientas de consulta y análisis, productos de minería de datos y un servidor de aplicaciones con una gran integración con el servidor de bases de datos.

Además de los servidores y herramientas relacionados con las bases de datos, la compañía ofrece software para la planificación empresarial de recursos y gestión de relaciones con el cliente, incluyendo áreas como finanzas, recursos humanos, manufactura, marketing, ventas y gestión de cadenas de proveedores. La unidad On Demand de Oracle ofrece servicios en estas áreas como un proveedor de servicios de aplicación.

Este capítulo cubre un subconjunto de características, opciones y funcionalidad de los productos Oracle. Continuamente se desarrollan nuevas versiones de los productos por lo que las descripciones de los productos están sujetas a cambios. Este conjunto de características descrito aquí está basado en la primera versión de Oracle10*g*.

27.1 Herramientas para el diseño de bases de datos y la consulta

Oracle proporciona una serie de herramientas para el diseño, consulta, generación de informes y análisis de datos para bases de datos, incluyendo OLAP.

27.1.1 Herramientas para el diseño de bases de datos

La mayor parte de las herramientas de diseño de Oracle están incluidas en Oracle Developer Suite. Es una familia de herramientas para los distintos aspectos de desarrollo de aplicaciones, incluyendo herramientas para el desarrollo de formularios, modelado de datos, informes y consultas. La familia de productos soporta el estándar UML (véase el Apartado 2.10) para el modelado en el desarrollo. Proporciona modelado de clases para generar código para componentes de negocio para un entorno Java así

como modelado de actividades para el modelado del flujo de control de propósito general. La familia también soporta XML para el intercambio de datos con otras herramientas UML.

La principal herramienta de diseño de bases de datos en la familia es Oracle Designer, que traduce la lógica de negocio y el flujo de datos en definiciones de esquemas y guiones procedimentales para la lógica de las aplicaciones. Soporta varias técnicas de modelado tales como diagramas E-R, ingeniería de información y análisis y diseño de objetos. Oracle Designer almacena el diseño en Oracle Repository, que sirve como un único punto de metadatos para la aplicación. Los metadatos se pueden entonces utilizar para generar formularios e informes. Oracle Repository permite la configuración de los objetos de bases de datos, formularios, clases Java, archivos XML y otros tipos de archivos.

La familia también contiene herramientas de desarrollo de aplicaciones para generar formularios, informes, y herramientas para distintos aspectos de desarrollo basado en Java y XML, incluyendo JDeveloper, que proporciona un entorno completo para el desarrollo de aplicaciones J2EE. El componente de inteligencia de negocio proporciona JavaBeans para funcionalidad analítica tal como visualización de datos, consultas y cálculos analíticos.

Oracle también tiene una herramienta de desarrollo de aplicaciones para el almacén de datos. Oracle Warehouse Builder. Warehouse Builder es una herramienta para el diseño e implantación de todos los aspectos de un almacén de datos, incluyendo el diseño del esquema, asignaciones de datos y transformaciones, procesamiento de carga de datos y gestión de metadatos. Oracle Warehouse Builder soporta los esquemas 3NF y en estrella y puede también importar diseños desde Oracle Designer. Esta herramienta, junto con características de la base de datos tales como las tablas externas y las funciones de tablas, elimina generalmente la necesidad de herramientas de terceros para la extracción y transformación.

27.1.2 Herramientas de consulta

Oracle proporciona herramientas de consulta, generación de informes y análisis de datos ad hoc, incluyendo OLAP.

Oracle Application Server Discoverer es una herramienta basada en Web para realizar consultas, informes, análisis y publicación Web ad hoc para usuarios finales y analistas de datos. Permite a los usuarios abstraer y concretar conjuntos de resultados, datos pivote y almacenar cálculos como informes que se pueden publicar en una serie de formatos tales como hojas de datos o HTML. Discoverer tiene asistentes que ayudan a los usuarios finales a visualizar los datos como gráficos. Oracle soporta un amplio conjunto de funciones analíticas tales como la agregación de clasificación y traslado en SQL. La interfaz de consulta de Discoverer puede generar SQL del que se puede aprovechar su funcionalidad y puede proporcionar a los usuarios finales grandes posibilidades de análisis. Puesto que el procesamiento tiene lugar en el sistema de gestión de la base de datos relacional, Discoverer no requiere generalmente un complejo motor de cálculo en el lado del cliente y existe una versión de Discoverer basada en el explorador.

27.2 Variaciones y extensiones de SQL

Oracle soporta parcial o totalmente todas las características principales de SQL:1999, con algunas pequeñas excepciones tales como distintos tipos de datos. Además, Oracle soporta un gran número de constructores del lenguaje, algunos de los cuales siguen la norma SQL:1999, mientras que otros son específicos de Oracle en sintaxis o funcionalidad. Por ejemplo Oracle soporta las operaciones OLAP descritas en el Apartado 18.2, incluyendo clasificación, agregación de traslado, cubos y abstracción.

Algunos ejemplos de las extensiones SQL de Oracle son:

- **connect by**, que es una forma de recorrido de árboles que permite cálculos al estilo del cierre transitivo en una única instrucción SQL. Es una sintaxis para una característica que Oracle tenía desde los años 80.
- **upsert** e **inserciones en varias tablas**. La operación upsert combina una actualización y una inserción y es útil para combinar datos nuevos con antiguos en aplicaciones de almacén de datos. Si una nueva fila tiene el mismo valor de clave que una fila antigua se actualiza la fila antigua (por ejemplo agregando los valores desde la nueva fila), en otro caso se inserta la nueva fila en la

tabla. Las inserciones en varias tablas permiten actualizar varias tablas basándose en una única exploración de los nuevos datos.

- cláusula **with**, que se describe en el Apartado 4.8.2.
- cláusula **model**, que permite cálculos algebráicos sobre arrays de datos relacionales. Para algunas aplicaciones, esta cláusula puede ser una alternativa al uso de hojas de cálculo.

27.2.1 Características relacionales orientadas a objetos

Oracle tiene soporte extensivo para constructores relacionales orientados a objetos, incluyendo:

- **Tipos de objetos**. Se soporta un único modelo de herencia para las jerarquías de tipos.
- **Tipos de colecciones**. Oracle soporta **varrays**, que son arrays de longitud variable, y tablas anidadas.
- **Tablas de objetos**. Se utilizan para almacenar objetos mientras se proporciona una vista relacional de los atributos de los objetos.
- **Funciones de tablas**. Son funciones que producen conjuntos de filas como salida y se pueden utilizar en la cláusula **from** de una consulta. Las funciones de tablas se pueden anidar en Oracle. Si una función de tablas se utiliza para expresar algún formulario de transformación de datos, el anidamiento de varias funciones permite que se expresen varias transformaciones en una única instrucción.
- **Vistas de objetos**. Proporcionan una vista de tablas de objetos virtuales de datos almacenados en una tabla relacional normal. Permite acceder o ver los datos en un estilo orientado a objetos incluso si los datos están realmente almacenados en un formato relacional tradicional.
- **Métodos**. Se pueden escribir en PL/SQL, Java o C.
- **Funciones de agregación definidas por el usuario**. Se pueden utilizar en instrucciones SQL de la misma forma que las funciones incorporadas tales como **sum** y **count**.
- **XML como tipo de datos nativo**. Se puede utilizar para almacenar e indexar documentos XML. Oracle también puede convertir automáticamente el resultado de cualquier consulta SQL en XML.

Oracle tiene dos lenguajes procedimentales principales, PL/SQL y Java. PL/SQL fue el lenguaje original de Oracle para los procedimientos almacenados y tiene una sintaxis similar al utilizado en el lenguaje Ada. Java se soporta mediante una máquina virtual Java dentro del motor de la base de datos. Oracle proporciona un paquete para encapsular procedimientos, funciones y variables relacionadas en unidades únicas. Oracle soporta SQLJ (SQL incorporado en Java) y JDBC y proporciona una herramienta para generar las definiciones de clases Java correspondientes a tipos de la base de datos definidos por el usuario.

27.2.2 OLAP

Anteriormente el producto OLAP de Oracle era un servidor de bases de datos multidimensionales separado. Actualmente el procesamiento OLAP se realiza dentro de la propia base de datos relacional. Existen muchas razones para no incluir un motor de almacenamiento multidimensional separado:

- Un motor relacional puede dimensionarse a conjuntos de datos mucho mayores.
- Se puede utilizar un modelo de seguridad común para las aplicaciones analíticas y el almacén de datos.
- El modelado multidimensional se puede integrar con el modelado del almacén de datos.
- El sistema de gestión de bases de datos relacional tiene un conjunto mayor de características y funcionalidades en muchas áreas tales como la alta disponibilidad, las copias de seguridad, la recuperación y el soporte de herramientas de terceros.

- No hay necesidad de formar administradores de bases de datos para dos motores de bases de datos.

El principal reto al evitar un motor de bases de datos multidimensional separado es proporcional el mismo rendimiento. Un sistema de gestión de bases de datos mutidimensional que materializa todo o grandes partes de un cubo de datos puede ofrecer tiempos de respuesta muy cortos para muchos cálculos. Oracle ha enfocado este problema desde varias perspectivas.

- Oracle ha agregado soporte SQL para un amplio rango de funciones analíticas, incluyendo cubos, abstracciones, conjuntos de agrupación, clasificaciones (*ranks*), agregación móvil, funciones *led* y *lag*, cajones de histograma, regresión lineal y desviación estándar, junto con la capacidad de optimizar la ejecución de dichas funciones en el motor de la base de datos.
- Oracle ha extendido las vistas materializadas para permitir funciones analíticas, en particular los conjuntos de agrupación. La capacidad de materializar partes o todo el cubo es primordial para el rendimiento de un sistema de gestión de bases de datos multidimensionales, y las vistas materializadas proporcionan al sistema de gestión de bases de datos relacionales la capacidad de realizar lo mismo.
- Oracle ha introducido los *espacios de trabajo analíticos*, que almacenan los datos en formatos multidimensionales dentro de una tabla relacional y tienen métodos asociados para las operaciones OLAP como el modelado, asignación, agregación, predicciones y análisis de escenarios. Los espacios de trabajo analíticos se pueden acceder con funciones de tablas en SQL.

27.2.3 Disparadores

Oracle proporciona varios tipos de disparadores y varias opciones para el momento y forma en que se invocan (véase el Apartado 8.6 para una introducción a los disparadores en SQL). Los disparadores se pueden escribir en PL/SQL o Java o como llamadas a C. Para los disparadores que se ejecutan sobre instrucciones LMD tales como insert, update y delete, Oracle soporta disparadores de filas (**row**) y disparadores de instrucciones (**statement**). Los disparadores de filas se pueden ejecutar una vez por cada fila que se vea afectada (actualización o borrado, por ejemplo) por la operación LMD. Un disparador de instrucciones se ejecuta solamente una vez por instrucción. En cada caso, el disparador se puede definir tanto como un disparador *before* o *after* dependiendo de si se va a invocar antes o después de que se lleva a cabo la operación LMD.

Oracle permite la creación de disparadores **instead of** para las vistas que no pueden estar sujetas a operaciones LMD. Dependiendo de la definición de la vista puede no ser posible para Oracle traducir una instrucción LMD en una vista a modificaciones de las tablas base subyacentes sin ambigüedad. Por ello las operaciones LMD sobre vistas están sujetas a numerosas restricciones. Se puede crear un disparador **instead of** sobre una vista para especificar manualmente las operaciones sobre las tablas base que van a ocurrir en respuesta a la operación LMD sobre la vista. Oracle ejecuta el disparador en lugar de la operación LMD y por consiguiente proporciona un mecanismo de rodeo de las restricciones sobre las operaciones LMD sobre las vistas.

Oracle también tiene disparadores que ejecutan otros eventos, tales como el inicio o finalización de la base de datos, mensajes de error del servidor, inicio o finalización de sesión de un usuario e instrucciones LDD tales como las instrucciones **create**, **alter** o **drop**.

27.3 Almacenamiento e índices

En la jerga de Oracle, una *base de datos* consiste en información almacenada en archivos y se accede a través de un *ejemplar*, que es un área de memoria compartida y un conjunto de procesos que interactúa con los datos en los archivos.

27.3.1 Espacios de tablas

Una base de datos consiste en una o más unidades de almacenamiento lógicas denominadas **espacios de tablas**. Cada espacio de tablas, a su vez, consiste en una o más estructuras físicas denominadas **archivos de datos**. Éstos pueden ser archivos gestionados por el sistema operativo o dispositivos en bruto.

Normalmente una base de datos Oracle tendrá los siguientes espacios de tablas:

- El espacio de tablas del **sistema**, que siempre se crea. Contiene las tablas diccionario de datos y almacenamiento para los disparadores y los procedimientos almacenados.
- Espacios de tablas creados para almacenar los datos de usuario. Aunque los datos de usuario se pueden almacenar en el espacio de tablas del **sistema** es frecuentemente deseable separar los datos de usuario de los datos del sistema. Normalmente la decisión sobre los otros espacios de tablas que se deben crear está basada en el rendimiento, disponibilidad, capacidad de mantenimiento y facilidad de administración. Por ejemplo, puede ser útil tener varios espacios de tablas para las operaciones de copia de seguridad parcial y recuperación.
- Los espacios de tablas temporales. Muchas operaciones de base de datos requieren la ordenación de los datos y la rutina de ordenación puede tener que almacenar los datos temporalmente en el disco si la ordenación no se puede realizar en memoria. Se asignan espacios de tablas temporales a la ordenación y asociación para realizar las operaciones de gestión de espacio involucradas en un volcado más eficiente a disco.

Los espacios de tablas también se pueden utilizar como un medio para trasladar datos entre las bases de datos. Por ejemplo, es común trasladar los datos desde un sistema transaccional a un almacén de datos a intervalos regulares. Oracle permite trasladar todos los datos en un espacio de tablas de un sistema a otro sencillamente copiando los archivos y exportando e importando una pequeña cantidad de metadatos del diccionario de datos. Estas operaciones pueden ser mucho más rápidas que descargar los datos de una base de datos y después usar un descargador para insertarlos en la otra.

27.3.2 Segmentos

El espacio en un espacio de tablas se divide en unidades, denominadas **segmentos**, cada una de las cuales contiene datos para una estructura de datos específica. Existen cuatro tipos de segmentos

- **Segmentos de datos**. Cada tabla en un espacio de tablas tiene su propio segmento de datos donde se almacenan los datos de la tabla a menos que la tabla esté dividida; si ocurre esto, hay un segmento de datos por división (la división en Oracle se describe en el Apartado 27.3.10).
- **Segmentos de índices**. Cada índice en un espacio de tablas tiene su propio segmento de índices, excepto los índices divididos, los cuales tienen un segmento de índices por división.
- **Segmentos temporales**. Son segmentos utilizados cuando una operación de ordenación necesita escribir datos al disco o cuando los datos se insertan en una tabla temporal.
- **Segmentos de retroceso**. Estos segmentos contienen información para deshacer de forma que se puede deshacer una copia no terminada. También juegan un papel importante en el modelo de control de concurrencia en Oracle y para la recuperación de la base de datos, descrito en los Apartados 27.5.1 y 27.5.2.

Debajo del nivel de segmentos se asigna espacio a un nivel de granularidad, denominado *extensión*. Cada extensión consiste en un conjunto de *bloques* contiguos de la base de datos. Un bloque de la base de datos es el nivel más bajo de granularidad en el cual Oracle ejecuta E/S a disco. Un bloque de base de la base de datos no tiene que tener el mismo tamaño que un bloque de un sistema operativo, pero debería ser un múltiplo.

Oracle proporciona parámetros de almacenamiento que permiten un control detallado de cómo se asigna y gestiona el espacio, tales como:

- El tamaño de una extensión nueva que se va a asignar para proporcionar espacio a las filas que se insertan en una tabla.
- El porcentaje de utilización de espacio con el cual un bloque de la base de datos se considera lleno y con el cual no se introducirán más filas en ese bloque (dejando algo de espacio libre en un bloque se puede conseguir que las filas existentes aumenten su tamaño cuando se realizan actualizaciones, sin agotar el espacio del bloque).

27.3.3 Tablas

Una tabla estándar en Oracle está organizada en montículo; esto es, la ubicación de almacenamiento de una fila en una tabla no está basada en los valores contenidos en la fila y se fija cuando la fila se inserta. Sin embargo, si la tabla se divide, el contexto de la fila afecta a la partición en la cual está almacenada. Hay varias características y variaciones.

Las tablas de montículos se pueden comprimir opcionalmente. Oracle usa un algoritmo de compresión sin pérdidas y basado en diccionario que se aplica individualmente a cada bloque de datos. Para las tablas que contienen grandes cantidades de valores repetidos el ahorro en espacio de disco y, por tanto, de operaciones E/S, puede ser muy grande, pero la compresión y descompresión de datos implica una ligera sobrecarga para la CPU.

Oracle soporta las tablas anidadas; esto es, una tabla puede tener una columna cuyo tipo de datos sea otra tabla. La tabla anidada no se almacena en línea en la tabla padre sino que se almacena en una tabla separada.

Oracle soporta tablas temporales donde la duración de los datos es la de la transacción en la cual se insertan los datos o la sesión de usuario. Los datos son privados a la sesión y se eliminan automáticamente al final de su duración.

Una *agrupación* es otra forma de organización de los datos de la tabla (véase el Apartado 11.7). El concepto, en este contexto, no se debería confundir con otros significados de la palabra *agrupación*, tales como los relacionados con la arquitectura de la computadora. En una agrupación las filas de tablas diferentes se almacenan juntas en el mismo bloque según algunas columnas comunes. Por ejemplo, una tabla de departamento y una tabla de empleados se podrían agrupar de forma que cada fila en la tabla departamento se almacene junto con todas las filas de los empleados que trabajan en ese departamento. Los valores de la clave principal o clave externa se utilizan para determinar la ubicación de almacenamiento. Esta organización mejora el rendimiento cuando las dos tablas están combinadas pero sin un aumento de espacio de un esquema desnormalizado puesto que los valores en la tabla de departamento no están repetidos para cada empleado. Como compromiso, una consulta que involucra solamente la tabla departamento puede tener que involucrar un número sustancialmente más grande de bloques que si la tabla se almacenara sola.

La organización en agrupación implica que una fila pertenece a un lugar específico; por ejemplo, una nueva fila de empleado se debe insertar con las otras filas para el mismo departamento. Por consiguiente, es obligatorio un índice en la columna de agrupación. Una organización alternativa es una agrupación asociativo. Aquí, Oracle calcula la localización de una fila aplicando una función asociativa al valor para la columna de agrupación. La función asociativa asigna la fila a un bloque específico en la agrupación asociativa. Puesto que no es necesario el recorrido del índice para acceder a una fila según su valor de columna de agrupación, esta organización puede ahorrar cantidades significativas de E/S a disco. Sin embargo, el número de cajones asociativos y otros parámetros de almacenamiento se deben establecer cuidadosamente para evitar problemas de rendimiento debido a demasiadas colisiones o malgasto de espacio debido a cajones asociativos vacíos.

La organización según agrupación asociativa y según agrupación normal se puede aplicar a una única tabla El almacenamiento de una tabla como una agrupación asociativa con la columna de la clave principal como la clave de la agrupación puede permitir un acceso basado en un valor de clave principal con un una única E/S a disco, siempre que no haya desbordamiento para ese bloque de datos.

27.3.4 Tablas organizadas con índices

En una tabla *organizada con índices* los registros se almacenan en un índice de árbol B Oracle en lugar de en un montículo. Una tabla organizada con índices requiere que se identifique una clave única para su uso como la clave del índice. Aunque una entrada en un índice normal contiene el valor de la clave y el identificador de fila de la fila indexada, una tabla organizada con índices reemplaza el identificador de fila con los valores de la columna para el resto de columnas en la tabla. Comparado con el almacenamiento de los datos en una tabla en montículo normal y la creación de un índice según las columnas clave, una tabla organizada con índices puede mejorar el rendimiento y el espacio. Considérese la lectura de todos los valores de columna de una fila, dado su valor de clave principal. Para una tabla en montículo se requeriría un examen del índice seguido por un acceso a tabla mediante identificador de fila. Para una tabla organizada con índices solamente es necesario el examen del índice.

Los índices secundarios sobre columnas que no sean clave de una tabla organizada con índices son distintos de los índices en una tabla en montículo normal. En una tabla en montículo cada fila tiene una identificador de fila fija que no cambia. Sin embargo, un árbol B se reorganiza al crecer o disminuir cuando se insertan o borran las entradas, y no hay garantía de que una fila permanezca en una ubicación dentro de una tabla organizada con índices. Por ello, un índice secundario en una tabla organizada con índices no contiene identificadores de fila normales, sino **identificadores lógicos de fila**. Un identificador lógico de fila consiste en: un identificador de fila física correspondiente a donde la fila estaba cuando se creó el índice o la última reconstrucción, y un valor para la clave única. El identificador de fila física se conoce como una "suposición" puesto que sería incorrecto si la fila se ha trasladado. En este caso la otra parte del identificador lógico de fila, el valor de la clave para la fila se utiliza para acceder a la fila; sin embargo, este acceso es más lento que si la suposición hubiera sido correcta puesto que involucra un recorrido del árbol B para la tabla organizada con índices desde la raíz hasta los nodos hoja, incurriendo potencialmente en varias operaciones E/S de disco. Sin embargo, si una tabla es altamente volátil y es probable que un buen porcentaje de suposiciones sean incorrectas, puede ser mejor crear un índice secundario con solamente valores clave, puesto que el uso de una suposición incorrecta puede producir una E/S a disco malgastada.

27.3.5 Índices

Oracle soporta varios tipos distintos de índices. El tipo más comúnmente utilizado es lo que Oracle (y otros fabricantes) denominan un índice de árbol B (aunque realmente es lo que se denomina en el Capítulo 12 como índice de árbol B^+) creado en una o varias columnas. Las entradas de los índices tienen el siguiente formato: para un índice en las columnas col_1, col_2 y col_3, cada fila en la tabla donde al menos una columna tenga un valor no nulo resultaría en la entrada de índice

$$<col_1><col_2><col_3><id\text{-}fila>$$

donde $<col_i>$ denota el valor para la columna i e $< id\text{-}fila >$ es el identificador de fila para la fila. Oracle puede opcionalmente comprimir el prefijo de la entrada para ahorrar espacio. Por ejemplo, si hay muchas combinaciones repetidas de valores $<col_1><col_2>$, la representación de cada prefijo $<col_1><col_2>$ distinto se puede compartir entre las entradas que tienen esa combinación de valores, en lugar de almacenarlo explícitamente para cada entrada. La compresión de prefijos puede llevar a ahorros de espacio sustanciales.

27.3.6 Índices de mapas de bits

Los índices de mapas de bits (descritos en el Apartado 12.9) utilizan una representación de mapa de bits para entradas de índice que pueden llevar a un ahorro sustancial de espacio (y por consiguiente ahorro de E/S a disco), cuando la columna indexada tiene un número moderado de valores distintos. Los índices de mapas de bits en Oracle utilizan la misma clase de estructura de árbol B para almacenar las entradas como un índice normal. Sin embargo, donde un índice normal en una columna tendría entradas de la forma $<col_1><id\text{-}fila>$, una entrada de índice de mapa de bits tiene la forma

$$<col_1><id\text{-}filainicial><id\text{-}filafinal><mapabitscomprimido>$$

El mapa de bits conceptualmente representa el espacio de todas las filas posibles en la tabla entre los identificadores de la fila inicial y final. El número de tales filas posibles en un bloque depende de cuántas filas se pueden alojar en un bloque, que es una función del número de columnas en la tabla y sus tipos de datos. Cada bit en el mapa de bits representa una fila posible en un bloque. Si el valor de la columna de esa fila es el de la entrada de índice, el bit se establece a 1. Si la fila tiene algún otro valor o la fila no existe realmente en la tabla, el bit se establece a 0 (es posible que la fila no exista realmente porque un bloque de la tabla puede tener un número más pequeño de filas que el número que se calculó como el máximo posible). Si la diferencia es grande, el resultado pueden ser grandes cadenas de ceros consecutivos en el mapa de bits, pero el algoritmo de compresión trata dichas cadenas de ceros por lo que el efecto negativo se limita.

El algoritmo de compresión es una variación de una técnica de compresión denominada compresión de mapas de bits alineados (Byte-Aligned Bitmap Compression, BBC). Esencialmente, una sección del mapa de bits donde la distancia entre dos unos consecutivos es suficientemente pequeña se almacena como mapas de bits como tales. Si la distancia entre dos unos es suficientemente grande—esto es, hay un número suficiente de ceros entre ellos—se almacena el número de ceros.

Los índices de mapas de bits permiten varios índices en la misma tabla para combinarse en la misma ruta de acceso si hay varias condiciones sobre las columnas indexadas en la cláusula **where** de una consulta. Por ejemplo, para la condición

$$(col_1 = 1 \textbf{ or } col_1 = 2) \textbf{ and } col_2 > 5 \textbf{ and } col_3 <> 10$$

Oracle podría calcular las filas que coinciden con la condición ejecutando operaciones booleanas sobre los mapas de bits a partir los mapas de bits de índices sobre las tres columnas. En este caso, estas operaciones se realizarían para cada índice:

- Para el índice en col_1, se realizaría la disyunción (**or**) de los valores de clave 1 y 2.
- Para el índice en col_2, todos los mapas de bits para los valores de la clave mayores que 5 se mezclarían en una operación que corresponde a una disyunción.
- Para el índice en col_3, se obtendrían los mapas de bits para los valores 10 y **null**. Entonces, se aplicaría una conjunción sobre los resultados de los dos primeros índices, seguido por dos operaciones menos booleanas de los mapas de bits para los valores 10 y **null** para col_3.

Todas las operaciones se realizan directamente sobre la representación comprimida de los mapas de bits—no es necesaria la descompresión—y el mapa de bits resultante (comprimido) representa las filas que cumplen todas las condiciones lógicas.

La capacidad de utilizar las operaciones booleanas para combinar varios índices no está limitada a los índices de mapas de bits. Oracle puede convertir identificadores de filas a la representación de mapa de bits comprimidos, por lo que se puede utilizar un índice de árbol B normal en cualquier lugar de un árbol binario u operación de mapa de bits simplemente poniendo un operador id−fila−a−mapa−de−bits en la parte superior del acceso a índices del plan de ejecución.

Como regla nemotécnica, los índices de mapas de bits tienden a ser más eficientes en el espacio que los índices de árbol B si el número de valores distintos de la clave es menor que la mitad del número de filas en una tabla. Por ejemplo, en una tabla con un millón de filas, un índice en una columna con menos de 500.000 valores distintos probablemente sería menor si se creara como un índice de mapa de bits. Para las columnas con un número muy pequeño de valores distintos—por ejemplo, las columnas que se refieren a propiedades tales como país, estado, género, estado marital y varios estados indicadores—un índice mapa de bits podría requerir solamente una pequeña fracción del espacio normal de un índice de árbol B normal. Cualquier ventaja en el espacio también puede dar lugar a mejoras en el rendimiento en la forma de menos operaciones E/S a disco cuando se explora el índice.

27.3.7 Índices basados en funciones

Además de crear índices sobre una o varias columnas de una tabla, Oracle permite crear índices sobre expresiones que involucran a una o más columnas, tales como $col_1 + col_2 * 5$. Por ejemplo, la creación

de un índice sobre la expresión *upper* (*nombre*), donde *upper* es una función que devuelve la versión en mayúsculas de una cadena y *nombre* es una columna, es posible realizar búsquedas independientes de la caja (mayúsculas o minúsculas) sobre la columna *nombre*. Con el fin de buscar todas las filas con el nombre "van Gogh" de una forma eficiente se puede utilizar la condición

$$upper(nombre) = \text{'VAN GOGH'}$$

en la cláusula **where** de la consulta. Oracle entonces casa la condición con la definición de índice y concluye que se puede utilizar el índice para recuperar todas las filas que coincidan con "van Gogh" sin considerar las mayúsculas y minúsculas del nombre cuando se almacenó en la base de datos. Se puede crear un índice basado en función como un mapa de bits o como un índice de árbol B.

27.3.8 Índices de reunión

Un índice de reunión es un índice donde las columnas clave no están en la tabla que se referencia mediante los identificadores de filas en el índice. Oracle soporta los índices de reunión mapa de bits principalmente para su uso con esquemas en estrella (véase el Apartado 18.3.2). Por ejemplo, si hay una columna para los nombres de los productos en una tabla de la dimensión productos, se podría utilizar un índice de reunión de mapas de bits sobre la tabla de hechos con esta columna clave para recuperar las filas de la tabla de hechos que corresponden a un producto con un nombre específico, aunque el nombre no esté almacenado en la tabla de hechos. La forma en la que las filas en las tablas de hechos y de la dimensión correspondientes está basada en una condición de reunión que se especifica cuando se crea el índice y se convierte en parte de los los metadatos de índices. Cuando se procesa una consulta, el optimizador buscará la misma condición de reunión en la cláusula **where** de la consulta con el fin de determinar si es aplicable el índice de reunión.

Oracle permite índices de reunión de mapa de bits para tener más de una columna clave y estas columnas pueden estar en tablas diferentes. En todos los casos las condiciones de reunión entre la tabla de hechos donde se construye el índice y las tablas dimensionales se deben referir a claves únicas en las tablas dimensionales; esto es, una fila indexada en la tabla de hechos debe corresponder a una única fila en cada una de las tablas de dimensión.

Oracle puede combinar un índice de reunión de mapa de bits en una tabla de hechos con otros índices en la misma tabla—tanto si hay índices de reunión o no—mediante el uso de operadores para las operaciones booleanas del mapa de bits. Por ejemplo, consideremos un esquema con una tabla de hechos para las ventas y tablas dimensionales para los clientes, productos y fechas. Supóngase que una consulta solicita información sobre las ventas a los clientes en un cierto código postal que compraron productos de una cierta categoría de producto durante un cierto periodo de tiempo. Si existe un índice de reunión de mapa de bits sobre varias columnas donde las columnas clave son las columnas de la tabla de dimensión restringidas (código postal, categoría de producto y fecha), Oracle puede utilizar el índice de reunión para buscar las filas en la tabla de hechos que coinciden con las condiciones de restricción. Sin embargo, si existen índices individuales sobre una única columna para las columnas clave (o un subconjunto de ellas), Oracle puede recuperar los mapas de bits de las filas de la tabla de hechos que coinciden con cada condición individual y utilizar la operación **and** booleana para generar un mapa de bits de la tabla de hechos para aquellas filas que satisfacen todas las condiciones. Si la consulta contiene condiciones sobre algunas columnas de la tabla de hechos, los índices de aquellas columnas se podrían incluir en la misma ruta de acceso, incluso si fueran índices normales de árbol B o índices de dominio (los índices de dominio se describen posteriormente en el Apartado 27.3.9).

27.3.9 Índices de dominio

Oracle permite que las tablas sean indexadas por estructuras de índices que no sean propias de Oracle. Esta característica de extensibilidad del servidor Oracle permite a los fabricantes de software desarrollar los llamados cartuchos con funcionalidad para dominios de aplicación específicos, tales como texto, datos espaciales e imágenes, con la funcionalidad de indexado más allá de la proporcionada por los tipos de índice Oracle estándar. Para implementar la lógica para crear, mantener y buscar en el índice,

el diseñador de índices debe asegurar que se adhiere a un protocolo específico en su interacción con el servidor Oracle.

Un índice de dominio se debe registrar en el diccionario de datos junto con los operadores que soporta. El optimizador de Oracle considera los índices de dominio como una de las posibles rutas de acceso para una tabla. Oracle permite a las funciones de coste registrarse con los operadores de forma que el optimizador pueda comparar el coste del uso del índice de dominio con los de otras rutas de acceso.

Por ejemplo, un índice de dominio para búsquedas de texto avanzadas puede soportar un operador *contains* (contiene). Una vez que se ha registrado este operador, el índice de dominio se considerará como una ruta de acceso para una consulta como

```
select *
from empleado
where contains(resumen, 'LINUX')
```

donde *resumen* es una columna de texto en la tabla *empleados*. El índice de dominio se puede almacenar en un archivo de datos externo o dentro de una tabla Oracle organizada con índices.

Los índices de dominio se pueden combinar con otros índices (mapa de bits o de árbol B) en la misma ruta de acceso con la conversión entre la representación de mapa de bits y el identificador de fila y usando operaciones booleanas del mapa de bits.

27.3.10 División en particiones

Oracle soporta varias clases de división horizontal de tablas e índices. Su función principal es dar soporte a bases de datos muy grandes. Esta capacidad de dividir una tabla o índice resulta de interés en muchas aplicaciones.

- La copia de seguridad y recuperación es más sencilla y rápida puesto que se puede realizar sobre particiones individuales en lugar de sobre toda la tabla.
- Las operaciones de carga en un entorno de almacén de datos son menos intrusivas: se pueden agregar datos a una partición y después agregar la partición a una tabla, lo que es una operación instantánea. De igual forma, eliminar una partición con datos obsoletos desde una tabla es muy sencillo en un almacén de datos que mantenga una ventana de datos históricos.
- El rendimiento de la consulta se mejora sustancialmente puesto que el optimizador puede reconocer que solamente se tiene que acceder a un subconjunto de las particiones de una tabla con el fin de resolver la consulta (poda de particiones). También el optimizador puede reconocer que en una reunión no es necesario intentar hacer corresponder todas las filas en una tabla con todas las filas en la otra, pero que las reuniones se necesitan realizar solamente entre pares coincidentes de divisiones (reunión por particiones).

Cada fila en una tabla dividida está asociada con una partición específica. Esta asociación está basada en la columna o columnas de la división que son parte de la definición de una tabla dividida. Hay varias formas para hacer corresponder valores de columna a divisiones, dando lugar a varios tipos de divisiones, cada una con distintas características: divisiones por rangos, asociativas, por listas y compuestas.

27.3.10.1 División por rangos

En la división por rangos los criterios de división son rangos de valores. Este tipo de división está especialmente indicado para columnas de fechas, en cuyo caso todas las filas en el mismo rango de fechas, digamos un día o un mes, pertenecen a la misma partición. En un almacén de datos donde los datos se cargan desde sistemas transaccionales a intervalos regulares, la división por rangos se puede utilizar para implementar eficientemente una ventana de datos históricos. Cada carga de datos obtiene su nueva partición propia, haciendo que el proceso de carga sea más rápido y eficiente. El sistema realmente carga los datos en una tabla separada con la misma definición de columna que en una tabla dividida. Se puede entonces verificar la consistencia de los datos, arreglarlos e indexarlos. Después de eso el sistema

puede hacer de la tabla separada una nueva partición mediante un sencillo cambio de los metadatos en el diccionario de datos—una operación casi instantánea.

Mientras no cambien los metadatos, el proceso de carga no afecta a los datos existentes en la tabla dividida en ningún caso. No hay necesidad de realizar ningún mantenimiento de los índices existentes como parte de la carga. Los datos antiguos se pueden eliminar de un tabla sencillamente eliminando su partición; esta operación no afecta al resto de particiones.

Además, las consultas en un entorno de almacén de datos frecuentemente contienen condiciones que los restringen a un cierto periodo de tiempo, tal como una quincena o mes. Si se utiliza la división de datos por rangos el optimizador de consulta puede restringir el acceso a los datos de aquellas particiones que son relevantes a la consulta y evitar una exploración de toda la tabla.

27.3.10.2 División asociativa

En la división asociativa, una función asociativa hace corresponder filas con divisiones según los valores en las columnas de la división. Este tipo de división es útil principalmente cuando es importante distribuir las filas equitativamente entre las particiones o cuando las reuniones por particiones son importantes para el rendimiento de la consulta.

27.3.10.3 División compuesta

En la división compuesta la tabla se divide por rangos, pero cada partición tiene subparticiones mediante el uso de la división asociativa o por listas. Este tipo de división combina las ventajas de la división por rangos y la división asociativa o por listas.

27.3.10.4 División por listas

En la división por listas los valores asociados con una partición particular están en una lista. Este tipo de división es útil si los datos en la columna de división tienen un conjunto relativamente pequeño de valores discretos. Por ejemplo, una tabla con una columna para la provincia se puede partir implícitamente por región geográfica si cada lista de particiones tiene las provincias que pertenecen a la misma región.

27.3.11 Vistas materializadas

La característica de la vista materializada (véase el Apartado 3.9.1) permite almacenar el resultado de una consulta SQL y utilizarlo en un procesamiento posterior. Además, Oracle mantiene el resultado materializado, actualizándolo cuando se actualizan las tablas a las que se hicieron referencia en la consulta. Las vistas materializadas se utilizan en el almacén de datos para acelerar el procesamiento de la consulta, pero esta tecnología también se utiliza para la réplica en entornos distribuidos y móviles.

En el almacén de datos, un uso común de vistas materializadas es resumir los datos. Por ejemplo, un tipo común de consulta solicita "la suma de las ventas de cada cuatrimestre durante los últimos dos años". El precálculo de los resultados, o algún resultado parcial, de dicha consulta puede acelerar drásticamente el procesamiento de la consulta comparado con calcularlo desde cero con la agregación de todos los registros de ventas por detalle.

Oracle soporta reescrituras automáticas de las consultas que aprovechan cualquier vista materializada útil cuando se resuelve una consulta. La reescritura consiste en cambiar la consulta para utilizar la vista materializada en lugar de las tablas originales en la consulta. Además, la reescritura puede agregar reuniones adicionales o procesamiento de agregación si son necesarias para obtener el resultado correcto. Por ejemplo, si una consulta necesita las ventas por cuatrimestre, la reescritura puede aprovechar una vista que materializa las ventas por mes, añadiendo agregación adicional para agrupar los meses en cuatrimestres. Oracle tiene un tipo de objeto de metadatos denominado **dimension** que permite las relaciones jerárquicas en las tablas a definir. Por ejemplo, una tabla de la dimensión temporal en un esquema en estrella Oracle puede definir un objeto de metadatos *dimension* para especificar cómo se

agrupan los días en meses, los meses en cuatrimestres, los cuatrimestres en años, y así sucesivamente. De igual forma se pueden especificar las propiedades jerárquicas relacionadas con la geografía—por ejemplo, cómo los distritos de ventas se agrupan en regiones. La lógica de la reescritura de la consulta examina estas relaciones puesto que permite utilizar una vista materializada para clases más amplias de consultas.

El objeto contenedor para una vista materializada es una tabla, lo que significa que una vista materializada se puede indexar, dividir o estar sujeta a otros controles para mejorar el rendimiento de la consulta.

Cuando hay cambios en los datos de las tablas referenciadas en la consulta que define una vista materializada se debe actualizar la vista materializada para reflejar dichos cambios. Oracle soporta tanto la actualización completa de una vista materializada como una actualización rápida incremental. En una actualización completa Oracle vuelve a calcular la vista materializada desde cero, lo cual puede ser la mejor opción si las tablas subyacentes han tenido cambios significativos, por ejemplo, debidos a una carga masiva. En una actualización incremental Oracle actualiza la vista utilizando registros que fueron cambiados en las tablas subyacentes; la actualización de la vista es inmediata—esto es, se ejecuta como parte de la transacción que cambió las tablas subyacentes. La actualización incremental puede ser mejor si el número de filas que se han cambiado es pequeño. Hay algunas restricciones sobre las clases de consultas según las que una vista materializada se puede actualizar de forma incremental (y otras que indican si una vista materializada siquiera se puede crear).

Una vista materializada es similar a un índice en el sentido que, aunque puede mejorar el rendimiento de la consulta, usa espacio, y su creación y mantenimiento consume recursos. Para ayudar a resolver este compromiso Oracle proporciona un asesor que puede ayudar al usuario a crear vistas materializadas menos costosas, dada una carga de trabajo particular como entrada.

27.4 Procesamiento y optimización de consultas

Oracle soporta una gran variedad de técnicas de procesamiento en su motor de procesamiento de consultas. Algunas de las más importantes se describen aquí brevemente.

27.4.1 Métodos de ejecución

Se puede acceder a los datos mediante una serie de métodos de acceso:

- **Exploración de tabla completa**. El procesador de consultas explora toda la tabla, obtiene información sobre los bloques que forman la tabla del mapa de extensión y explora estos bloques.
- **Exploración de índices**. El procesador crea una clave de comienzo y/o finalización a partir de las condiciones en la consulta y la utiliza para explorar una parte relevante del índice. Si hay columnas que se tienen que recuperar, y que no son parte del índice, la exploración del índice irá seguida de un acceso a la tabla mediante el índice del identificador de fila. Si no hay disponible ninguna clave de inicio o parada la exploración será una exploración de índice completa.
- **Exploración rápida completa de índices**. El procesador explora las extensiones de la misma forma que la extensión de tabla en una exploración de tabla completa. Si el índice contiene todas las columnas de la tabla que se necesitan en ella y no hay buenas claves de inicio y parada que puedan reducir significativamente esa porción del índice que se exploraría en una exploración de índices normal, este método puede ser la forma más rápida de acceder a los datos. Esto es porque la exploración rápida completa aprovecha de forma completa la E/S de disco de varios bloques. Sin embargo, a diferencia de una exploración completa normal, que recorre los bloques hoja del índice en orden, una exploración rápida completa no garantiza que la salida preserve el orden del índice.
- **Reunión de índices**. Si una consulta necesita solamente un pequeño subconjunto de columnas de una tabla ancha, pero ningún índice contiene todas estas columnas, el procesador puede utilizar una reunión de índices para generar la información relevante sin acceder a la tabla, reuniendo

varios índices que contienen en conjunto las columnas necesarias. Ejecuta las reuniones como reunión por asociación sobre los identificadores de filas desde los distintos índices.

- Acceso a agrupaciones y agrupaciones asociadas. El procesador accede a los datos utilizando la clave de agrupación.

Oracle tiene diversas formas de combinar información desde varios índices en una única ruta de acceso. Esta posibilidad permite varias condiciones en la cláusula **where** que se pueden utilizar conjuntamente para calcular el conjunto de resultados de la forma más eficientemente posible. La funcionalidad incluye la capacidad de ejecutar las operaciones booleanas conjunción, disyunción y diferencia sobre mapas de bits que representan los identificadores de filas. Hay también operadores que hacen corresponder una lista de identificadores de filas con mapas de bits y viceversa, lo que permite que los índices de árbol B normales y los índices de mapas de bits utilicen la misma ruta de acceso. Además, para muchas consultas que involucran **count**(*) en selecciones sobre una tabla el resultado se puede calcular simplemente contando los bits activados en el mapa de bits generado mediante la aplicación de las condiciones de la cláusula **where** sin acceder a la tabla.

Oracle soporta varios tipos de reuniones en el motor de ejecución: reuniones internas, externas, semirreuniones y antirreuniones (una antirreunión en Oracle devuelve las filas de la parte izquierda de la entrada que no coinciden con ninguna fila en la parte derecha de la entrada; esta operación se denomina antisemirreunión en otros libros). Evalúa cada tipo de reunión mediante uno de los tres métodos: reunión por asociación, reunión por mezcla–ordenación o reunión en bucle anidado.

27.4.2 Optimización

En el Capítulo 14 se ha estudiado el tema general de la optimización de la consulta. Aquí se trata la optimización en el contexto de Oracle.

27.4.2.1 Transformaciones de consultas

Oracle realiza la optimización de consultas en diferentes pasos. Uno es realizar varias transformaciones y reescrituras que fundamentalmente cambian la estructura de la consulta. Otro paso es realizar la selección de la ruta de acceso para determinar las rutas y los métodos y el orden de reunión. Dado que no todas las técnicas de transformación de consultas tienen garantizado su beneficio, Oracle realiza transformaciones basadas en el coste en las que éstas y la selección de camino se entrelazan. Para cada transformación generada se realiza una selección de la ruta de acceso y se genera una estimación del coste, aceptándola o rechazándola dependiendo del coste del plan de ejecución resultante.

Algunos de los tipos principales de transformaciones y reescrituras soportados por Oracle son los siguientes:

- **Mezcla de vistas**. La referencia de la vista en una consulta es reemplazada por la definición de la vista. Esta transformación no es aplicable a todas las vistas.
- **Mezcla compleja de vistas**. Oracle ofrece esta característica para ciertas clases de vistas que no están sujetas a la mezcla normal de vistas puesto que tienen un **group by** o **select distinct** en la definición de la vista. Si dicha vista se combina con otras tablas, Oracle puede conmutar las reuniones y las operaciones de ordenación y asociación utilizada por **group by** o **distinct**.
- **Subconsultas planas**. Oracle tiene una serie de transformaciones que convierten varias clases de subconsultas en reuniones, semirreuniones o antirreuniones.
- **Reescritura de vistas materializadas**. Oracle tiene la capacidad de reescribir una consulta automáticamente para aprovechar las vistas materializadas. Si alguna parte de la consulta se puede casar con una vista materializada existente, Oracle puede remplazar esta parte de la consulta con una referencia a la tabla en la cual la vista está materializada. Si es necesario, Oracle agrega condiciones de reunión u operaciones **group by** para preservar la semántica de la consulta. Si son aplicables varias vistas materializadas, Oracle recoge la que reduce la mayor cantidad de datos

que se tienen que procesar. Además, Oracle somete la consulta reescrita y la versión original al proceso completo de optimización produciendo un plan de ejecución y un coste asociado estimado para cada una. Oracle entonces decide si ejecutar la versión original o la reescrita de la consulta según la estimación del coste.

- **Transformación en estrella**. Oracle posee una técnica especial para evaluar las consultas en esquemas en estrella, conocidas como transformación en estrella. Cuando una consulta contiene una reunión de una tabla de hechos con tablas dimensionales y selecciones sobre los atributos de las tablas dimensionales, la consulta se transforma borrando la condición de la reunión entre la tabla de hechos y las tablas dimensionales y remplazando la condición de selección en cada tabla dimensional por una subconsulta del formulario:

  tabla_de_hechos$_i$ **in**
  (**select** *cp* **from** *tabla_dimensional*$_i$
  where <condiciones sobre *tabla_dimensional*$_i$ >)

  Se genera dicha subconsulta para cada tabla dimensional que tiene algún predicado restrictivo. Si la dimensión tiene un esquema en copo de nieve (véase el Apartado 18.3) la subconsulta contendrá una reunión de las tablas aplicables que forman la dimensión.

  Oracle utiliza los valores que son devueltos desde cada subconsulta para probar un índice sobre la columna de la tabla de hechos correspondiente, obteniendo un mapa de bits como resultado. Los mapas de bits generados desde distintas subconsultas se combinan con una operación **and** de mapas de bits. El mapa de bits resultante se puede utilizar para acceder a las filas de las tablas de hechos coincidentes. Por ello, solamente se accederá a las filas en la tabla de hechos que coinciden simultáneamente en las condiciones de las dimensiones restringidas.

  Tanto la decisión de si el uso de una subconsulta para una dimensión particular es ventajoso y la decisión de si la consulta reescrita es mejor que la original están basadas en la estimación de coste del optimizador.

27.4.2.2 Selección de la ruta de acceso

Oracle tiene un optimizador basado en el coste que determina el orden de la reunión, métodos de reunión y rutas de acceso. Cada operación que el optimizador considera tiene una función de coste asociada y el optimizador intenta generar la combinación de operaciones que tiene el coste global menor.

Para estimar el coste de una operación, el optimizador considera las estadísticas que se han calculado para los objetos del esquema tales como tablas e índices. La estadística contiene información sobre el tamaño del objeto, la cardinalidad, la distribución de datos de las columnas de la tabla y cosas similares. Para la estadística de columnas, Oracle soporta histogramas equilibrados en altura e histogramas de frecuencia. Para facilitar la recogida de las estadísticas del optimizador, Oracle puede supervisar la actividad de la modificación sobre tablas y sigue la pista de aquellas tablas que han sido objeto de suficientes cambios como para que pueda ser apropiado un nuevo cálculo de las estadísticas. Oracle también sigue las columnas que se utilizan en las cláusulas **where** de las consultas, lo que hace que sean candidatas potenciales para la creación del histograma. Con un solo comando es posible que Oracle actualice las estadísticas de las tablas que se han modificado sustancialmente. Oracle utiliza un muestreo para acelerar el proceso de recoger la nueva estadística y elige de forma automática el menor porcentaje de la muestra que sea adecuado. También determina si la distribución de las columnas marcadas merece la creación de histogramas; si la distribución está cerca de ser uniforme Oracle utiliza una representación más sencilla de la estadística de columnas.

En algunos casos puede ser imposible que el optimizador estime de forma precisa la selectividad de una condición en la cláusula **where** de una consulta simplemente a partir de las estadísticas almacenadas. Por ejemplo, la condición puede ser una expresión sobre una sola columna, como $f(col + 3) > 5$. Otro caso de consulta problemática es cuando se tienen varios predicados sobre columnas correlacionadas de alguna forma. Puede ser difícil determinar la selectividad combinada de estos predicados. Oracle trata estos problemas con el *muestreo dinámico*. El optimizador puede muestrear aleatoriamente

una pequeña parte de la tabla y aplicar todos los predicados relevantes a la muestra para determinar el porcentaje de filas que los cumplen.

Oracle utiliza el coste de CPU y E/S en disco en el modelo de coste en el optimizador. Para equilibrar los dos componentes almacena las medidas sobre la velocidad de CPU y rendimiento de E/S de disco como parte de la estadística del optimizador. El paquete de Oracle para recoger la estadística del optimizador calcula estas medidas.

Para consultas que involucran un número no trivial de reuniones, el espacio de búsqueda es un tema para el optimizador de consultas. Oracle soluciona este tema de varias formas. El optimizador genera un orden inicial de la reunión y después decide sobre los mejores métodos de la reunión y rutas de acceso para ese orden de la reunión. A continuación cambia el orden de las tablas y determina los mejores métodos de reunión y rutas de acceso para el nuevo orden y así sucesivamente, guardando el mejor plan que se ha encontrado hasta entonces. Oracle mantiene pequeña la optimización si el número de los distintos órdenes de la reunión que se han considerado es tan grande que el tiempo gastado en el optimizador puede ser grande comparado con el que se gastaría para ejecutar el mejor plan encontrado hasta entonces. Puesto que este corte depende del coste estimado para el mejor plan encontrado hasta entonces, es importante hallar pronto un buen plan de forma que el optimizador se pueda parar después de un pequeño número de órdenes de la reunión, resultando un mejor tiempo de respuesta. Oracle utiliza varias heurísticas para el orden inicial y aumentar la probabilidad de que el primer orden de reunión se considere bueno.

Por cada orden de reunión que se considera, el optimizador puede hacer pasadas adicionales por las tablas para decidir los métodos de reunión y las rutas de acceso. Tales pasadas adicionales capturarían efectos globales colaterales específicos sobre la selección de la ruta de acceso. Por ejemplo, una combinación específica de métodos de reunión y rutas de acceso pueden eliminar la necesidad de ejecutar una ordenación **order by**. Puesto que tal efecto lateral global puede no ser obvio cuando se consideran localmente los costes de los distintos métodos de reunión y de rutas de acceso, se utiliza una pasada separada que capture un efecto colateral específico para encontrar un posible plan de ejecución con un mejor coste conjunto.

27.4.3 SQL Tuning Advisor

Además del proceso normal de optimización, el optimizador de Oracle se puede usar en modo de ajuste como parte del asesor de ajuste de SQL (SQL Tuning Advisor) para generar planes de ejecución más eficientes de los que normalmente se generarían. Oracle supervisa la actividad de la base de datos y almacena automáticamente información sobre instrucciones SQL de gran coste en un repositorio de carga de trabajo. Estas instrucciones son las que usan más recursos porque se ejecutan gran número de veces o porque son inherentemente costosas. El asesor se puede usar para mejorar el rendimiento de estas instrucciones sugiriendo recomendaciones que se catalogan en:

- **Análisis estadístico.** Oracle comprueba si están disponibles y actualizadas las estadísticas necesarias para el optimizador y proporciona recomendaciones para recopilarlas.
- **Perfiles de SQL.** El perfil de una instrucción SQL es un conjunto de informaciones para ayudar al optimizador para que realice mejores decisiones la próxima vez que se optimice la instrucción. El optimizador puede generar a veces planes de ejecución ineficientes si no es capaz de estimar de forma precisa las cardinalidades y las selectividades, lo cual puede ocurrir como resultado de una correlación de datos o del uso de ciertos tipos de constructores. Al ejecutar el optimizador en modo de ajuste para generar un perfil, el optimizador intenta comprobar que sus suposiciones son correctas con el muestreo dinámico y la evaluación parcial de la instrucción SQL. Si encuentra pasos en el proceso de optimización en los que las suposiciones son incorrectas, genera un factor de corrección para cada paso que será parte del perfil. La optimización en el modo de ajuste puede consumir mucho tiempo, pero puede ser útil si el perfil mejora significativamente el rendimiento de la instrucción. Si se crea un perfil se almacena de forma persistente y se usa cada vez que se optimice la instrucción. Las perfiles se pueden usar para ajustar las instrucciones SQL sin cambiar su forma textual, lo cual es importante porque a menudo es imposible que el administrador de bases de datos modifique las instrucciones generadas por las aplicaciones.

- **Análisis de la ruta de acceso.** Oracle sugiere la creación de índices adicionales que podrían acelerar la ejecución de la instrucción usando el análisis del optimizador.
- **Análisis de la estructura SQL.** Oracle sugiere cambios en la estructura de las instrucciones SQL que permitirían una ejecución más eficiente.

Para tablas divididas el optimizador intenta ajustar las condiciones en la cláusula **where** de una consulta con el criterio de división de la tabla con el fin de evitar acceder a particiones que no son necesarias para el resultado. Por ejemplo, si una tabla se divide por el rango de fechas y la consulta se restringe a datos entre dos fechas específicas, el optimizador determina las particiones que contienen los datos entre las fechas específicas y asegura que solamente se accede a dichas particiones. Este escenario es muy común y la aceleración puede ser dramática si solamente es necesario un pequeño subconjunto de particiones.

27.4.4 Ejecución en paralelo

Oracle permite ejecutar en paralelo una única instrucción SQL mediante la división del trabajo entre varios procesos en una computadora multiprocesadora. Esta característica es especialmente útil para operaciones intensivas en cómputo que de otra forma se ejecutarían en un tiempo inaceptablemente largo. Ejemplos representativos son las consultas de apoyo para la toma de decisiones que necesitan procesar grandes cantidades de datos, cargas de datos en un almacén de datos y creación o reconstrucción de índices.

Con el fin de lograr una buena aceleración mediante el paralelismo es importante que el trabajo involucrado en la ejecución de la instrucción se divida en gránulos que se pueden procesar de forma independiente mediante los distintos procesadores en paralelo. Dependiendo del tipo de operación Oracle tiene diversas formas de dividir el trabajo.

Para operaciones que acceden a objetos base (tablas e índices) Oracle puede dividir el trabajo mediante segmentos horizontales de datos. Para algunas operaciones, tales como una exploración completa de una tabla, cada uno de dichos segmentos puede ser un rango de bloques—cada proceso de consulta en paralelo explora la tabla desde el bloque al comienzo del rango hasta el bloque al final. Para otras operaciones en una tabla dividida, como la actualización y borrado, el segmento podría ser una partición. Para inserciones en una tabla no dividida, los datos a insertar se dividen de forma aleatoria entre los procesos en paralelo.

Las reuniones se pueden realizar en paralelo de distintas formas. Una forma es dividir una de las entradas a la reunión entre procesos paralelos y permitir que cada proceso reúna su segmento con la otra entrada de la reunión; éste es el método de reunión con fragmentos y réplicas del Apartado 21.5.2.2. Por ejemplo, si una tabla grande se reúne con una pequeña mediante una reunión por asociación, Oracle divide la tabla grande entre los procesos y envía una copia de la tabla pequeña a cada proceso, la cual a su vez reúne su segmento con la tabla menor. Si ambas tablas son grandes sería prohibitivamente costoso enviar una de ellas a todos los procesos. En ese caso Oracle logra el paralelismo mediante la división de los datos entre los procesos con la asociación de los valores de las columnas de la reunión (el método de reunión por asociación dividida del Apartado 21.5.2.1). Cada tabla se explora en paralelo mediante un conjunto de procesos y cada fila en la salida se pasa a un proceso de un conjunto de procesos que van a ejecutar la reunión. El proceso que obtiene la fila se determina mediante una función de asociación sobre los valores de la columna de reunión. Por ello, cada proceso de reunión obtiene solamente las filas que podrían potencialmente coincidir y las filas correspondientes que no podrían ir a parar a procesos diferentes.

Oracle organiza en paralelo las operaciones de ordenación mediante los rangos de valores de la columna en la cual se ejecuta la ordenación (esto es, usando la ordenación de división por rangos del Apartado 21.5.1). A cada proceso que participa en la ordenación se le envían filas con los valores en este rango y ordena las filas en su rango. Para maximizar las ventajas del paralelismo las filas se tienen que dividir lo más equitativamente posible entre los procesos en paralelo, pero entonces surge el problema de determinar las fronteras de rango que generan una buena distribución. Oracle soluciona el problema

mediante un muestreo dinámico de un subconjunto de las filas en la entrada a la ordenación antes de decidir las fronteras del rango.

Los procesos involucrados en la ejecución en paralelo de una instrucción SQL consiste en un proceso coordinador y una serie de procesos servidores en paralelo. El coordinador es responsable de asignar trabajos a los servidores en paralelo y de recoger y devolver los datos a los procesos del usuario que enviaron la instrucción. El grado de paralelismo es el número de procesos servidores en paralelo que se asignan para ejecutar una operación primitiva como parte de la instrucción. El grado de paralelismo se determina mediante el optimizador, pero se puede reducir dinámicamente si la carga en el sistema aumenta.

Los servidores en paralelo operan sobre un modelo productor/consumidor. Cuando es necesario una secuencia de operaciones para procesar una instrucción, el conjunto productor de servidores ejecuta la primera operación y pasa los datos resultantes al conjunto de consumidores. Por ejemplo, si una exploración de tabla completa es seguida por una ordenación y el grado de paralelismo es 12 habría 12 servidores productores que ejecutan la exploración de la tabla y pasan el resultado a 12 servidores consumidores que ejecutan la ordenación. Si es necesaria una operación posterior, como otra ordenación, las funciones de los dos conjuntos de servidores se cambian. Los servidores que originalmente ejecutaban la exploración de la tabla adoptan la función de consumidores de la salida producida por la primera ordenación y la utilizan para ejecutar la segunda ordenación. Por ello se realiza una secuencia de operaciones pasando los datos entre dos conjuntos de servidores que alternan sus funciones como productores y consumidores. Los servidores se comunican entre sí mediante las memorias intermedias sobre hardware de memoria compartida y mediante las conexiones de red de alta velocidad sobre configuraciones MPP (sin compartimiento) y sistemas agrupados (discos compartidos).

Para sistemas sin compartimiento el coste para acceder a los datos en el disco no es uniforme entre los procesos. Un proceso que se ejecuta en un nodo que tiene acceso directo a un dispositivo puede procesar los datos sobre ese dispositivo más rápidamente que un proceso que tiene que recuperar los datos a través de la red. Oracle utiliza el conocimiento sobre la afinidad dispositivo a nodo y dispositivo a proceso—esto es, la capacidad de acceder a los dispositivos directamente—cuando distribuye el trabajo entre servidores en ejecución paralela.

27.5 Control de concurrencia y recuperación

Oracle soporta técnicas de control de concurrencia y recuperación que proporcionan una serie de características útiles

27.5.1 Control de concurrencia

El control de concurrencia multiversión de Oracle difiere de los mecanismos de concurrencia utilizados por la mayoría de los fabricantes de bases de datos. Para las consultas de sólo lectura se proporcionan instantáneas consistentes en lectura, que son vistas de la base de datos tal como existía en un cierto momento, y que contienen todas las actualizaciones que se comprometieron hasta ese momento y no el resto. Por ello, no se utilizan los bloqueos de lectura y las consultas de sólo lectura no interfieren con otra actividad de la base de datos en términos de bloqueos (esto es básicamente el protocolo de bloqueo multiversión en dos fases descrito en el Apartado 16.5.2).

Oracle soporta la consistencia de lectura en un nivel de instrucción y de transacción. Al comienzo de la ejecución de una instrucción o transacción (dependiendo del nivel de consistencia que se utilice) Oracle determina el número de cambio del sistema (System Change Number, SCN) actual. El SCN esencialmente actúa como una marca temporal donde el tiempo se mide en términos de compromisos de la base de datos, en lugar del tiempo de reloj.

Si en el transcurso de una consulta se determina que un bloque de datos tiene un SCN mayor que el que se está asociando a la consulta, es evidente que se ha modificado el bloque de datos después del SCN de la consulta original mediante alguna otra transacción y puede o no haberse comprometido. Por ello, los datos en el bloque no se pueden incluir en una vista consistente de la base de datos como existía a la hora del SCN de la consulta. En su lugar, se debe utilizar una versión anterior de los datos en el bloque—en concreto el que tenga el SCN mayor que no exceda el SCN de la consulta. Oracle recupera

la versión de los datos desde el segmento de retroceso (los segmentos de retroceso se describen en el Apartado 27.5.2). Por esta razón, supuesto que el segmento de retroceso es lo suficientemente grande, Oracle puede devolver un resultado consistente de la consulta incluso si los datos se han modificado varias veces desde que comenzara la ejecución de la consulta. Si el bloque con el SCN deseado ya no existe en el segmento de retroceso, la consulta devolverá un error. Habría una indicación de que el segmento de retroceso no se ha dimensionado adecuadamente, dada la actividad del sistema.

En el modelo de concurrencia de Oracle las operaciones de lectura no bloquean las operaciones de escritura y las operaciones de escritura no bloquean las operaciones de lectura, una propiedad que permite un alto grado de concurrencia. En concreto, el esquema permite consultas largas (por ejemplo, consultas de informes) para ejecutar en un sistema con una gran cantidad de actividad transaccional. Esta clase de escenario es normalmente problemático para sistemas de bases de datos donde las consultas utilizan bloqueos de lectura puesto que la consulta puede fallar al adquirirlos o bloquear grandes cantidades de datos por mucho tiempo evitando, por consiguiente, la actividad transaccional de los datos y reduciendo la concurrencia (una alternativa que se emplea en algunos sistemas es utilizar un grado inferior de consistencia, tal como la consistencia en grado dos, pero eso podría producir resultados inconsistentes en la consulta).

El modelo de concurrencia de Oracle se utiliza como base para las características *Flashback*. Estas características permiten a los usuarios establecer un cierto número SCN o tiempo de reloj en su sesión y ejecutar operaciones sobre los datos que existían en esa fecha (supuesto que los datos todavía existían en el segmento de retroceso). Normalmente, en un sistema de bases de datos, una vez que se ha realizado el cambio no hay forma de retroceder al estado anterior de los datos a menos que se realicen restauraciones desde copias de seguridad. Sin embargo, la recuperación de una base de datos muy grande puede ser muy costosa, especialmente si el objetivo es solamente recuperar algunos datos que ha sido borrados inadvertidamente por un usuario. Las características de Flashback proporcionan un mecanismo mucho más sencillo para tratar los errores del usuario. Entre ellas se encuentra la posibilidad de restaurar una tabla o una base de datos completa a un momento pasado sin recuperar de copias de seguridad, la posibilidad de realizar consultas sobre los datos que existieron en algún momento anterior, la posibilidad de realizar un seguimiento del cambio de una o más filas a lo largo del tiempo, y la posibilidad de examinar los cambios de la base de datos en el nivel de transacciones.

Oracle soporta dos niveles de aislamiento ANSI/ISO "con compromiso de lectura" y "secuenciable". No hay soporte para lecturas no actualizadas puesto que no hay necesidad. Los dos niveles de aislamiento corresponden a si se utiliza la consistencia de la lectura en el nivel de instrucción o en el nivel de transacción. El nivel se puede establecer para una sesión o para una transacción individual. La consistencia de lectura en el nivel de la instrucción es el nivel de aislamiento predeterminado.

Oracle utiliza un bloqueo en el nivel de las filas. Las actualizaciones de distintas filas no entran en conflicto. Si dos escritores intentan modificar la misma fila, uno espera hasta que el otro comprometa o retroceda y entonces puede devolver un error de conflicto de escritura o seguir y modificar la fila. Los bloqueos se mantienen mientras dura la transacción.

Además de los bloqueos en el nivel de las filas que evitan las inconsistencias debido a la actividad, el LMD de Oracle utiliza los bloqueos de tabla para evitar las inconsistencias debido a la actividad LDD. Estos bloqueos evitan que, por ejemplo, un usuario elimine una tabla mientras otro usuario tiene una transacción aún no comprometida que está accediendo a la tabla. Oracle no utiliza el dimensionamiento de bloqueos para convertir los bloqueos de filas a bloqueos de tabla con el propósito de su control de concurrencia normal.

Oracle detecta los interbloqueos automáticamente y los resuelve haciendo que retroceda una de las transacciones involucradas en el interbloqueo.

Oracle soporta transacciones autónomas que son transacciones independientes generadas con otras transacciones. Cuando Oracle invoca a una transacción autónoma genera una nueva transacción en un contexto separado. La nueva transacción se puede comprometer o retroceder antes de que el control vuelva a la transacción invocante. Oracle soporta varios niveles de anidamiento de transacciones autónomas.

27.5.2 Estructuras básicas de recuperación

Con el fin de comprender cómo se recupera Oracle de un fallo, tal como una caída del disco, es importante comprender las características básicas que están involucradas. Además de los archivos de datos que contienen las tablas e índices hay archivos de control, registros históricos rehacer, registros históricos rehacer archivados y segmentos de retroceso.

El archivo de control contiene varios metadatos que son necesarios para operar en la base de datos, incluyendo la información sobre las copias de seguridad.

Oracle registra cualquier modificación transaccional de una memoria intermedia de la base de datos en el registro histórico rehacer, que consiste en dos o más archivos. Registra la modificación como parte de la operación que la causa y sin considerar si la transacción finalmente se produce. Registra los cambios de los índices y segmentos de retroceso así como los cambios a la tabla de datos. Cuando se llenan los registros históricos rehacer se archivan mediante uno o varios procesos en segundo plano (si la base de datos se ejecuta en modo **archivelog**).

El segmento de retroceso contiene información sobre versiones anteriores de los datos (esto es, información para deshacer). Además de esta función en el modelo de consistencia de Oracle, la información se utiliza para restaurar la versión anterior de los datos cuando se deshace una transacción que ha modificado los datos.

Para poder recuperar un fallo de almacenamiento se debería realizar una copia de seguridad de los archivos de datos y archivos de control periódicamente. La frecuencia de la copia de seguridad determina el tiempo mayor de recuperación, puesto que lleva más tiempo la recuperación si la copia de seguridad es antigua. Oracle soporta copias de seguridad en caliente—copias de seguridad ejecutadas en una base de datos en línea que está sujeta a una actividad transaccional.

Durante la recuperación de una copia de seguridad, Oracle ejecuta dos pasos para alcanzar un estado consistente de la base de datos como existía antes del fallo. En primer lugar Oracle rehace las transacciones aplicando los archivos históricos rehacer (archivados) a la copia de seguridad. Esta acción lleva a la base de datos a un estado que existía en la fecha del fallo, pero no necesariamente un estado consistente puesto que los registros históricos deshacer incluyen datos no comprometidos. En segundo lugar, Oracle deshace las transacciones no comprometidas mediante el uso del segmento de retroceso. La base de datos está ahora en un estado consistente.

La recuperación en una base de datos que ha sido objeto de una actividad transaccional grande debido a la última copia de seguridad puede ser costosa en tiempo. Oracle soporta recuperación en paralelo en la cual se utilizan varios procesos para aplicar información de rehacer simultáneamente. Oracle proporciona una herramienta GUI, el gestor de recuperación (**Recovery Manager**), que automatiza la mayor parte de las tareas asociadas con copias de seguridad y recuperación.

27.5.3 Oracle Data Guard

Para asegurar una alta disponibilidad, Oracle proporciona la característica bases de datos en espera denominada *Data Guard* (esta característica es la misma que la de las copias de seguridad remotas, descrita en el Apartado 17.9). Una base de datos en espera es una copia de la base de datos normal que se instala en un sistema separado. Si ocurre un fallo catastrófico en el sistema principal el sistema en espera se activa y asume el control, minimizando el efecto del fallo en la disponibilidad. Oracle mantiene la base de datos en espera actualizada mediante la aplicación constante de archivos históricos rehacer archivados que se envían desde la base de datos principal. La base de datos de seguridad se puede usar en línea en modo sólo lectura y utilizarla para informes y consultas para el apoyo a la toma de decisiones.

27.6 Arquitectura del sistema

Siempre que una aplicación de base de datos ejecuta una instrucción SQL hay un proceso del sistema operativo que ejecuta código en el servidor de bases de datos. Oracle se puede configurar de forma que el proceso del sistema operativo esté *dedicado* exclusivamente a la instrucción que se está procesando o de forma que el proceso se pueda compartir entre varias instrucciones. La última configuración, conocida como *servidor compartido*, tiene propiedades diferentes respecto a la arquitectura del proceso y memoria.

En primer lugar se estudiará la arquitectura del servidor dedicado y, posteriormente, la arquitectura del servidor multienhebrado.

27.6.1 Servidor dedicado: estructuras de memoria

La memoria utilizada por Oracle se divide principalmente en tres categorías: áreas de código software, que son las partes de la memoria en las que reside el código del servidor Oracle, área global del sistema (System Global Area, SGA) y el área global del programa (Program Global Area, PGA).

Para cada proceso se asigna un PGA para albergar sus datos locales e información de control. Este área contiene espacio en pilas para diversos datos de la sesión y la memoria privada para la instrucción SQL que se está ejecutando. También contiene memoria para operaciones de ordenación y asociación que pueden ocurrir durante la evaluación de la instrucción. El rendimiento de estas operaciones depende de la cantidad de memoria disponible. Por ejemplo, una reunión por asociación que se pueda realizar en memoria será más rápida que si es necesario acceder a disco. Dado que hay un gran número de operaciones de ordenación y asociación activas simultáneamente (ya que hay varias consultas y varias operaciones en cada una de ellas), la decisión del tamaño de memoria a asignar para cada operación no es algo trivial, especialmente si fluctúa la carga de trabajo del sistema. Oracle permite que el administrador de bases de datos especifique un parámetro para la cantidad total de memoria que se debería considerar disponible para estas operaciones y decide dinámicamente la mejor forma de dividir esta mrmoria entre las operaciones activas para maximizar la productividad. El algoritmo de asignación de memoria conoce la relación entre la memoria y el rendimiento de las diferentes operaciones y trata de asegurar que la memoria disponible se use de la forma más eficaz posible.

SGA es un área de memoria para estructuras que son compartidas entre los usuarios. Está formada por varias estructuras principales, incluyendo:

- **Caché de memoria intermedia**. Esta caché mantiene bloques de datos a los que se accede frecuentemente (tablas e índices) en memoria para reducir la necesidad de ejecutar E/S a disco físico. Se usa la política "utilizado menos recientemente" salvo para los bloques a los que se acceda durante una exploración de tabla completa. Sin embargo, Oracle permite crear varias colas de memoria intermedia que tienen distintos criterios para la datación de los datos. Algunas operaciones Oracle omiten la caché de memoria intermedia y leen los datos directamente del disco.

- **Memoria intermedia de registro histórico rehacer**. Esta memoria intermedia contiene la parte del registro histórico rehacer que no se ha escrito todavía en el disco.

- **Cola compartida**. Oracle busca maximizar el número de usuarios que pueden utilizar la base de datos concurrentemente minimizando la cantidad de memoria que es necesaria para cada usuario. Un concepto importante en este contexto es la capacidad de compartir la representación interna de instrucciones SQL y el código procedimental escrito en PL/SQL. Cuando varios usuarios ejecutan la misma instrucción SQL pueden compartir la mayoría de estructuras de datos que representan el plan de ejecución de la instrucción. Solamente los datos que son locales a cada invocación específica de la instrucción necesitan mantenerse en una memoria privada.

  Las partes que se pueden compartir de las estructuras de datos que representan la instrucción SQL se almacenan en la cola compartida, incluyendo el texto de la instrucción. El almacenamiento en caché de instrucciones SQL en la cola compartida también se guarda en tiempo de compilación, puesto que una nueva invocación de la instrucción que ya está almacenada en caché no tiene que pasar por el proceso de compilación completo. La determinación de si una instrucción SQL es la misma que la existente en la cola compartida se basa en la coincidencia exacta del texto y en el establecimiento de ciertos parámetros de sesión. Oracle puede remplazar automáticamente las constantes en una instrucción SQL con variables vinculadas; las consultas futuras que son iguales salvo por los valores de constantes coincidirán con la consulta anterior en la cola compartida. La cola compartida también contiene cachés para información de diccionario y diversas estructuras de control.

27.6.2 Servidor dedicado: estructuras de proceso

Hay dos tipos de procesos que ejecutan código servidor Oracle: procesos servidor que procesan instrucciones SQL y procesos en segundo plano que ejecutan diversas tareas administrativas relacionadas con el rendimiento. Algunos de estos procesos son opcionales y en algunos casos se pueden utilizar varios procesos del mismo tipo por razones del rendimiento. Algunos de los tipos más importantes de procesos en segundo plano son:

- **Escritor de la base de datos**. Cuando una memoria intermedia se elimina de la caché de la memoria intermedia se debe volver a escribir en el disco si se ha modificado desde que se introdujo en la caché.

  Los procesos del escritor de la base de datos ejecutan esta tarea, lo que ayuda al rendimiento del sistema liberando espacio en la caché de la memoria intermedia.
- **Escritor del registro histórico**. El escritor del registro histórico procesa las entradas de escritura de la memoria intermedia del registro histórico rehacer al archivo del registro histórico rehacer en el disco. También escribe un registro de compromiso al disco siempre que se compromete una transacción.
- **Punto de revisión**. El proceso punto de revisión actualiza las cabeceras del archivo de datos cuando ocurre un punto de revisión.
- **Monitor del sistema**. Este proceso realiza la recuperación ante una caída en caso necesario. También ejecuta cierta administración del espacio para reclamar espacio no utilizado en espacios temporales.
- **Monitor de procesos**. Este proceso ejecuta recuperación de procesos para procesos del servidor que fallan, liberando recursos y ejecutando diversas operaciones de limpieza.
- **Recuperador**. El proceso recuperador resuelve los fallos y dirige la limpieza de transacciones distribuidas.
- **Archivador**. El archivador copia el archivo de registro histórico rehacer en línea a un registro histórico rehacer cada vez que se llena el archivo de registro histórico en línea.

27.6.3 Servidor compartido

La configuración de servidor compartido aumenta el número de usuarios que un número dado de procesos servidor puede soportar compartiendo los procesos servidor entre las instrucciones. Difiere de la arquitectura de servidor dedicado en los siguientes aspectos principales:

- Un proceso de envío en segundo plano encamina las solicitudes de usuarios al siguiente proceso servidor disponible. Al realizar esto utiliza una cola de solicitudes y una cola de respuestas en el SGA. El distribuidor pone una nueva solicitud en la cola de solicitudes donde será recogida por un proceso servidor. Un proceso servidor completa una solicitud, pone el resultado en la cola de respuestas para ser recogida por el distribuidor y ser devuelta al usuario.
- Puesto que un proceso servidor se comparte entre varias instrucciones SQL, Oracle no mantiene datos privados en el PGA. Almacena los datos específicos de la sesión en el SGA.

27.6.4 Oracle Real Application Clusters

Las agrupaciones de aplicaciones reales de Oracle (Oracle Real Application Clusters) es una característica que permite que varios ejemplares de Oracle se ejecuten en la misma base de datos (recuérdese que, en terminología de Oracle, un ejemplar es la combinación de procesos en segundo plano y áreas de memoria). Esta característica permite a Oracle ejecutarse en arquitecturas de hardware agrupadas y MPP (disco compartido y sin compartimiento). La capacidad de agrupar varios nodos tiene importantes ventajas en la dimensionabilidad y disponibilidad que son útiles en entornos OLTP y de almacén de datos.

Las ventajas de dimensionabilidad de la característica son obvias, puesto que más nodos significa más potencia de procesamiento. En las arquitecturas sin compartimiento la adición de nodos a un agrupamiento normalmente requiere la redistribución de los datos entre los nodos. Oracle usa una arquitectura de disco compartido donde todos los nodos tienen acceso a todos los datos y, como resultado, se pueden añadir más nodos a un RAC sin preocuparse de que los datos se dividan entre los nodos. Oracle optimiza más todavía el uso del hardware a través de las características tales como las reuniones por afinidad y por particiones.

RAC también se puede utilizar para lograr una alta disponibilidad. Si un nodo falla, los restantes todavía están disponibles para que la aplicación acceda a la base de datos. Las instancias restantes automáticamente retroceden las transacciones sin compromiso que están siendo procesadas en el nodo que falló con el fin de evitar un bloqueo de la actividad en el resto de nodos.

La ejecución de varias instancias en la misma base de datos da lugar a varios temas técnicos que no existen en un único ejemplar. Mientras que algunas veces es posible dividir una aplicación entre los nodos, de forma que los nodos raramente accedan a los mismos datos, siempre hay posibilidad de solapamiento, que afecta a la gestión de la caché. Para solucionarlo, Oracle usa la característica *mezcla de cachés*, que permite a los bloques de datos fluir directamente entre las cachés de distintos ejemplares mediante el uso de la interconexión, sin ser escritas a disco.

27.7 Réplica, distribución y datos externos

Oracle proporciona soporte para la réplica y las transacciones distribuidas con compromiso de dos fases.

27.7.1 Réplica

Oracle soporta varios tipos de réplica (véase el Apartado 19.2.1 para una introducción a la réplica). En su forma más sencilla los datos en un sitio maestro se duplican en otros sitios en forma de **instantáneas** (el término **instantánea** en este contexto no se debería confundir con el concepto de instantánea consistente en lectura en el contexto del modelo de concurrencia). Una instantánea no tiene que contener todos los datos maestros (puede, por ejemplo, excluir ciertas columnas de una tabla por razones de seguridad). Oracle soporta dos tipos de instantáneas: *sólo de lectura* y *actualizable*. Una instantánea actualizable se puede modificar en el sitio esclavo y las modificaciones se propagan hasta la tabla maestra. Sin embargo, las instantáneas sólo de lectura permiten un rango más amplio de definiciones de instantánea. Por ejemplo una instantánea de sólo lectura se puede definir en términos de conjuntos de operaciones sobre tablas en el sitio maestro.

Oracle también soporta varios sitios maestros para los mismos datos, donde todos los sitios maestros actúan como pares. Se puede actualizar una tabla duplicada en cualquiera de los sitios maestro y la actualización se propaga al resto de sitios. Las actualizaciones se pueden propagar de forma asíncrona o sincrónica.

Para la réplica asíncrona la información de actualización se envía mediante procesos por lotes al resto de sitos maestros y a continuación se aplican. Puesto que los mismos datos podrían estar sujetos a modificaciones conflictivas en sitios diferentes, se podría necesitar una resolución del conflicto basada en algunas reglas del negocio. Oracle proporciona una serie de métodos de resolución de conflictos incorporados y permite a los usuarios escribir el suyo propio si fuera necesario.

Con la réplica asíncrona una actualización de un sitio maestro se propaga de forma inmediata al resto de sitios. Si falla la transacción de actualización en cualquier sitio maestro, la actualización se deshace en todos los sitios.

27.7.2 Bases de datos distribuidas

Oracle soporta consultas y transacciones sobre varias bases de datos en distintos sistemas. Con el uso de pasarelas los sistemas remotos pueden incluir bases de datos que no sean de Oracle. Oracle tiene capacidades incorporadas para optimizar una consulta que incluya tablas en distintos sitios, recuperar los datos relevantes y devolver los resultados como si hubiera sido una consulta normal local. Oracle también soporta la emisión transparente de transacciones a varios sitios mediante un protocolo de compromiso en dos fases incorporado.

27.7.3 Orígenes de datos externos

Oracle tiene varios mecanismos para soportar orígenes de datos externos. El uso más común es el almacén de datos cuando se cargan normalmente grandes cantidades de datos desde un sistema transaccional.

27.7.3.1 SQL*Loader

Oracle tiene una utilidad de carga directa, SQL*Loader, que soporta cargas rápidas en paralelo de grandes cantidades de datos desde archivos externos. Soporta una serie de formatos de datos y puede ejecutar varias operaciones de filtrado sobre los datos que se están cargando.

27.7.3.2 Tablas externas

Oracle permite hacer referencia a los orígenes de datos externos, tales como archivos planos, en la cláusula **from** de una consulta como si fueran tablas normales. Una tabla externa se define mediante metadatos que describen los tipos de columna Oracle y la correspondencia entre los datos externos y dichas columnas. También es necesario un controlador de acceso para acceder a los datos externos. Oracle proporciona un controlador predeterminado para archivos planos.

La característica de tabla externa tiene el objetivo principal de operaciones de extracción, transformación y carga (ETL) en un entorno de almacén de datos. Los datos se pueden cargar en el almacén de datos desde un archivo plano utilizando

create table *tabla* **as**
    select ... **from** < tabla externa >
    where ...

Mediante la agregación de operaciones sobre los datos en la lista **select** o cláusula **where**, se pueden realizar transformaciones y filtrados como parte de la misma instrucción SQL. Puesto que estas operaciones se pueden expresar en SQL nativo o en funciones escritas en PL/SQL o Java, la característica de tabla externa proporciona un mecanismo potente para expresar todas las clases de operaciones de transformación y filtrado de los datos. Para la dimensionabilidad, se puede realizar en paralelo el acceso a la tabla externa con la ejecución en paralelo de Oracle.

27.8 Herramientas de gestión de bases de datos

Oracle proporciona a los usuarios una serie herramientas para la gestión del sistema y desarrollo de aplicaciones. Se ha hecho un gran énfasis en la versión Oracle 10*g* sobre el concepto de *administración*, es decir, en la reducción de la complejidad de todos los aspectos de la creación y administración de una base de datos Oracle. Este esfuerzo cubre una amplia variedad de áreas, incluyendo la creación de bases de datos, el ajuste, la gestión de espacio, de almacenamiento, la copia de seguridad y la recuperación, la gestión de memoria, los diagnósticos de rendimiento y la gestión de la carga de trabajo.

27.8.1 Repositorio automático de carga de trabajo

El repositorio automático de carga de trabajo (Automatic Workload Repository, AWR) es uno de los componentes principales de la infraestructura proporcionada por Oracle para reducir el esfuerzo de la administración. Oracle supervisa la actividad en el sistema de bases de datos y registra diferentes tipos de información relativa a las cargas de trabajo y al consumo de recursos. Esta información se usa en los diagnósticos de rendimiento y proporciona el fundamento de varios *asesores* que analizan diferentes aspectos de rendimiento y aconsejan cómo mejorarlo. Oracle tiene asesores para el ajuste de SQL, la creación de estructuras de acceso, como los índices y las vistas materializadas, y el dimensionamiento de la memoria. Oracle también proporciona asesores para la desfragmentación de segmentos y el redimensionamiento.

27.8.2 Gestión de los recursos de la base de datos

Un administrador de la base de datos necesita poder controlar cómo se divide la potencia de procesamiento entre los usuarios y grupos de usuarios. Algunos grupos pueden ejecutar consultas interactivas donde el tiempo de respuesta es crítico; otros pueden ejecutar informes largos que se pueden ejecutar como tareas de procesos por lotes en segundo plano cuando la carga del sistema sea baja. También es importante poder evitar que un usuario envíe inadvertidamente una consulta ad hoc extremadamente costosa que retrasará demasiado al resto.

La característica de gestión de los recursos de la base de datos de Oracle permite al administrador de la base de datos dividir los usuarios entre grupos consumidores de recursos con distintas prioridades y propiedades. Por ejemplo, un grupo de usuarios interactivos de alta prioridad pueden tener garantizado al menos un 60 por ciento de CPU. El resto, más alguna parte del 60 por ciento no utilizado por el grupo de alta prioridad, se asignaría entre los grupos de consumidores de recursos con baja prioridad. Un grupo de, realmente, baja prioridad podría tener asignado un 0 por ciento, lo que significaría que las consultas enviadas por este grupo se ejecutarían solamente cuando hubiera disponibles ciclos de CPU no utilizados. Se pueden establecer para cada grupo límites para el grado de paralelismos para la ejecución en paralelo.

El administrador de la base de datos también puede establecer límites de tiempo sobre cuánto tiempo máximo de ejecución se permite a una instrucción SQL. Cuando un usuario envía una instrucción, el gestor de recursos estima cuánto tiempo tardaría en ejecutarse y devuelve un error si la instrucción viola el límite. El gestor de recursos también puede limitar el número de sesiones de usuario que se pueden activar simultáneamente para cada grupo de consumidores de recursos. Otro recurso que puede controlar este gestor es el espacio para deshacer transacciones.

27.8.3 Oracle Enterprise Manager

El gestor corporativo de Oracle (Oracle Enterprise Manager) es la principal característica de Oracle para la gestión de sistemas de bases de datos. Proporciona una interfaz de usuario gráfica (GUI) sencilla de utilizar para la mayoría de tareas asociadas con la administración de bases de datos Oracle, incluyendo la configuración, la supervisión del rendimiento, la gestión de recursos, de la seguridad y acceso a los asesores.

27.9 Minería de datos

El componente de minería de datos de Oracle (Oracle Data Mining) proporciona varios algoritmos que incorporan el proceso de minería de datos dentro de la propia base de datos tanto para la construcción de un modelo sobre un conjunto de programas, como para su aplicación para evaluar los datos de producción reales. El hecho de que no sea necesario que los datos sean externos a la base de datos es una ventaja significativa sobre los motores separados de minería de datos. Tener que extraer e insertar grandes conjuntos de datos en un motor separado es incómodo y costoso, y además puede impedir que los nuevos datos se analicen justo después de que se introduzcan en la base de datos. Oracle proporciona funcionalidades para el aprendizaje supervisado y sin supervisar, tales como:

- Clasificación.
- Regresión.
- Importancia de los atributos.
- Agrupamiento.
- Análisis de mercado.
- Extracción de características.
- Minería de texto.
- Bioinformática (BLAST).

Oracle proporciona dos interfaces para la minería de datos: una de Java y otra basada en el lenguaje procedimental PL/SQL de Oracle. Una vez que se construye un modelo en una base de datos de Oracle, se puede enviar o implatar en otras bases de datos de Oracle. Oracle puede importar y exportar modelos usando una representación tanto en PL/SQL como en PMML (Predictive Model Markup Language). Los modelos PMML generados por Oracle se pueden usar con otras herramientas que soporten PMML.

Notas bibliográficas

Se puede encontrar información actualizada, incluyendo documentación, sobre productos Oracle en http://www.oracle.com y http://technet.oracle.com.

La indexación extensible en Oracle se describe en Srinivasan et al. [2000b] y en Srinivasan et al. [2000b]. Srinivasan et al. [2000a] describen las tablas organizadas con índices. Banerjee et al. [2000], Murthy y Banerjee [2003], y Krishnaprasad et al. [2004] describen el soporte XML en Oracle8i. Bello et al. [1998] describen las vistas materializadas en Oracle. Antoshenkov [1995] describe la técnica de compresión de mapas de bits alineadas por bytes utilizada en Oracle; véase también Johnson [1999]. Lahiri et al. [2001b] describen la mezcla de caché de RAC.

La recuperación en Oracle se describe en Joshi et al. [1998] y en Lahiri et al. [2001a]. La mensajería y las colas en Oracle se describen en Gawlick [1998]. Witkowski et al. [2003a] y Witkowski et al. [2003b] describen la cláusula MODEL.

Los algoritmos de gestión de memoria para la ordenación y asociación se describen en Dageville y Zait [2002]. Poess y Potapov [2003] describen la compresión de tablas de Oracle. El algoritmo de conjuntos de elementos frecuentes para el análisis de mercado de Oracle Data Mining se describe en Li y Mozes [2004].

El ajuste automático de SQL se describe en Dageville et al. [2004]. Cruanes et al. [2004] describen la ejecución paralela de Oracle.

CAPÍTULO 28

DB2 Universal Database de IBM

Sriram Padmanabhan
Centro de Investigación de IBM T. J. Watson

La familia de productos DB2 Universal Database de IBM es el buque insignia de servidores de base de datos y las familias de productos para inteligencia de negocio, integración de información y gestión de contenidos ampliamente reconocidos por su robustez. El servidor de base de datos DB2 Universal Database Server está disponible en gran número de plataformas hardware y sistemas operativos. La lista de las plataformas del servidor soportadas incluye sistemas de alto nivel como *mainframes*, procesadores masivamente paralelos (MPP) y grandes servidores multiprocesadores simétricos (SMP); sistemas medios como SMPs de cuatro y ocho vías; estaciones de trabajo; e incluso pequeños dispositivos de bolsillo. Los sistemas operativos que están soportados incluyen variantes de Unix tales como Linux, AIX, Solaris y HP-UX, así como Windows 2000, Windows XP, MVS, VM, OS/400, entre otros. DB2 Everyplace Edition soporta sistemas operativos tales como PalmOS y Windows CE. DB2 Cloudscape es un motor de base de datos Java puro que se puede incluir fácilmente en servidores de aplicaciones y otras aplicaciones. Estas aplicaciones pueden migrarse sin problemas de plataformas de gama baja a grandes servidores debido a la portabilidad de los interfaces y servicios de DB2. Además del motor del núcleo de la base de datos, la familia DB2 incluye otros productos que proporcionan herramientas, administración, replicación, acceso a datos distribuido, acceso ubicuo a datos, OLAP, y muchas otras características. En la Figura 28.1 se describen los diferentes productos de la familia.

28.1 Visión general

El origen de DB2 se remonta al proyecto System R en el Centro de investigación de Almadén (Almaden Research Center) de IBM (entonces denominado Laboratorio de investigación de San José, IBM San Jose Research Laboratory). El primer producto DB2 se comercializó en 1984 sobre la plataforma *mainframe* de IBM, seguido tiempo después por versiones para otras. IBM ha mejorado continuamente DB2 en áreas tales como procesamiento de transacciones (registro histórico de escritura anticipada y los algoritmos de recuperación ARIES), procesamiento y optimización de consultas (proyecto de investigación Starburst), procesamiento en paralelo (DB2 Parallel Edition), soporte para bases de datos activas (restricciones y disparadores), técnicas avanzadas de consultas y almacenes de datos tales como vistas materializadas, agrupaciones multidimensionales, características "autónomas" y soporte del modelo relacional orientado a objetos (ADTs, UDFs).

Puesto que IBM soporta un gran número de plataformas de servidor y de sistema operativo, el motor de base de datos DB2 consiste en cuatro bases de código diferentes: (1) Linux, Unix y Windows, (2) z/OS, (3) VM, y (4) OS/400. Todos estos soportan un subconjunto común de lenguaje de definición de

- Servidores de bases de datos
  - –DB2 UDB para Linux, Unix, Windows
  - –DB2 UDB para z/OS
  - –DB2 UDB para OS/400
  - –DB2 para VM/VSE
- Inteligencia de negocio
  - –DB2 Data Warehouse Edition
  - –DB2 OLAP Server
  - –DB2 Alphablox
  - –DB2 CubeViews
  - –DB2 Intelligent Miner
  - –DB2 Query Patroller
- Integración de datos
  - –DB2 Information Integrator
  - –DB2 Replication
  - –DB2 Connect
  - –Omnifind (para Enterprise Search)
- Gestión de contenidos
  - –DB2 Content Manager
  - –IBM Enterprise Content Manager
- Desarrollo de aplicaciones
  - –IBM Rational Application Developer Studio
  - –DB2 Forms para z/OS
  - –QMF
- Herramientas de gestión de bases de datos
  - –DB2 Control Center
  - –DB2 Admin Tool para z/OS
  - –DB2 Performance Expert
  - –DB2 Query Patroller
  - –DB2 Visual Explain
- Acceso móvil a datos
  - –DB2 Cloudscape
  - –DB2e (EveryPlace)

Figura 28.1 Familia de productos DB2.

datos, de SQL y de los interfaces de administración. Sin embargo, los motores presentan características algo diversas debido a los orígenes de cada plataforma. Este capítulo está centrado en el motor DB2 Universal Database (UDB) para Linux, Unix y Windows. Se reseñarán las características específicas de interés en otros sistemas DB2 cuando se considere apropiado.

La última versión de la base de datos universal (UDB, Universal Database) DB2 para Linux, Unix y Windows es la 8.2. Esta versión contiene varias características que mejoran la ampliabilidad, la disponibilidad y la robustez general del motor DB2. En el área de la ampliabilidad, dos características significativas son las *tablas de consultas materializadas* y las *agrupaciones multidimensionales*. Con respecto a la disponibilidad, se describen las mejoras en las áreas de utilidades en línea y las réplicas. Además, esta versión proporciona características autónomas tales como el *asesor de diseño* y el ajuste y supervisión automáticos de memoria. Éstas y otras características adicionales se describirán en los apartados correspondientes.

28.2 Herramientas de diseño de bases de datos

La mayor parte de las herramientas de diseño de base de datos y herramientas CASE se pueden usar para diseñar una base de datos DB2. En particular, las herramientas de modelado de datos tales como ERWin y Rational Rose permiten al diseñador generar sintaxis LDD específica de DB2. Por ejemplo, la herramienta UML Data Modeler de Rational Rose puede generar instrucciones **create distinct type** del LDD específico de DB2 para tipos definidos por el usuario y usarlos posteriormente en definiciones de columnas. La mayor parte de herramientas de diseño también soportan una característica de ingeniería inversa que lee las tablas del catálogo de DB2 y construye un diseño lógico para manipulaciones adicionales. Las herramientas pueden generar restricciones e índices.

DB2 proporciona constructores SQL para soportar muchas características lógicas de bases de datos, tales como restricciones, disparadores y recursión. De igual forma, DB2 soporta ciertas características físicas de bases de datos tales como espacios de tablas, colas de memoria intermedia, y particionamiento mediante el uso de instrucciones SQL. Para ello, la herramienta Centro de control de DB2 permite a los diseñadores o administradores emitir las instrucciones LDD apropiadas. Otra herramienta, *db2look*, permite al administrador obtener un conjunto completo de instrucciones LDD para una base de datos incluyendo espacios de tablas, tablas, índices, restricciones, disparadores y funciones que crean una reproducción exacta del esquema de la base de datos para pruebas o réplica.

```
select xmlelement(name 'PO',
                  xmlattributes(idprod, fechapedido),
                  (select xmlagg(xmlelement(name 'producto',
                                    xmlattributes(idproducto, cantidad, fechaenvío),
                                    (select xmlelement(name 'descproducto',
                                                xmlattributes(nombre, precio))
                                    from producto
                                    where producto.idproducto = líneaproducto.idproducto)))
                  from líneaproducto
                  where líneaproducto.idprod = pedidos.idprod))
from pedidos
where pedidos.idprod= 349;
```

Figura 28.2 Consulta XML en SQL de DB2.

El Centro de control de DB2 incluye una serie de herramientas de diseño y administración. Respecto al diseño, el Centro de control proporciona una vista en árbol de un servidor, sus bases de datos, tablas, vistas y todos los demás objetos. También permite que los usuarios definan nuevos objetos, creen consultas SQL ad hoc y visualicen resultados de consultas. Las herramientas de diseño para ETL, OLAP, réplicas y federación también están integradas en el Centro de control. Toda la familia DB2 soporta el Centro de control para la definición de bases de datos, así como otras herramientas relacionadas. DB2 también proporciona módulos predeterminados para el desarrollo de aplicaciones en IBM Rational Application Development, así como en Microsoft Visual Studio.

28.3 Variaciones y extensiones de SQL

DB2 soporta un amplio conjunto de características SQL para varios aspectos del procesamiento de bases de datos. Muchas de las características y sintaxis de DB2 han proporcionado la base de los estándares SQL-92 o SQL-99. Este apartado resalta las características XML del modelo relacional orientado a objetos y de integración de aplicaciones en la versión 8 de DB2 UDB.

28.3.1 Características XML

Se ha incluido en DB2 un extenso conjunto de funciones XML. A continuación se muestran algunas funciones XML importantes.

- **xmlelement**. Construye una etiqueta elemento con un nombre dado. Por ejemplo, **xmlelement**(*libro*) crea el elemento libro.
- **xmlattributes**. Construye el conjunto de atributos de un elemento.
- **xmlforest**. Construye una secuencia de elementos XML a partir de los argumentos.
- **xmlconcat.** Devuelve la concatenación de un número variable de argumentos XML.
- **xmlserialize**. Proporciona una versión del argumento serializada orientada a caracteres.
- **xmlagg.** Devuelve la concatenación de un conjunto de valores XML.
- **xml2clob.** Construye una representación del XML de tipo objeto de gran tamaño de caracteres (**clob**). Este **clob** puede ser recuperado por aplicaciones SQL.

Las funciones XML incorporadas en SQL proporcionan una gran capacidad en la manipulación de XML. Por ejemplo, supóngase que se necesita construir un documento XML de pedido de compra de las tablas *pedidos*, *líneaproducto* y *producto* para el número de pedido 349. En la Figura 28.2 se muestra una consulta SQL con extensiones XML que se pueden usar para crear dicho pedido de compra. La salida resultante se muestra en la Figura 28.3.

```
<PO idprod = "349" fechapedido = "2004-10-01">
      <item idproducto="1", cantidad="10", fechaenvío="2004-10-03">
      <descproducto nombre = "IBM ThinkPad T41", precio = "1000.00 EUR¨/>
      </item>
</PO>
```

Figura 28.3 Pedido de compra en XML para el producto 349.

DB2 integra una característica de XML extendido que proporciona procedimientos almacenados y funciones definidas por el usuario para almacenar y manipular XML como grandes objetos de tipo carácter o atributos de varias tablas. Se puede acceder a estos objetos de XML mediante SQL con las extensiones XML antes mencionadas o con las funciones del extensor de XML.

28.3.2 Soporte para tipos de datos

DB2 proporciona soporte para tipos de datos definidos por el usuario. Se pueden definir tipos de datos *distintos* o *estructurados*. Los tipos de datos *distintos* están basados en tipos de datos incorporados en DB2. Sin embargo, los usuarios pueden definir semánticas adicionales o alternativas para estos nuevos tipos. Por ejemplo, el usuario puede definir un tipo de datos *distinto* llamado *euro*, usando

create distinct type *euro* **as decimal**(9,2)

Posteriormente, se puede crear un campo (por ejemplo, *precio*) en una tabla cuyo tipo sea *euro*. Este campo se puede usar en predicados de consultas como en el siguiente ejemplo:

select *producto* **from** *ventas_Europa*
where *precio* > *euro*(1000)

Los tipos de datos estructurados son objetos complejos que normalmente están formados por dos o más atributos. Por ejemplo, el siguiente código declara un tipo de datos estructurado denominado *t_departamento*:

```
create type t_departamento as
      (nombredept varchar(32),
       directordept varchar(32),
       número integer)
mode db2/sql

    create type t_punto as
          (coord_x float,
           coord_y float)
    mode db2/sql
```

Los tipos estructurados se pueden usar para definir *tablas con tipos*.

create table *departamento* **of** *t_departamento*

Con el LDD se puede crear una jerarquía de tipos y tablas que pueden heredar métodos específicos y privilegios. Los tipos estructurados también se pueden usar para definir atributos anidados dentro de una columna de una tabla. Aunque este tipo de definiciones violaría las reglas de normalización, pueden ser convenientes para aplicaciones orientadas a objetos que se basan en la encapsulación y en métodos bien definidos sobre los objetos.

28.3.3 Funciones y métodos definidos por el usuario

Otra característica importante es que los usuarios pueden definir sus propias funciones y métodos. Estas funciones se pueden incluir posteriormente en instrucciones y consultas SQL. Las funciones pueden

```
create function db2gse.GsegeFilterDist (
        operación integer, g1XMín double, g1XMáx double,
        g1YMín double, g1YMáx double, dist double,
        g2XMín double, g2XMáx double, g2YMín double,
        g2YMáx double )
    returns integer
    specific db2gse.GsegeFilterDist
    external name 'db2gsefn!gsegeFilterDist'
    language C
    parameter style db2 sql
    deterministic
    not fenced
    threadsafe
    called on null input
    no sql
    no external action
    no scratchpad
    no final call
    allow parallel
    no dbinfo;
```

Figura 28.4 Definición de una FDU.

generar escalares (único atributo) o tablas (fila multiatributo) como resultado. Los usuarios pueden definir funciones (escalares o de tablas) mediante el uso de la instrucción **create function**. Pueden escribir las funciones en lenguajes de programación comunes tales como C y Java o lenguajes de guiones tales como REXX y PERL. Las funciones definidas por el usuario (FDU) pueden operar en los modos separado (*fenced*) y compartido (*unfenced*). En el modo separado las funciones se ejecutan mediante una hebra separada en su propio espacio de dirección. En el modo compartido se permite al agente de procesamiento de la base de datos ejecutar la función en el espacio de direcciones del servidor. Las FDU pueden definir un área de trabajo donde pueden mantener variables locales y estáticas en invocaciones diferentes. Por tanto, las FDU pueden realizar manipulaciones complejas de las filas intermedias que son su entrada. En la Figura 28.4 se muestra una definición de una FDU en DB2, *db2gse.GsegeFilterDist*, que apunta a un método externo específico el cual realiza realmente la operación.

Otra característica son los métodos asociados a un objeto, los cuales definen su comportamiento. A diferencia de las FDU, los métodos están asociados con tipos de datos estructurados particulares y se registran mediante el uso de la instrucción **create method**.

28.3.4 Objetos de gran tamaño

Las nuevas aplicaciones de las bases de datos requieren la manipulación de texto, imágenes, vídeo y otros tipos de datos típicos de gran tamaño. DB2 soporta estos requisitos proporcionando tres tipos de objetos de gran tamaño (LOB, Large Object) distintos. Cada LOB puede ocupar hasta 2 gigabytes. Los objetos de gran tamaño en DB2 son (1) objetos en binario (Binary Large Objetcs, BLOBs), (2) objetos de caracteres de un único byte (Character Large Objects, CLOBs) y (3) objetos de caracteres de dos bytes (Double Byte Character Large Objects, **dbclob**s). DB2 organiza estos LOBs como objetos separados, con cada fila en la tabla manteniendo punteros a sus LOBs correspondientes. Los usuarios pueden registrar FDUs que manipulen estos LOBs según los requisitos de la aplicación.

28.3.5 Extensiones de índices y restricciones

Una característica reciente de DB2 proporciona un constructor **create index extension** que ayuda a crear índices sobre atributos con tipos de datos estructurados mediante la generación de claves a partir de los tipos de datos estructurados. Por ejemplo, un usuario puede crear un índice en un atributo cuyo tipo es *t_departamento* definido anteriormente mediante la generación de claves con el nombre del departamen-

```
create index extension db2gse.índice_espacial(
                        gS1 double, gS2 double, gS3 double)
from source key(geometry db2gse.ST_Geometry)
generate key using
        db2gse.GseGridIdxKeyGen(geometry..srid,
        geometry..xMín, geometry..xMáx,
        geometry..yMín, geometry..yMáx,
        gS1, gS2, gS3)

with target key(srsId integer,
          lvl integer, gX integer, gY integer, xMín double,
          xMáx double, yMín double, yMáx double)
search methods <condiciones> <acciones>
```

Figura 28.5 Extensión de índice espacial en DB2.

to. El extensor espacial de DB2 utiliza el método de extensión de índice para crear índices sobre los datos espaciales, como se muestra en la Figura 28.5.

Finalmente, DB2 también proporciona un rico conjunto de características de verificación de restricciones para imponer la semántica de los objetos tales como unicidad, validez y herencia.

28.3.6 Servicios Web

En la Versión 8, DB2 permite integrar servicios Web como productor o consumidor. Se puede definir un servicio Web para invocar DB2 usando instrucciones SQL. La llamada al servicio Web resultante es procesada por un motor de servicios Web incorporado en DB2, que genera la correspondiente respuesta SOAP. Por ejemplo, si hay un servicio Web denominado *ObtenerActividadReciente(id_cliente)* que llama a la siguiente instrucción SQL, el resultado será la última transacción realizada por el cliente.

```
select id_trn, importe, fecha
from transacciones
where id_cliente = <input>
order by fecha
fetch first 1 row only;
```

La siguiente instrucción SQL muestra cómo DB2 actúa como un cliente de un servicio Web. En este ejemplo, la función *ObtenerCotización()* definida por el usuario es un servicio Web. DB2 realiza la llamada al servicio Web mediante un motor incorporado de servicios Web. En este caso, *ObtenerCotización* devuelve un valor de cotización para cada *id_acción* que aparece en la tabla *cartera*.

```
select id_acción, ObtenerCotización(id_acción)
from cartera
```

28.3.7 Colas de mensajes

DB2 también soporta el producto de IBM Websphere MQ mediante la definición de las FDU apropiadas, tanto para interfaces de lectura como de escritura. Estas FDU pueden incluirse en instrucciones SQL para la lectura o escritura sobre colas de mensajes.

28.4 Almacenamiento e indexación

La arquitectura de almacenamiento e indexación de DB2 está formada por la capa de sistema de ficheros o gestión de disco, los servicios para gestionar las memorias intermedias, objetos de datos tales como tablas, LOBs, objetos índices y gestores de concurrencia y recuperación. Este apartado muestra una visión

general de la arquitectura de almacenamiento. Además, en el siguiente apartado se describe una nueva característica en la versión 8 de DB2 denominada agrupación multidimensional.

28.4.1 Arquitectura de almacenamiento

DB2 proporciona abstracciones de almacenamiento para gestionar tablas de base de datos lógicas en entornos multinodo (paralelo) y multidisco. Se pueden definir *grupos de nodos* para soportar la división de la tabla en conjuntos especificados de nodos en un sistema multinodo. Esto permite flexibilidad al asignar particiones de tabla a nodos diferentes en un sistema. Por ejemplo, las tablas de gran tamaño se pueden dividir entre todos los nodos en un sistema, mientras que las tablas pequeñas pueden residir en un único nodo.

Dentro de un nodo, DB2 usa *espacios de tablas* para organizar las tablas. Un espacio de tablas consiste en uno o más *contenedores* que son referencias a directorios, dispositivos o archivos. Un espacio de tablas puede contener cero o más objetos de base de datos tales como tablas, índices o LOBs. La Figura 28.6 ilustra estos conceptos. En esta figura se definen dos espacios de tablas para un grupo de nodos. Al espacio de tablas *RECHUMANOS* se le asignan cuatro contenedores mientras que al espacio de tablas *smallPLAN* sólo uno. Las tablas *empleado* y *departamento* se encuentran en el espacio de tablas *RECHUMANOS* mientras que la tabla *proyecto* está en el espacio de tablas *smallPLAN*. La distribución de datos asigna fragmentos (extensiones) de las tablas *empleado* y *departamento* a los contenedores del espacio de tablas *RECHUMANOS*. DB2 permite al administrador crear tanto espacios de tablas gestionados por el *sistema* como por el *SGBD*. Los espacios de tablas gestionados por el sistema (System-managed spaces, SMS) son directorios o sistemas de archivo que mantiene el sistema operativo subyacente. En un SMS, DB2 crea objetos archivo en los directorios y asigna datos a cada uno de los archivos. Los espacios de tablas gestionados por el SGBD (Data Managed Spaces, DMS) son dispositivos en bruto o archivos preasignados que son controlados por DB2. El tamaño de estos contenedores nunca puede crecer o disminuir. DB2 crea mapas de asignación y gestiona el espacio de tablas DMS. En ambos casos la unidad de espacio de almacenamiento es una extensión de páginas. El administrador puede elegir el tamaño de la extensión para un espacio de tabla.

DB2 soporta la distribución en distintos contenedores. Por ejemplo, cuando se insertan los datos en una tabla recientemente creada, DB2 asigna la primera extensión a un contenedor. Una vez que la extensión está llena asigna los siguientes datos al siguiente contenedor por turnos rotatorios. La distribución proporciona dos ventajas significativas: E/S paralela y equilibrio de carga.

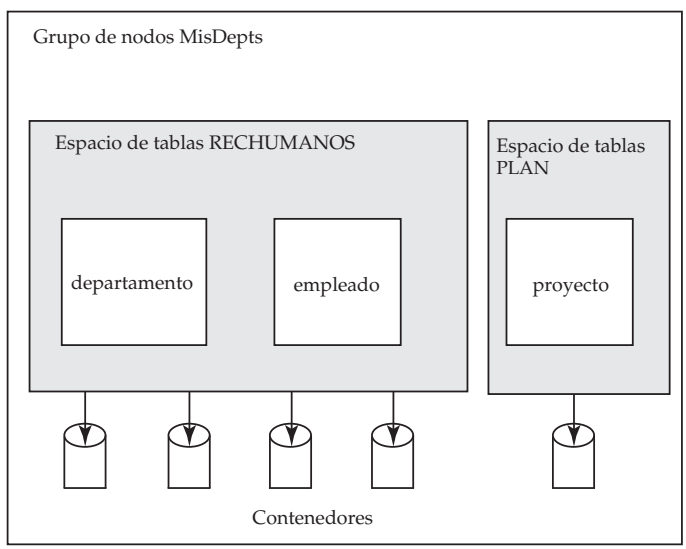


Figura 28.6 Espacios de tablas y contenedores en DB2.

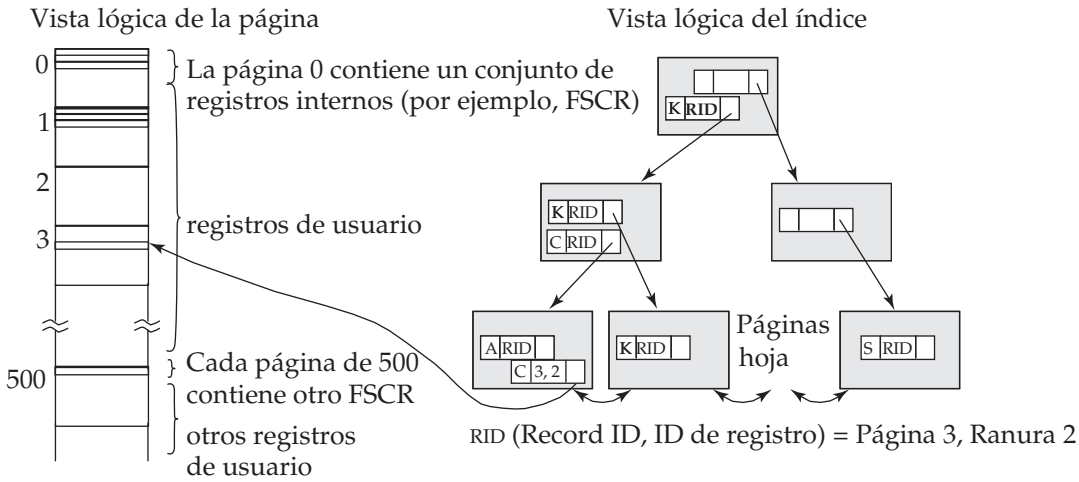


Figura 28.7 Vista lógica de las tablas e índices en DB2.

28.4.2 Colas de memorias intermedias

Se puede asociar una o varias memorias intermedias con cada espacio de tablas para gestionar diferentes objetos tales como datos e índices. Una memoria intermedia es un área de datos compartida que mantiene copias de objetos en memoria. Estos objetos habitualmente están organizados en páginas para gestionar la memoria intermedia. DB2 permite la definición de memorias intermedias usando instrucciones SQL. La versión 8 de DB2 incluye la posibilidad de aumentar o disminuir el tamaño de las memorias intermedias de forma interactiva, o bien automáticamente si se selecciona la opción **automatic** en parámetro de configuración de memorias intermedias. Un administrador puede añadir más páginas a una memoria intermedia o bien disminuir su tamaño sin detener la actividad de la base de datos.

create bufferpool <cola-mem-intermedia> ....
alter bufferpool <cola-mem-intermedia> **size** <n>

DB2 también soporta la preextracción y escrituras asíncronas mediante el uso de hebras separadas. El componente de gestión de datos de DB2 desencadena la preextracción de páginas de datos y de índices según los patrones de acceso de las consultas. Por ejemplo, una exploración de una tabla siempre desencadena la preextracción de páginas de datos. La exploración del índice puede desencadenar la preextracción de páginas de índices así como las páginas de datos si se está accediendo de una forma agrupada. El número de preextracciones concurrentes, así como el tamaño de la preextracción, son parámetros configurables que es necesario establecer según el número de discos o contenedores en el espacio de tablas.

28.4.3 Tablas, registros e índices

DB2 organiza los datos relacionales como registros en las páginas. La Figura 28.7 muestra la vista lógica de una tabla y un índice asociado. La tabla consiste en un conjunto de páginas. Cada página consiste en un conjunto de registros (tanto registros de datos del usuario como registros especiales del sistema). La página cero de la tabla contiene registros del sistema especiales sobre la tabla y su estado. DB2 usa un registro del mapa de espacio denominado registro de control de espacio libre (Free Space Control Record, FSCR) para encontrar el espacio libre en la tabla. El registro FSCR normalmente contiene un mapa de espacio de 500 páginas. Una entrada FSCR consiste en unos pocos bits que proporcionan una indicación aproximada del porcentaje de espacio libre en la página. El algoritmo de inserción o actualización debe validar las entradas del FSCR realizando una comprobación física del espacio disponible en la página.

Los índices también se organizan como páginas que contienen registros índice y punteros a páginas hijas y hermanas. DB2 proporciona soporte para los mecanismos de índices de árbol B^+. El índice de árbol B^+ contiene páginas internas y páginas hoja. Los índices contienen punteros bidireccionales en el

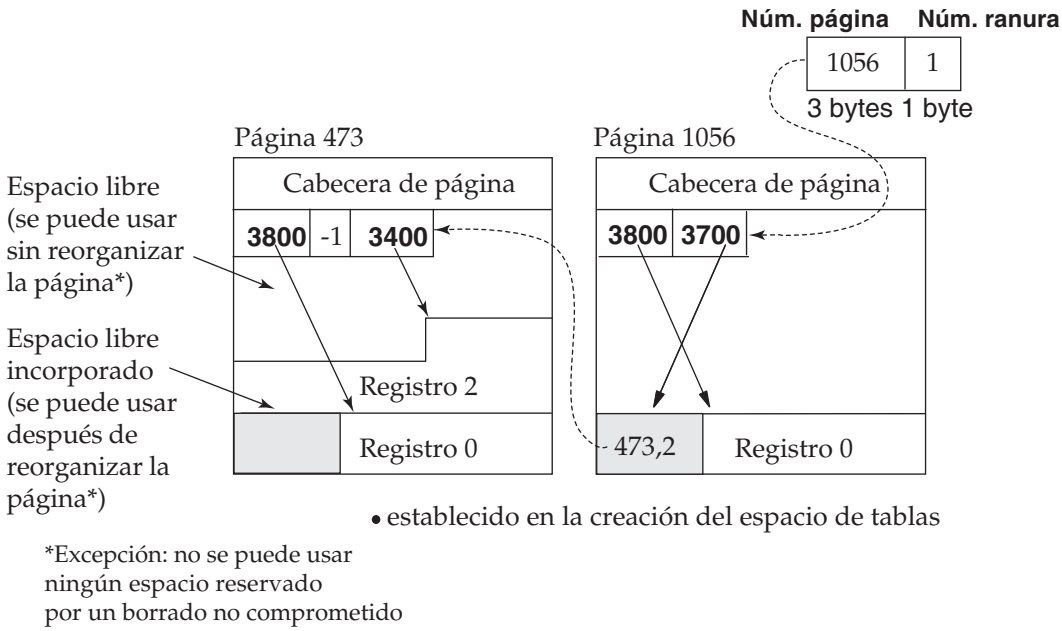


Figura 28.8 Diseño de las páginas de datos y de los registros en DB2.

nivel hoja para soportar exploraciones hacia delante y hacia atrás. Las páginas hoja contienen entradas de índice que apuntan a los registros de la tabla. Cada registro de una tabla se puede identificar unívocamente usando su información de página y de ranura, que se denominan *identificador de registro* o RID (Record ID).

DB2 admite "columnas incluidas" (include) en la definición del índice, como en:

create unique index I1 **on** T1 (C1) **include** (C2)

Las columnas del índice incluidas permiten a DB2 extender el uso de las técnicas de procesamiento "sólo con índices" siempre que sea posible. Se pueden usar directivas adicionales tales como **minpctused** y **pctfree** para controlar la unión de páginas de índices y su asignación de espacio inicial.

La Figura 28.8 muestra el formato de datos típico en DB2. Cada página de datos contiene una cabecera y un directorio de ranuras. El directorio de ranuras es un array de 255 entradas que apuntan a los desplazamientos de los registros en la página. La figura muestra que el número de página 473 contiene el registro cero en el desplazamiento 3800 y el registro 2 en el desplazamiento 3400. La página 1056 contiene un registro 1 en el desplazamiento 3700, que es realmente un puntero hacia delante al registro <473,2>. Por ello el registro <473,2> es un registro de desbordamiento que fue creado por una operación de actualización del registro <1056,1> original. DB2 soporta distintos tamaños de página tales como 4 KB, 8 KB, 16 KB y 32 KB. Sin embargo, cada página puede contener solamente 255 registros de usuario. Los tamaños de página mayores son útiles en aplicaciones tales como almacén de datos donde la tabla contiene muchas columnas. Los tamaños de página menores son útiles para datos operacionales con frecuentes actualizaciones.

28.5 Agrupación multidimensional

Esta sección proporciona una breve visión general de las principales características de las agrupaciones multidimensionales (Multidimensional Clustering, MDC). Con la agrupación multidimensional, una tabla DB2 puede crearse especificando una o varias claves como dimensiones para las que se agruparán los datos de la tabla. Existe una nueva cláusula denominada **organize by dimensions** para este cometido. Por ejemplo, la siguiente instrucción LDD describe una tabla de ventas organizada tomando los atributos *idAlmacén*, *year*(*fechaPedido*) e *idProducto* como dimensiones.

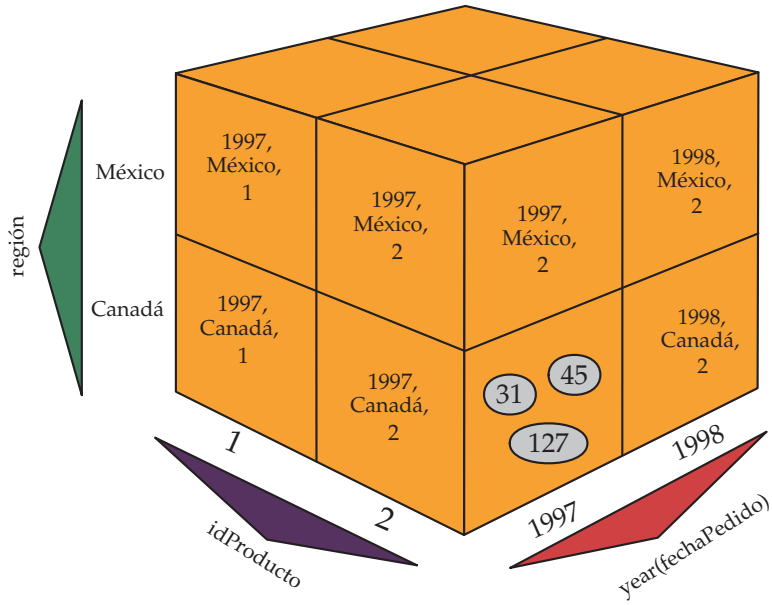


Figura 28.9 Vista lógica de la disposición física de una tabla MDC.

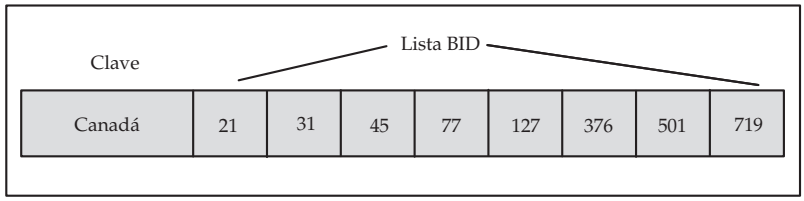
```
create table ventas(idAlmacén int,
                    fechaPedido date,
                    fechaEnvío date,
                    fechaRecepción date,
                    región int,
                    idProducto int,
                    precio float
                    añoPedido int generated always as year(fechaPedido))
organized by dimensions (región, añoPedido, idProducto)
```

Cada una de estas dimensiones puede estar formada por una o varias columnas, del mismo modo que los índices. De hecho, se crea automáticamente un 'índice dimensional de bloques' (descrito más adelante) para cada una de las dimensiones especificadas, y se usa para acceder rápida y eficientemente a los datos. Si es necesario, se crea un índice de bloque compuesto que contiene todas las columnas clave de la dimensión y se usa para mantener la agrupación de datos durante las actividades de inserción y actualización.

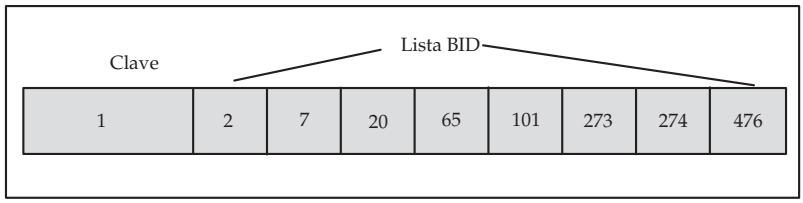
Cada combinación única de valores dimensionales forma una "celda" lógica que está físicamente organizada como bloques de páginas, donde un bloque es un conjunto de páginas consecutivas en disco. El conjunto de los bloques que contienen páginas con datos que verifican un cierto valor de clave de uno de los índices de bloque de dimensión se denomina "banda". Cada página de la tabla forma parte de exactamente un bloque, y todos los bloques de la tabla están formados por el mismo número de páginas, que se denomina tamaño de bloque. DB2 asocia el tamaño de bloque con el tamaño de extensión del espacio de tablas, de forma que los límites de los bloques estén alineados con los límites de las extensiones.

La Figura 28.9 ilustra estos conceptos. Esta tabla MDC se encuentra agrupada por las dimensiones *year*(*fechaPedido*)[1], *región* e *idProducto*. La figura muestra un cubo lógico sencillo con solamente dos valores por cada atributo de dimensión. En realidad, los atributos de dimensión pueden extenderse fácilmente a gran número de valores sin necesidad de administración. Las celdas lógicas están representadas en la figura por los cubos más pequeños. Los registros de la tabla se almacenan en bloques, que contienen una extensión de páginas consecutivas en disco. En el diagrama se representa cada bloque con una elipse gris, numerada de acuerdo al orden lógico de extensiones realizadas en la tabla. Sólo se muestran

1. Las dimensiones se pueden crear usando una función generada.



(a) Entrada del índice dimensional de bloques para la región 'Canadá'.



(b) Entrada del índice dimensional de bloques para el identificador de producto 1

Figura 28.10 Entradas de las claves del índice a bloques.

algunos bloques de datos para la celda identificada por los valores de dimensión <1997,Canadá,2>. Una columna o una fila de la retícula representa una banda para una dimensión particular. Por ejemplo, todos los registros que contienen el valor "Canadá" en la dimensión *región* se encuentran en los bloques contenidos en la banda definida por la columna "Canadá" del cubo. De hecho, cada bloque en esta banda solamente contiene registros que contienen "Canadá" en el campo *región.*

28.5.1 Índices de bloque

En el ejemplo anterior, se crea un índice dimensional de bloques sobre cada uno de los atributos *región*, *year*(*fechaPedido*) e *idProducto*. Cada índice dimensional de bloques se estructura de la misma forma que un árbol B tradicional excepto que, en el nivel de las hojas, las claves apuntan a un *identificador de bloque* (Block Identifier, BID) en lugar de apuntar a un identificador de registro (Record Identifier, RID). Como cada bloque contiene potencialmente muchas páginas de registros, los índices de bloques son mucho más pequeños que los índices RID, y necesitan actualizarse solamente cuando se añade un nuevo bloque a la celda, o cuando se vacían y eliminan bloques existentes en la celda. Una banda, o el conjunto de bloques que contienen páginas con un valor clave particular en una dimensión, se representan en el índice dimensional de bloques asociado mediante una lista de BIDs para ese valor clave. La Figura 28.10 ilustra las bandas de bloques para valores específicos de las dimensiones *región* e *idProducto*, respectivamente.

En el ejemplo anterior, para encontrar la banda que contiene todos los registros con valor "Canadá" en la dimensión *región* hay que buscar este valor clave en el índice dimensional de bloques y encontrar una clave como se muestra en la Figura 28.10a. Esta clave apunta al conjunto exacto de BIDs para ese valor particular.

28.5.2 Mapas de bloques

La tabla también tiene asociado un mapa de bloques. Este mapa registra el estado de cada bloque de la tabla. Un bloque puede estar en diferentes estados tales como **in use**, **free**, **loaded** y **requiring constraint enforcement**. La capa de gestión de datos usa el estado del bloque para determinar diversas opciones de procesamiento. La Figura 28.11 muestra un ejemplo de mapa de bloques de una tabla.

El elemento 0 en el mapa de bloques representa el bloque 0 en el diagrama MDC de la tabla. El estado de disponibilidad del bloque es "U", que indica que está en uso. Sin embargo, es un bloque especial y no contiene registros de usuario. Los bloques 2, 3, 9, 10, 13, 14 y 17 no se están usando en la tabla, por lo que se consideran "F" (libres) en el mapa de bloques. Los bloques 7 y 18 se han cargado recientemente a la tabla. El bloque 12 fue cargado anteriormente y requiere la comprobación de restricciones.

0	1	2	3	4	5	6	7	8	9	19	11	12	13	14	15	16	17	18	19
U	U	F	F	U	U	U	L	U	F	F	U	C	F	F	U	U	F	L	...

Figura 28.11 Entradas del mapa de bloques.

28.5.3 Consideraciones de diseño

Un aspecto crucial de las MDCs es la elección del conjunto correcto de dimensiones para agrupar una tabla y el tamaño correcto de bloque para minimizar el uso de espacio. Si las dimensiones y el tamaño de bloque se eligen apropiadamente, los beneficios de la agrupación se traducen en significativas ventajas de rendimiento y mantenimiento. Por otra parte, si se eligen incorrectamente, el rendimiento se puede degradar y el uso de espacio puede ser sensiblemente peor. Hay muchas técnicas de ajuste que se pueden explotar para organizar la tabla. Entre ellas se encuentran la variación del número de dimensiones, la granularidad de una o varias dimensiones, el tamaño de bloque (tamaño de extensión) y el tamaño de página del espacio de tablas que contiene la tabla. Se pueden usar una o varias de estas técnicas para identificar la mejor organización de la tabla.

28.5.4 Impacto sobre las técnicas existentes

Es natural preguntarse si la nueva característica de MDC tiene un impacto adverso o invalida algunas de las características existentes en DB2 para las tablas clásicas. Todas las características existentes tales como índices RID secundarios, restricciones, disparadores, vistas materializadas definidas y opciones de procesamiento de consultas están disponibles para las tablas MDC. Por consiguiente, las tablas MDC se comportan exactamente como las tablas clásicas excepto por sus aspectos mejorados de agrupación y procesamiento.

28.6 Procesamiento y optimización de consultas

El compilador de consultas de DB2 transforma las consultas en un árbol de operaciones. DB2 usa entonces el árbol de operadores de la consulta en tiempo de ejecución para el procesamiento. DB2 soporta un rico conjunto de operadores de consulta que permiten considerar mejores estrategias de procesamiento y proporcionan flexibilidad en la ejecución de consultas complejas.

Las Figuras 28.12 y 28.13 muestran una consulta y su plan de consulta asociado. Se trata de una consulta compleja representativa (consulta 5) de la prueba TPC-H y contiene varias reuniones y agregaciones. El plan de consulta en este ejemplo es bastante simple puesto que solamente se definen pocos

```
– – 'Consulta TPCD de volumen de proveedor local (Q5)';
select nac_nombre, sum(lp_precioextendido*(1-lp_descuento)) as ingresos
from tpcd.cliente, tpcd.pedidos, tpcd.linea_pedido,
     tpcd.proveedor, tpcd.nación, tpcd.región
where cli_clave_cliente = ped_clave_cliente and
      ped_clave_pedido = lp_clave_pedido and
      lp_clave_proveedor = prv_clave_proveedor and
      cli_clave_nación = prv_clave_nación and
      prv_clave_nación = nac_clave_nación and
      nac_clave_región = reg_clave_región and
      reg_nombre = 'MIDDLE EAST' and
      ped_fecha_pedido >= date('1995-01-01') and
      ped_fecha_pedido < date('1995-01-01') + 1 year
group by nac_nombre
order by ingresos desc;
```

Figura 28.12 Consulta SQL.

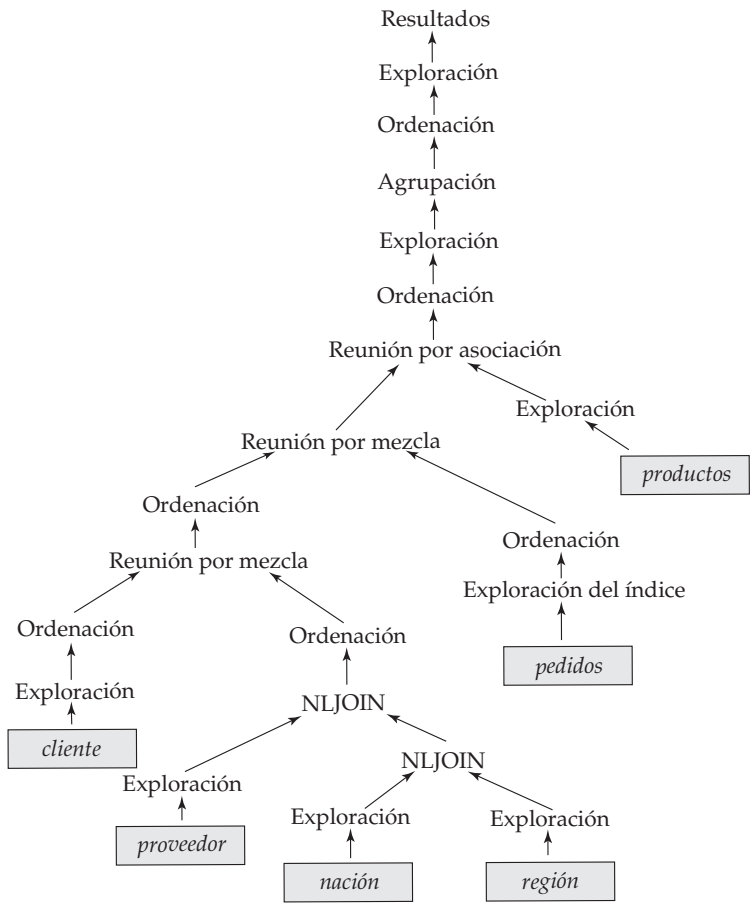


Figura 28.13 Plan de consulta de DB2 (explicación gráfica).

índices y no están disponibles para esta consulta estructuras auxiliares como las vistas materializadas. DB2 proporciona varias características de "explicación" del plan incluyendo una potente característica visual en el Centro de control que puede ayudar a los usuarios a comprender los detalles del plan de ejecución de la consulta. El plan de consulta en la figura está basado en la explicación visual de la consulta. La explicación visual permite al usuario comprender los costes y otras propiedades relevantes de las distintas operaciones de un plan de consulta.

DB2 transforma todas las consultas e instrucciones SQL, sin importar lo complejas que sean, en un árbol de consulta. La base u operadores hoja del árbol de consulta manipulan los registros en tablas de base de datos. Estas operaciones también se denominan *métodos de acceso*. Las operaciones intermedias del árbol incluyen operaciones del álgebra relacional tales como reuniones, operaciones de conjuntos y agregaciones. La raíz del árbol produce los resultados de la consulta o instrucción SQL.

28.6.1 Métodos de acceso

DB2 soporta un conjunto detallado de métodos de acceso sobre tablas relacionales, incluyendo:

- **Exploración de tabla**. Con este método, el más básico, se accede a todos los registros en la tabla página por página.
- **Exploración de índice**. DB2 usa un índice para seleccionar los registros específicos que satisfacen la consulta. Accede a los registros usando los RIDs en el índice. DB2 detecta las posibilidades de la preextracción de las páginas de datos cuando observa un patrón de acceso secuencial.

- **Exploración de índice de bloques**. Es un nuevo método de acceso para tablas MDC. Se usa uno de los índices de bloque para explorar un conjunto específico de bloques de datos. DB2 accede y procesa los bloques seleccionados mediante operaciones de exploración de tablas de bloques.

- **Sólo con el índice**. Este tipo de exploración se usa cuando el índice contiene todos los atributos que requiere la consulta. Por ello es suficiente una exploración de las entradas de índice y no hay necesidad de extraer los registros. La técnica sólo con el índice es normalmente una buena elección desde el punto de vista del rendimiento.

- **Lista de preextracción**. Este método de acceso es una buena elección para una exploración de índices no agrupada con un número significativo de RIDs. DB2 recoge los RIDs de los registros relevantes usando una exploración de índices, después ordena los RIDs por el número de página y finalmente realiza una extracción de los registros de forma ordenada desde las páginas de datos. El acceso ordenado cambia el patrón E/S de aleatorio a secuencial y también ofrece posibilidades de preextracción.

- **Conjunción de índices de bloques y de registros**. DB2 usa este método cuando determina que se puede usar más de un índice para restringir el número de registros satisfactorios en una tabla base. Procesa el índice más selectivo para generar una lista de BIDs o RIDs. Después procesa el siguiente índice selectivo para devolver los BIDs o RIDs que encuentra. Un BID o RID requiere más procesamiento solamente si está presente en la intersección (operación AND) de los resultados de la exploración del índice. El resultado de una operación AND del índice es una pequeña lista de RIDs o BIDs que se usan para extraer los registros correspondientes desde la tabla base.

- **Ordenación de índices de bloque y de registro**. Esta estrategia es una buena elección si se pueden usar dos o más índices para satisfacer los predicados de la consulta que se combinan usando la operación OR. DB2 elimina los BIDs o RIDs duplicados realizando una ordenación y después extrae el conjunto de registros resultante. La disyunción de índices se ha extendido para considerar combinaciones de índices de bloque y RIDs.

DB2 normalmente envía todos los predicados de selección y proyección de una consulta a los métodos de acceso. Además DB2 envía ciertas operaciones tales como la ordenación y la agregación, siempre que es posible, con el fin de reducir el coste.

La característica de MDC aprovecha el nuevo conjunto de mejoras en los métodos de acceso para exploraciones de índices de bloques, preextracción de índices de bloques, conjunción de índices de bloques y disyunción de índices de bloques para procesar los bloques de datos.

28.6.2 Operaciones de reunión, agregación y de conjuntos

DB2 soporta una serie de técnicas para las operaciones de reunión, agregación y de conjuntos. Para la reunión DB2 puede elegir entre técnicas de bucles anidados, mezcla-ordenación y de asociación. Para describir las operaciones binarias de reuniones y de conjuntos se usará la notación de las tablas "externas" e "internas" para distinguir los dos flujos de entrada. La técnica de bucles anidados es útil si la tabla interna es muy pequeña o se puede acceder usando un índice sobre un predicado de reunión. Las técnicas de reunión de mezcla-ordenación y reunión por asociación son útiles para reuniones que involucran tablas internas y externas grandes. DB2 implementa las operaciones de conjuntos mediante el uso de técnicas de ordenación y mezcla. La técnica de mezcla elimina los duplicados en el caso de la unión mientras que los no duplicados se eliminan en el caso de intersección. DB2 también soporta operaciones de reunión externa de todas las clases.

DB2 procesa las operaciones de agregación en modo impaciente o de envío siempre que sea posible. Por ejemplo, puede realizar la agregación mientras que ordena la entrada de la agregación en el grupo por columnas. Los algoritmos de reunión y agregación aprovechan el procesamiento superescalar en CPUs modernas usando técnicas orientadas a bloques y conscientes de la caché de memoria.

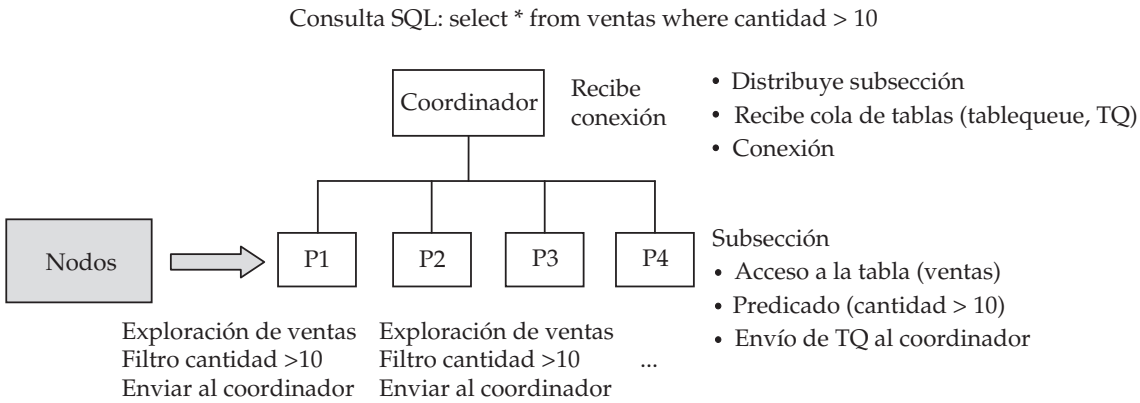


Figura 28.14 Procesamiento de consultas DB2 MPP usando envío de funciones.

28.6.3 Soporte para el procesamiento de SQL complejo

Uno de los aspectos más importantes de DB2 es que usa la infraestructura de procesamiento de la consulta de forma extensible para soportar operaciones SQL complejas. Las operaciones SQL complejas incluyen soporte para subconsultas profundamente anidadas y correlacionadas, así como restricciones, integridad referencial y disparadores. Como la mayor parte de estas acciones están incluidas dentro del plan de consulta, DB2 está preparado para soportar una gran cantidad de restricciones y acciones. Las restricciones y comprobaciones de integridad se construyen como operaciones del árbol de consulta a partir de las instrucciones SQL de inserción, borrado o actualización. DB2 también soporta el mantenimiento de vistas materializadas mediante el uso de disparadores incorporados.

28.6.4 Procesamiento de consultas en multiprocesadores

DB2 extiende el conjunto base de operaciones de consulta con primitivas de intercambio de datos y control para soportar los modos SMP, MPP y SMP por agrupaciones del procesamiento de consultas. DB2 usa una abstracción *tabla-cola* para el intercambio de datos entre hebras sobre distintos nodos o sobre el mismo nodo. La tabla-cola es una memoria intermedia que redirige los datos a receptores apropiados mediante el uso de métodos de difusión, uno a uno o multidifusión dirigida. Las operaciones de control crean hebras y coordinan la operación de distintos procesos y hebras.

En todos estos modos DB2 usa un proceso coordinador para controlar las operaciones de colas y la reunión del resultado final. Los procesos de coordinación también pueden ejecutar algunas acciones globales de procesamiento de la base de datos si es necesario. Un ejemplo es la operación de agregación global para combinar los resultados de agregación local. Los subagentes o hebras esclavos ejecutan las operaciones base en uno o más nodos. En el modo SMP los subagentes usan memoria compartida para sincronizarse entre sí cuando comparten datos. En un MPP, los mecanismos de tabla-cola proporcionan memoria intermedia y control de flujo para la sincronización entre distintos nodos durante la ejecución. DB2 usa técnicas exhaustivas de optimización y procesa consultas de forma eficiente en entornos MPP o SMP. La Figura 28.14 muestra una consulta simple ejecutándose en un sistema MPP de cuatro nodos. En este ejemplo, la tabla *ventas* está dividida entre los cuatro nodos $P_1, \ldots, P_4$. La consulta se ejecuta produciendo agentes que se ejecutan en cada uno de los nodos para explorar y filtrar las filas de la tabla *ventas* en ese nodo (denominado envío de funciones) y las filas resultantes se envían al nodo coordinador.

28.6.5 Optimización de consultas

El compilador de consultas de DB2 utiliza una representación interna de la consulta, denominada Query Graph Model (QGM, modelo de grafos de consultas) con el fin de ejecutar transformaciones y optimizaciones. Después de analizar la instrucción SQL, DB2 ejecuta transformaciones semánticas sobre el QGM para hacer cumplir las restricciones, integridad referencial y los disparadores. El resultado de estas transformaciones es un QGM mejorado. Seguidamente DB2 intenta ejecutar transformaciones de *reescritura* de la consulta que se consideran beneficiosas en la mayoría de las consultas. Se activan las

reglas de reescritura, si son aplicables, para ejecutar las transformaciones requeridas. Los ejemplos de transformaciones de reescritura incluyen (1) descorrelación de subconsultas correlacionadas, (2) transformación de subconsultas en reuniones donde sea posible, (3) trasladar las operaciones **group by** bajo las reuniones si es aplicable y (4) uso de vistas materializadas para fragmentos de la consulta original.

El optimizador de consultas usa QGM mejorado y transformado como su entrada para la optimización. El optimizador se basa en el coste y usa un entorno extensible, controlado por reglas. Se puede configurar el optimizador para operar a distintos niveles de complejidad. En el nivel más alto usa un algoritmo de programación dinámica para considerar todas las opciones del plan de consulta y elige el plan de coste óptimo. En un nivel intermedio el optimizador no considera ciertos planes o métodos de acceso (por ejemplo, disyunción de índices) así como algunas reglas de reescritura. En el nivel inferior de complejidad el optimizador usa una heurística impaciente simple para elegir un buen, aunque no necesariamente óptimo, plan de consulta. El optimizador emplea modelos detallados de las operaciones de procesamiento de la consulta (teniendo en cuenta detalles tales como tamaño de la memoria y preextracción) para obtener estimaciones adecuadas de los costes de E/S y CPU. Depende de la estadística de los datos para estimar la cardinalidad y selectividades de las operaciones. DB2 permite a un usuario generar histogramas de distribuciones en el nivel de las columnas y combinaciones de columnas mediante el uso de la utilidad *runstats*. Los histogramas contienen información sobre las apariciones del valor más frecuente así como sobre las distribuciones de frecuencia basadas en los cuantiles de los atributos. El optimizador de consultas utiliza estas estadísticas. El optimizador genera el que considera el mejor plan de consulta interno y entonces lo convierte en hebras de operadores y estructuras de datos asociados de la consulta para su ejecución mediante el motor de procesamiento de consultas.

28.7 Tablas de consultas materializadas

Las vistas materializadas están permitidas en la versión 8 de DB2 en Linux, Unix y Windows, así como en las plataformas z/OS. Cualquier definición de vista sobre una o varias tablas o vistas puede ser una vista materializada. La utilidad de este tipo de vistas estriba en que mantienen una copia persistente de los datos de la vista para proporcionar un procesamiento de consultas más rápido. En DB2 estas vistas materializadas se denominan tablas de consultas materializadas (materializad query tables, MQTs). Las MQTs se especifican usando una instrucción **create table** como la mostrada en el ejemplo de la Figura 28.15.

En DB2 las MQTs pueden referirse a otras MQTs para crear un árbol o un bosque de vistas dependientes. Las MQTs son muy ampliables ya que pueden ser divididas en un entorno MPP y pueden tener claves de agrupación MDC. Las MQTs resultan de máximo valor si el motor de la base de datos es capaz de encaminar perfectamente consultas hacia la MQT, y también cuando el motor de la base de datos puede mantenerlas eficientemente siempre que sea posible. DB2 proporciona ambas características.

28.7.1 Encaminamiento de consultas a MQTs

La infraestructura del compilador de consultas de DB2 resulta ideal para impulsar toda la potencia de las MQTs. El modelo interno del QGM permite al compilador comparar la consulta de entrada con las definiciones de MQTs disponibles y elegir las MQTs apropiadas. Después de la comparación, el compilador considera varias opciones de optimización. Entre ellas se encuentran tanto la consulta base como las versiones de reencaminamiento de MQTs apropiadas. El optimizador itera a través de estas opciones

```
create table empleado_departamento(id_departamento integer, id_empleado integer,
                                   nombre_empleado varchar(100), id_jefe integer) as
    select id_departamento, id_empleado, nombre_empleado, id_jefe
    from empleado, departmento
        data initially deferred
        refresh immediate – – (o deferred)
        maintained by user – – (o system)
```

Figura 28.15 Tablas de consultas materializadas de DB2.

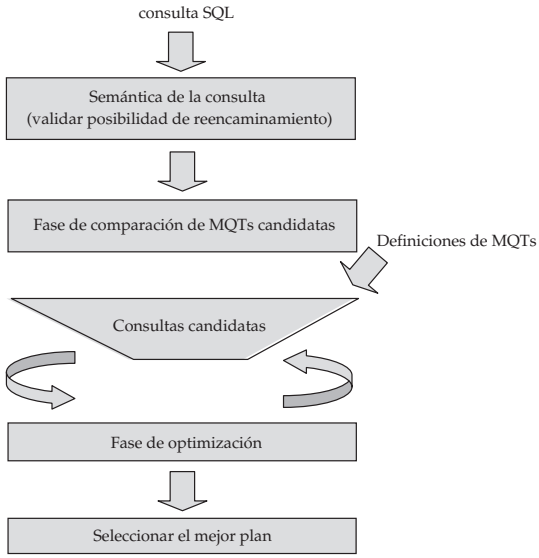


Figura 28.16 Comparación y optimización de MQTs en DB2.

antes de elegir la versión de ejecución óptima. El flujo completo de reencaminamiento y optimización se muestra en la Figura 28.16.

28.7.2 Mantenimiento de MQTs

Las MQTs sólo son útiles si el motor de la base de datos proporciona técnicas eficientes de mantenimiento. Hay dos dimensiones en el mantenimiento: tiempo y coste. En la dimensión temporal, las dos opciones son *inmediato* y *diferido*. DB2 soporta ambas. Si se selecciona mantenimiento inmediato, se crean disparadores internos que se compilan en las instrucciones de inserción, actualización o borrado de los objetos fuente para procesar las actualizaciones de las MQTs dependientes. En el caso del mantenimiento diferido, las tablas actualizadas se pasan a modo integridad y se debe ejecutar una instrucción **refresh** explícitamente para realizar la actualización. Con respecto a la dimensión de coste, se puede elegir entre **incremental** o **completo**. En el mantenimiento incremental sólo se utilizan para el mantenimiento las filas que han sido actualizadas recientemente. El mantenimiento completo actualiza toda la MQT desde las fuentes. La matriz de la Figura 28.17 muestra ambas dimensiones y las opciones más útiles en cada una de ellas. Por ejemplo, mantenimiento inmediato y completo sólo son compatibles si las fuentes son extremadamente pequeñas. DB2 también permite que las MQTs sean mantenidas por el usuario (**user**). En este caso, la actualización de las MQTs viene determinada por procesos explícitamente realizados por los usuarios usando SQL o utilidades.

Los siguientes comandos (mandatos en DB2) proporcionan un ejemplo sencillo de mantenimiento diferido de la vista materializada *emp_dept* después de una operación de carga de una de sus fuentes.

load from datosnuevos.txt **of type del**
insert into *empleado*;

refresh table *empleado_departamento*

Elecciones	*Incremental*	*Completa*
Inmediata	Sí, Después de insert/update/delete	Normalmente no
Diferida	Sí, Después de la carga	Sí

Figura 28.17 Opciones para el mantenimiento de MQTs en DB2.

28.8 Características autónomas de DB2

La versión 8.2 de DB2 UDB proporciona varias características para simplificar el diseño y la administración de bases de datos. La informática autónoma engloba un conjunto de técnicas que permiten a un entorno informático administrarse a sí mismo y reducir dependencias externas frente a cambios en la seguridad externa e interna, carga del sistema u otros factores. La configuración, optimización, protección y supervisión son ejemplos de áreas que se benefician de las mejoras de la computación autónoma. En los siguientes apartados se describen las áreas de configuración y optimización.

28.8.1 Configuración

DB2 proporciona soporte de ajuste automático de diversos parámetros de configuración de memoria y del sistema. Por ejemplo, los parámetros de tamaño de la memoria intermedia y del montículo de ordenación se pueden especificar como automáticos. En este caso, DB2 supervisa el sistema y aumenta o disminuye lentamente estos tamaños en función de las características de la carga de trabajo.

28.8.2 Optimización

Las estructuras de datos auxiliares (índices, MQTs) y las características de organización de datos (división, agrupación) son aspectos importantes para la mejora del rendimiento de bases de datos en DB2. En el pasado, el administrador de la base de datos (DBA) tenía que basarse en su experiencia y en pautas conocidas para elegir índices, MQTs, claves de división y claves de agrupación significativos. Dado el número potencial de opciones, ni siquiera los más expertos son capaces de encontrar la combinación exacta de estas características para una carga de trabajo dada en un tiempo limitado. La versión 8.2 de DB2 introduce un asesor de diseño (*Design Advisor*) que proporciona sugerencia para todas estas características. El asesor de diseño analiza automáticamente la carga de trabajo y usa técnicas de optimización para presentar una serie de recomendaciones. La sintaxis del comando del asistente es:

```
db2advis -d <nombre BD> -i <fichero carga de trabajo> -m MICP
```

El parámetro "-m " permite al usuario especificar las siguientes opciones:

- **M**—Tablas de consultas materializadas
- **I**—Índices
- **C**—Agrupaciones (MDC)
- **P**—Selección de claves de división

El asesor utiliza todo el poder del marco de optimización de consultas de DB2 en estas recomendaciones. Usa una carga de trabajo y restricciones de tamaño y tiempo como parámetros de entrada. Al ser la base del marco de optimización de DB2, tiene un conocimiento completo del esquema y las estadísticas de los datos subyacentes. El asesor emplea varias técnicas combinatorias para identificar índices, MQTs, MDCs y claves de división para mejorar el rendimiento de la carga de trabajo dada.

Otro aspecto de la optimización es el equilibrado de la carga de procesamiento en el sistema. En particular, las utilidades tienden a aumentar la carga en el sistema y causan la reducción significativa del rendimiento en la carga de trabajo del usuario. Dada la tendencia hacia el empleo de utilidades interactivas, hay una necesidad de equilibrar el consumo de la carga de las utilidades. La versión 8.2 de DB2 introduce un mecanismo de regulación de carga. La técnica de regulación se basa en la teoría de control de realimentación, que ajusta y regula continuamente el funcionamiento de la utilidad de copia de seguridad usando parámetros específicos de control.

28.9 Herramientas y utilidades

DB2 proporciona una serie de herramientas para facilitar su uso y administración. Se ha aumentado y mejorado el núcleo de este conjunto de herramientas mediante con un gran número de herramientas de otros fabricantes.

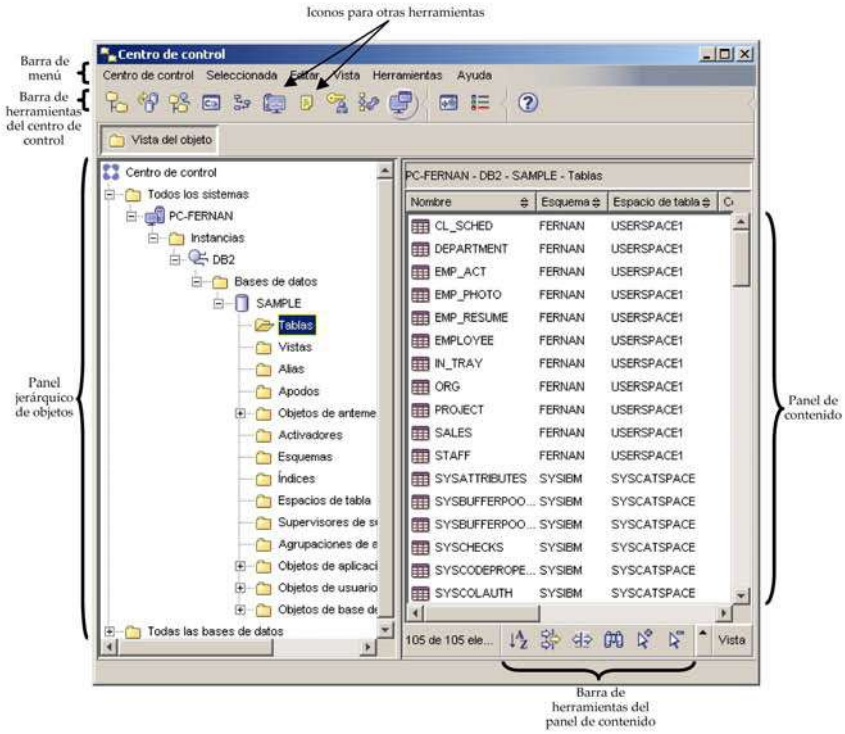


Figura 28.18 Centro de control de DB2.

El Centro de control de DB2 es la herramienta primaria para el uso y administración de bases de datos DB2. El Centro de control se ejecuta sobre muchas plataformas del tipo estación de trabajo. Está organizado a partir de objetos de datos tales como servidores, bases de datos, tablas e índices. Contiene interfaces orientadas a las tareas para ejecutar comandos y permite a los usuarios generar guiones SQL. La Figura 28.18 es una pantalla del panel principal del Centro de control. Muestra una lista de tablas en la base de datos *SAMPLE* en la instancia DB2 sobre el nodo *PC-FERNAN*. El administrador puede utilizar el menú para invocar un conjunto de herramientas componentes. Los componentes principales del Centro de control son el centro de comandos, el centro de guiones, diario, centro de licencias, centro de alertas, supervisor del rendimiento, explicación visual, administración de bases de datos remotas, gestión de almacenamiento y soporte para la réplica. El centro de comandos permite a los usuarios y administradores ejecutar comandos de la base de datos y SQL. El centro de guiones permite a los usuarios ejecutar guiones SQL construidos de forma interactiva o desde un archivo. El supervisor del rendimiento permite al usuario supervisar varios eventos en el sistema de la base de datos y obtener instantáneas del rendimiento. "SmartGuides" proporciona ayuda para la configuración de parámetros del sistema DB2. Un constructor de procedimientos almacenados ayuda al usuario a desarrollar e instalar estos procedimientos. La explicación visual proporciona al usuario vistas gráficas del plan de ejecución de la consulta. Un asistente de índices ayuda al administrador sugiriendo índices de rendimiento.

Aunque el Centro de control es una interfaz integrada de muchas de las tareas, DB2 también proporciona acceso directo a la mayoría de las herramientas. Para los usuarios las herramientas tales como el servicio de explicación, las tablas de explicación y la explicación gráfica proporcionan un análisis detallado de los planes de consulta. Los usuarios pueden usar el Centro de control para modificar las estadísticas (si tienen privilegios para ello) con el fin de generar los mejores planes de consulta.

28.9.1 Utilidades

Para los administradores, DB2 proporciona un soporte completo para la carga, importación, exportación, reorganización, redistribución y otras utilidades relacionadas con los datos. En la versión 8 la mayor parte de ellas permiten su ejecución de forma incremental e interactiva. Por ejemplo, se puede ejecutar un comando de carga en modo interactivo para permitir que las aplicaciones accedan a los contenidos

originales de una tabla de forma concurrente. Las utilidades de DB2 están completamente preparadas para ejecutarse en modo paralelo.

Además, DB2 soporta varias herramientas tales como:

- Auditoría para el mantenimiento de la traza de auditoría de las acciones sobre la base de datos.
- Regulador para controlar la prioridad y tiempos de ejecución en distintas aplicaciones.
- Supervisor de consultas para gestionar los trabajos de consulta en el sistema.
- Características de traza y diagnóstico para la depuración.
- Supervisión de eventos para seguir los recursos y eventos durante la ejecución del sistema.

DB2 para OS/390 tiene un gran conjunto de herramientas. QMF es una herramienta ampliamente usada para generar consultas ad hoc e integrarlas en aplicaciones.

28.10 Control de concurrencia y recuperación

DB2 soporta un completo conjunto de técnicas de control de concurrencia, aislamiento y recuperación.

28.10.1 Concurrencia y aislamiento

Para el aislamiento DB2 soporta los modos *lectura repetible* (Repeatable Read, RR), *estabilidad en lectura* (Read Stability, RS), *estabilidad del cursor* (Cursor Stability, CS) y *lectura no comprometida* (Uncommitted Read, UR). Los modos RR, CS y UR no necesitan mayor explicación. El modo de aislamiento RS bloquea solamente las filas que recupera una aplicación en una unidad de trabajo. En una exploración posterior la aplicación tiene garantizado ver todas estas filas (como RR) pero podría no ver nuevas filas que debería ver. Sin embargo esto podría ser un compromiso aceptable para algunas aplicaciones con respecto al aislamiento RR estricto. Normalmente el nivel de aislamiento predeterminado es CS. Las aplicaciones pueden elegir el nivel de aislamiento en la fase de enlace. La mayoría de las aplicaciones comerciales disponibles soportan los distintos niveles de aislamiento y los usuarios pueden elegir la versión correcta de la aplicación para sus requisitos.

Los distintos modos de aislamiento se implementan mediante el uso de bloqueos. DB2 soporta bloqueos en el nivel de registros y de tablas. Mantiene una estructura de datos de bloqueo de tablas separado del resto de información de bloqueo. DB2 dimensiona el bloqueo del nivel de registros al de tablas si la tabla de bloqueos se llena. DB2 implementa un bloqueo estricto de dos fases para todas las transacciones de actualización. Mantiene bloqueos de escritura y actualización hasta el momento del compromiso o retroceso. La Figura 28.19 muestra los distintos modos de bloqueo y sus descripciones. Los modos de bloqueo incluyen bloqueos intencionales en el nivel de tabla para maximizar la concurrencia. DB2 también implementa el bloqueo de clave siguiente y variaciones de este esquema para las actualizaciones que afecten a las exploraciones de índices para eliminar el problema de la lectura fantasma y de Halloween.

Una transacción puede seleccionar la granularidad de bloqueo en el nivel de tabla usando la instrucción **lock table**. Esto es útil para aplicaciones que saben que el nivel de aislamiento deseado es de nivel de tabla. Además, DB2 elige la granularidad de bloqueo apropiada para utilidades tales como reorg y load. Las versiones no interactivas de estas utilidades habitualmente bloquean la tabla en modo exclusivo, mientras que las versiones interactivas permiten a otras transacciones acceder concurrentemente realizando bloqueos de fila.

Por cada base de datos se activa un agente de detección de interbloqueos que comprueba periódicamente si se producen interbloqueos entre transacciones. El intervalo de detección de interbloqueos es un parámetro configurable. Si se produce un interbloqueo, el agente elige una víctima y aborta su ejecución con un código de error de interbloqueo.

Modo de bloqueo	Objetos	Interpretación
IN (intent none, sin intención)	Espacios de tablas, tablas	Lectura sin bloqueos de filas
IS (intent share, intentar compartir)	Espacios de tablas, tablas	Lectura con bloqueos de filas
NS (next key share, siguiente clave compartido)	Filas	Bloqueos de lectura para los niveles de aislamiento RS o CS
S (share, compartido)	Filas, tablas	Bloqueo de lectura
IX (intent exclusive, intencional exclusivo)	Espacios de tablas, tablas	Intención de actualizar filas
SIX (share with intent exclusive, compartido intencional exclusivo)	Tablas	Sin bloqueos de lectura en las filas pero con bloqueos X en las filas actualizadas
U (Update, actualización)	Filas, tablas	Bloqueo de actualización pero permitiendo leer a otros
NX (next-key exclusive, siguiente clave exclusivo)	Filas	Bloqueo de la siguiente clave para inserciones y borrados para prevenir las lecturas fantasma durante las exploraciones de índice RR
X (exclusive, exclusivo)	Filas, tablas	Sólo se permiten lectores no comprometidos
Z (superexclusive, superexclusivo)	Espacios de tablas, tablas	Acceso completo exclusivo

Figura 28.19 Modos de bloqueo de DB2.

28.10.2 Compromiso y retroceso

Las aplicaciones pueden comprometerse o retrocederse mediante el uso de las instrucciones explícitas **commit** y **rollback**. Las aplicaciones también pueden emitir instrucciones **begin transaction** y **end transaction** para controlar el ámbito de las transacciones. No se soportan las transacciones anidadas. Normalmente DB2 libera todos los bloqueos que se mantienen por una transacción en **commit** o **rollback**. Sin embargo, si se ha declarado una instrucción de cursor mediante la cláusula **with hold** entonces se mantienen algunos bloqueos durante los compromisos.

28.10.3 Registro histórico y recuperación

DB2 implementa estrictamente el registro histórico y los esquemas de recuperación ARIES. Emplea el registro histórico de escritura anticipada para enviar registros del registro histórico al archivo de registro histórico persistente antes de que las páginas de datos se escriban en el compromiso. DB2 soporta dos tipos de modos de registro: registro histórico circular y registro de archivo. En el registro histórico circular, se utiliza un conjunto predefinido de archivos de registro histórico primario y secundario. El registro histórico circular es útil para la recuperación de caídas o la recuperación de un fallo de la aplicación. En el registro histórico de archivo, DB2 crea nuevos archivos de registro histórico y debe guardar los archivos de registro histórico antiguos con el fin liberar espacio en el sistema de archivos. Los registros históricos de archivo son necesarios para la recuperación hacia adelante de una copia de seguridad de archivo. En ambos casos DB2 permite al usuario configurar el número de archivos de registro histórico y los tamaños de los archivos de los registros históricos.

En entornos con muchas actualizaciones, DB2 puede configurarse para buscar compromisos en grupo con el fin de acumular las operaciones de escritura de archivo histórico.

DB2 soporta *retroceso* de transacciones y recuperación de caídas, así como recuperaciones por instantes (point-in-time) o hacia adelante (roll-forward). En el caso de una recuperación tras una caída, DB2 ejecuta las fases de *deshacer* estándar de procesamiento y procesamiento *rehacer* hasta y desde el último punto de revisión con el fin de recuperar el estado comprometido adecuado de la base de datos. Para la recuperación por instantes, se puede restaurar la base de datos desde una copia de seguridad y avanzar a un punto específico en el tiempo usando los archivos históricos guardados. El comando de recuperación hacia adelante soporta tanto los niveles de bases de datos como de espacios de tablas. También se pueden emitir en nodos específicos sobre un sistema multinodo. Recientemente se ha implementado un esquema de recuperación en paralelo para mejorar el rendimiento en sistemas multiprocesador SMP me-

diante el uso de muchas CPUs. DB2 ejecuta la recuperación coordinada a través de nodos MPP mediante un esquema global de puntos de revisión.

28.11 Arquitectura del sistema

La Figura 28.20 muestra algunos de los distintos procesos o hebras en un servidor DB2. Las aplicaciones remotas cliente se conectan al servidor de la base de datos empleando agentes de comunicación tales como *db2tcpcm*. Se asigna un agente a cada aplicación (agente coordinador en entornos MPP o SMP) denominado hebra *db2agent*. Este agente y sus agentes subordinados ejecutan las tareas relacionadas con la aplicación. Cada base de datos tiene un conjunto de procesos o hebras que ejecutan tareas tales como preextracción, limpieza de páginas de la cola de la memoria intermedia, archivo histórico y detección de interbloqueos. Finalmente, se encuentra disponible un conjunto de agentes en el entorno del servidor para ejecutar tareas tales como detección de caídas, servicios de licencia, creación de procesos y control de recursos del sistema. DB2 proporciona parámetros de configuración para controlar el número de hebras y procesos en un servidor. Casi todos los tipos distintos de agentes se pueden controlar mediante el uso de parámetros de configuración.

La Figura 28.21 muestra los distintos tipos de segmentos de memoria en DB2. La memoria privada en los agentes o hebras se utiliza principalmente para variables locales y estructuras de datos que son relevantes solamente para la actividad actual. Por ejemplo, una ordenación privada podría asignar memoria desde el montículo privado del agente. La memoria compartida se divide en *memoria compartida del servidor*, *memoria compartida de la base de datos* y *memoria compartida de la aplicación*. El nivel de la base de datos de memoria compartida contiene estructuras de datos útiles tales como las colas de memoria intermedia, las listas de bloqueos, las cachés de los paquetes de aplicación y las áreas de ordenación compartida. Las áreas de memoria compartida del servidor y de la aplicación se usan principalmente para estructuras de datos y memorias intermedias de comunicaciones.

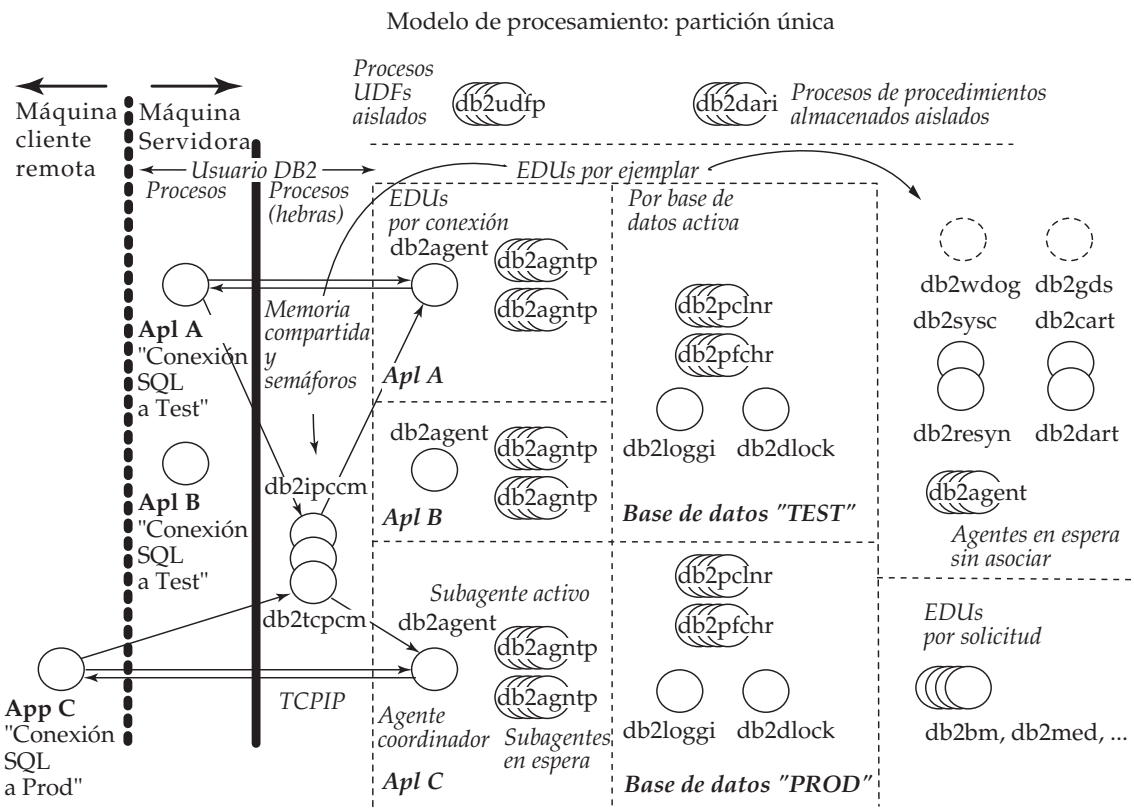


Figura 28.20 Modelo de procesos en DB2.

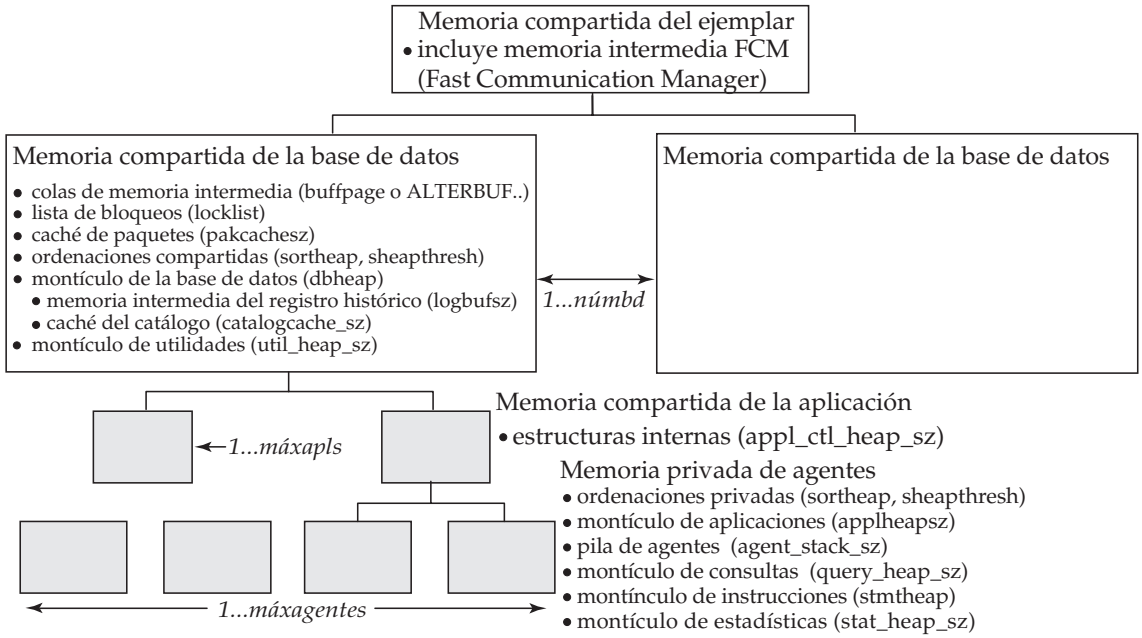


Figura 28.21 Modelo de memoria DB2.

DB2 soporta varias colas de memoria intermedia para una base de datos. Las colas de memoria intermedia se pueden crear mediante el uso de la instrucción **create bufferpool** y se puede asociar con espacios de tablas. Es útil disponer de varias colas de memoria intermedia por diversas razones, pero se deberían definir después de un cuidadoso análisis de los requisitos de la carga de trabajo. DB2 soporta una completa lista de configuración de memoria y parámetros de ajuste. Esto incluye parámetros para todas las áreas de montículos de estructuras de datos grandes tales como las colas de memoria intermedia predeterminadas, el montículo de ordenación, la caché de paquetes, los montículos de control de la aplicación y el área de lista de bloqueos.

28.12 Réplicas, distribución y datos externos

La réplica de DB2 (*DB2 Replication*) es un producto en la familia DB2 que proporciona réplica de datos entre DB2 y otros sistemas de bases de datos relacionales tales como Oracle, SQL Server de Microsoft, SQL Server de Sybase e Informix, y orígenes de datos no relacionales tales como IMS de IBM. Consiste en componentes *capturar* y *aplicar* que se controlan mediante interfaces de administración. Los mecanismos de captura de cambios se basan en tablas DB2 basadas en registros históricos o "basadas en disparadores" en el caso de otros orígenes de datos. Los cambios capturados se almacenan en áreas temporales de tablas bajo el control de DB2 Replication. Estas tablas intermedias con cambios se aplican después a las tablas destino mediante el uso de instrucciones SQL normales: inserciones, actualizaciones y borrados. Las transformaciones basadas en SQL se pueden ejecutar sobre estas tablas intermedias usando condiciones de filtro además de agregaciones. Las filas resultantes se pueden aplicar a una o más tablas destino. Los medios de administración controlan todas estas acciones.

La versión 8.2 de DB2 soporta una nueva característica denominada *réplica de colas* (*queue replication*). La réplica de colas (Q) crea un mecanismo de transporte de colas usando el producto de IBM de colas de mensajes para enviar registros de archivo históricos como mensajes. Estos mensajes se extraen de las colas en el receptor y se aplican a los destinos. El proceso de aplicación se puede paralelizar y tiene en cuenta las reglas de resolución de conflictos especificadas por el usuario.

Otro miembro de la familia DB2 es el producto integrador de información, que proporciona soporte para federación, réplica (usando el motor de réplica descrito anteriormente) y búsqueda. La edición federada integra tablas en bases de datos DB2 remotas u otras bases de datos relacionales en una única base de datos distribuida. Los usuarios y desarrolladores pueden acceder a diversas fuentes de datos

no relacionales en formato tabular usando envolturas. El motor de la edición federada proporciona un método basado en el coste para la optimización de consultas entre los distintos sitios de datos.

DB2 soporta funciones de tabla definidas por el usuario que pueden permitir el acceso de orígenes de datos no relacionales y externos. Se pueden crear funciones de tabla definidas por el usuario mediante la instrucción **create function** con la cláusula **returns table**. Con estas características DB2 puede participar en los protocolos OLE-DB.

Finalmente DB2 proporciona soporte completo para procesamiento de transacciones distribuidas mediante el protocolo de compromiso en dos fases. DB2 puede actuar como el coordinador o agente para el soporte XA distribuido. Como coordinador, DB2 puede ejecutar todos los estados del protocolo de compromiso en dos fases. Como participante, DB2 puede interactuar con cualquiera de los administradores de transacciones distribuidas comerciales.

28.13 Características de inteligencia de negocio

DB2 Data Warehouse Edition es un producto de la familia DB2 que incorpora características de inteligencia de negocio. Esta edición tiene como base el motor de DB2, y lo mejora con características para ETL, OLAP, minería y generación interactiva de informes. El motor de DB2 proporciona dimensionabilidad mediante sus características MPP. En el modo MPP, DB2 puede soportar configuraciones dimensionables a varios cientos de nodos para bases de datos de gran tamaño (terabytes). Adicionalmente, las características como MDC y MQT proporcionan soporte para los requisitos de procesamiento de consultas complejas de la inteligencia de negocio.

Otro aspecto de la inteligencia de negocio es el procesamiento analítico interactivo (On-Line Analytical Processing, OLAP). La familia DB2 incluye un componente denominado *vista de cubos* que proporciona un mecanismo para construir estructuras de datos apropiadas para MQTs dentro de DB2 que se puedan usar para procesamiento OLAP relacional. La vista de cubos porporciona soporte para el modelado de cubos multidimensionales y proporciona un mecanismo de correspondencia con un esquema relacional en estrella. Este modelo se usa para recomendar los MQTs, índices y definiciones MDC apropiados para mejorar el rendimiento de consultas OLAP sobre la base de datos. Además, las vistas de cubos pueden aprovechar el soporte nativo en DB2 para las operaciones **cube by** y **rollup** para la generación de cubos agregados. La vista de cubos es una herramienta que se puede usar para integrar estrechamente DB2 con distribuidores de productos OLAP como Business Objects, Microstrategy y Cognos.

Además, DB2 también proporciona soporte multidimensional OLAP usando el servidor OLAP de DB2. Este servidor puede crear un almacén de datos (*data mart*) desde una base de datos DB2 para realizar análisis usando técnicas OLAP. El servidor OLAP de DB2 usa el motor OLAP del producto Essbase.

DB2 Alphablox es una nueva característica que permite crear informes y realizar análisis interactivamente. Un aspecto muy atractivo de Alphablox es la posibilidad de construir rápidamente formularios de análisis basados en Web usando un enfoque basado en bloques constructivos denominados *blox*.

Para análisis avanzados, DB2 Intelligent Miner proporciona diversos componentes para modelar, puntuar (*scoring*) y visualizar datos. La minería de datos permite a los usuarios realizar clasificaciones, predicciones, agrupaciones, segmentaciones y asociaciones en grandes conjuntos de datos.

Notas bibliográficas

El origen de DB2 se remonta al proyecto del Sistema R (Chamberlin et al. [1981]). Las contribuciones de investigación de IBM incluyen áreas tales como procesamiento de transacciones (archivo histórico con escritura anticipada y algoritmos de recuperación ARIES) (Mohan et al. [1992]), procesamiento y optimización de consultas (Starburst) (Haas et al. [1990]), procesamiento paralelo (DB2 Parallel Edition) (Baru et al. [1995]), soporte para bases de datos activas (restricciones, disparadores) (Cochrane et al. [1996]), técnicas avanzadas de consultas y gestión de almacenes de datos tales como vistas materializadas (Zaharioudakis et al. [2000], Lehner et al. [2000]), agrupaciones multidimensionales (Padmanabhan et al. [2003], Bhattacharjee et al. [2003]), características autónomas (Zilio et al. [2004]) y soporte del modelo relacional orientado a objetos (ADTs, UDFs) (Carey et al. [1999]). Se pueden encontrar detalles sobre el procesamiento de consultas en multiprocesadores en Baru et al. [1995] o en las guías de administración y rendimiento de la documentación en línea de DB2 DB2 Online documentation.

A continuación se detalla una lista de material de referencia sobre DB2. Los libros de Don Chamberlin proporcionan una buena revisión de y las características de SQL y programación de DB2 (Chamberlin [1996], Chamberlin [1998]). Los primeros libros de C. J. Date y otros proporcionan un buen repaso de las características de DB2 para OS/390 (Date [1989], Martin et al. [1989]). Los manuales de DB2 proporcionan revisiones definitivas de cada versión particular de DB2. La mayoría de estos manuales están disponibles en línea (Documentación en línea de DB2 DB2 Online documentation). Libros recientes como *DB2 for Dummies* (Zikopoulos et al. [2000]), *DB2 SQL developer's guide* (Sanders [2000]) y *DB2 Administration Certification Guides* (Cook et al. [1999]) proporcionan formación práctica para usar y administrar DB2. Finalmente, Prentice Hall está publicando una serie de completa de libros sobre el enriquecimiento y certificación sobre varios aspectos de DB2.

Chamberlin [1998], Zikopoulos et al. [2004] y la biblioteca de documentación de DB2 proporcionan una descripción completa del soporte de SQL.

CAPÍTULO 29

SQL Server de Microsoft

Sameet Agarwal, José A. Blakeley, Thierry D'Hers, Gerald Hinson, Dirk Myers, Vaqar Pirzada, Bill Ramos, Balaji Rathakrishnan, Michael Rys, Florian Waas, Michael Zwilling
Microsoft

SQL Server de Microsoft es un sistema de gestión de bases de datos relacionales que se puede instalar tanto en computadoras portátiles y de sobremesa como en servidores corporativos y en dispositivos de bolsillo, como los PocketPC y los lectores de códigos de barras, con una versión basada en el sistema operativo PocketPC. SQL Server se desarrolló originalmente en los años ochenta del siglo veinte en SyBase para sistemas UNIX y posteriormente Microsoft lo tradujo a sistemas Windows NT. Desde 1994 Microsoft ha distribuido versiones de SQL Server desarrolladas independientemente de Sybase, que dejó de usar el nombre SQL Server a finales de los años noventa. La última versión, SQL Server 2005, está disponible en las ediciones exprés (Express), estándar (Standard) y corporativa (Enterprise), y se ha traducido a muchos idiomas de todo el mundo. En este capítulo el término SQL Server se refiere a todas estas ediciones de SQL Server 2005.

SQL Server proporciona servicios de duplicación (réplica) entre varias copias de SQL Server y con otros sistemas de bases de datos. Su componente Analysis Services (servicios de análisis), una parte esencial del sistema, incluye servicios de procesamiento analítico en línea (OLAP, Online Analytical Processing) y de minería de datos. SQL Server proporciona una gran colección de herramientas gráficas y de "asistentes" que guían a los administradores de las bases de datos en tareas como la configuración de copias de seguridad periódicas, la duplicación de datos entre los distintos servidores y el ajuste del rendimiento de las bases de datos. Muchos entornos de desarrollo soportan SQL Server, incluidos Visual Studio de Microsoft y los productos relacionados, en especial los productos y servicios .NET.

29.1 Herramientas para la administración, el diseño y la consulta de las bases de datos

SQL Server proporciona un conjunto de herramientas para gestionar todos los aspectos del desarrollo, la consulta, el ajuste, la prueba y la administración de SQL Server. La mayor parte de estas herramientas giran alrededor de Management Studio (conocido formalmente como Enterprise Manager) de SQL Server. Management Studio ofrece una interfaz de comandos común para la administración de todos los servicios asociados con SQL Server, que incluye el motor de la base de datos (Database Engine), los servicios de análisis (Analysis Services), los servicios de informes (Reporting Services), el servidor móvil de SQL Server (SQL Server Mobile) y los servicios de integración (Integration Services).

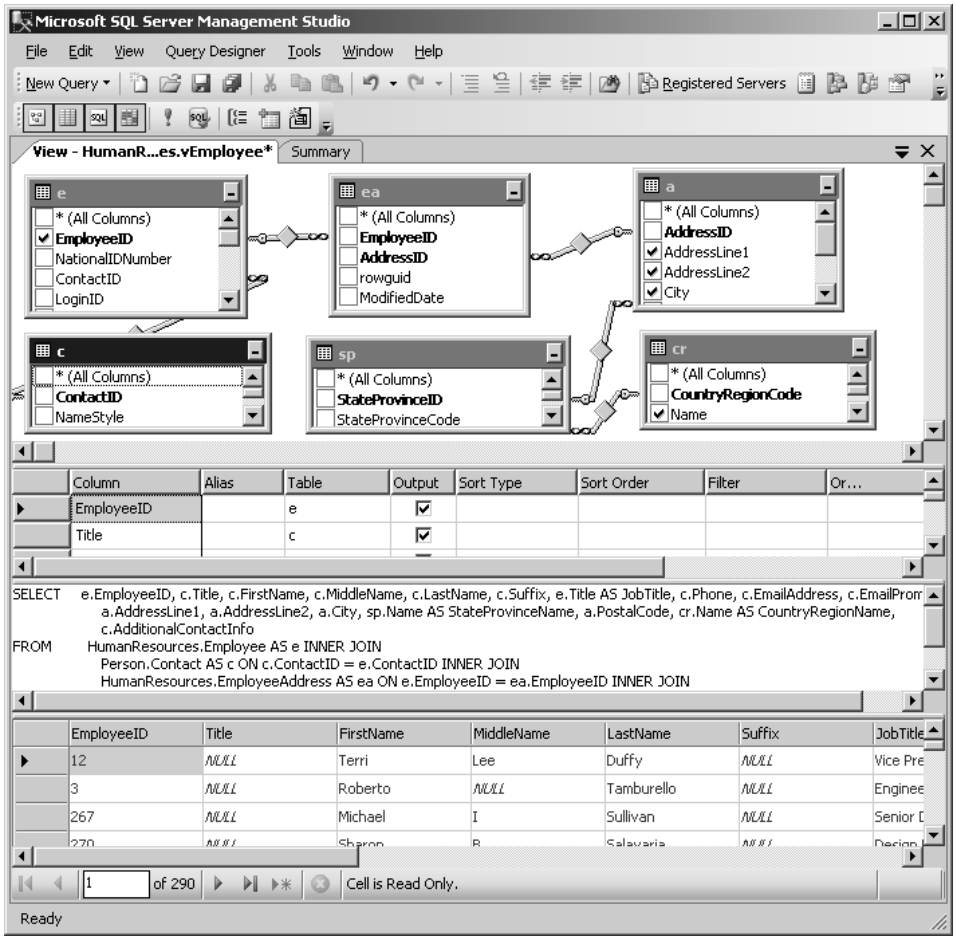


Figura 29.1 View Designer abierto para la vista HumanResources.vEmployee.

29.1.1 Desarrollo de bases de datos y herramientas visuales para las bases de datos

Al diseñar una base de datos, el administrador de la base de datos crea objetos de la base de datos como las tablas, las columnas, las claves, los índices, las relaciones, las restricciones y las vistas. Management Studio de SQL Server proporciona acceso a herramientas visuales para la creación de estos objetos en las bases de datos. Estas herramientas proporcionan tres mecanismos de ayuda al diseño de las bases de datos: Database Designer (diseñador de bases de datos), Table Designer (diseñador de tablas) y View Designer (diseñador de vistas).

Database Designer es una herramienta visual que permite al propietario de la base de datos y a sus delegados crear tablas, columnas, claves, índices, relaciones y restricciones. En esta herramienta el usuario puede interactuar con los objetos de la base de datos mediante los diagramas de la base de datos, que muestran de forma gráfica la estructura de la base de datos. View Designer proporciona una herramienta visual de consulta que permite al usuario crear y modificar vistas de SQL mediante el uso de las posibilidades de arrastrar y soltar de Windows. La Figura 29.1 muestra una vista abierta desde Management Studio.

29.1.2 Herramientas para la consulta y el ajuste de las bases de datos

Management Studio de SQL Server proporciona varias herramientas de ayuda al proceso de desarrollo de las aplicaciones. Se pueden desarrollar y probar inicialmente las consultas y los procedimientos almacenados mediante Query Editor (editor de consultas) integrado que sustituye al Analizador de

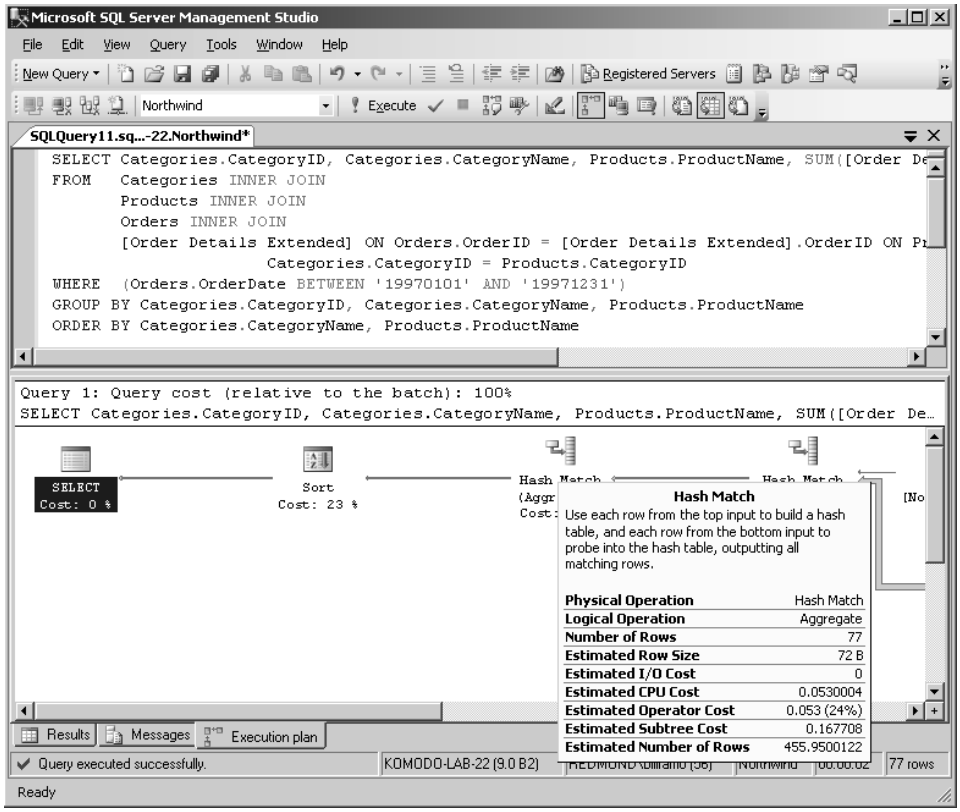


Figura 29.2 Plan de ejecución de una reunión de cuatro tablas con la agregación **group by**.

consultas de SQL Server. Query Editor soporta la creación y edición de guiones para T-SQL, SQLCMD, MDX, DMX, XMLA y SQL Server Mobile. Se pueden realizar más análisis usando ServerProfiler de SQL. Las recomendaciones de ajuste de las bases de datos las proporciona una tercera herramienta: Database Tuning Advisor.

29.1.2.1 Query Editor

Query Editor proporciona una interfaz de usuario gráfica sencilla para la ejecución de consultas de SQL y el examen de sus resultados. Query Editor proporciona también una representación gráfica de **showplan**, los pasos elegidos por el optimizador para la ejecución de cada consulta. Query Editor está integrado con Object Explorer de Management Studio, que permite que el usuario arrastre objetos o tablas a las ventanas de las consultas y ayuda a crear instrucciones **select**, **insert**, **update** y **delete** para cualquier tabla.

El administrador o desarrollador de la base de datos puede usar Query Editor para:

- Analizar consultas: Query Editor puede mostrar el plan de ejecución en formato gráfico o de texto de cualquier consulta, así como mostrar las estadísticas relativas al tiempo y a los recursos necesarios para su ejecución.
- Dar formato a las consultas de SQL: incluido el sangrado y la codificación sintáctica por colores.
- Usar plantillas para los procedimientos almacenados, las funciones y las instrucciones básicas de SQL: Management Studio trae docenas de plantillas predefinidas para la creación de instrucciones de LDD, y los usuarios también pueden definir las suyas propias.

La Figura 29.2 muestra Management Studio y Query Editor con el plan gráfico de ejecución de una consulta que supone una reunión de cuatro tablas y una agregación.

29.1.2.2 SQL Profiler

SQL Profiler es una utilidad gráfica que permite a los administradores de bases de datos supervisar y registrar la actividad de Database Engine y de Analysis Services de SQL Server. Puede mostrar toda la actividad del servidor en tiempo real o crear filtros que se centren en las acciones de usuarios, aplicaciones o tipos de comandos concretos. También permite mostrar cualquier instrucción o procedimiento almacenado de SQL enviado a cualquier ejemplar de SQL Server (si los privilegios de seguridad lo permiten), así como los datos de rendimiento que indican el tiempo que la consulta ha tardado en ejecutarse, la cantidad de CPU y de E/S necesarias y el plan de ejecución usado.

SQL Profiler permite ahondar aún más en SQL Server para supervisar automáticamente cada instrucción ejecutada como parte de un procedimiento almacenado, cada operación de modificación de los datos, cada bloqueo adquirido o liberado, o cada ocasión en que crece un archivo de la base de datos. Se pueden capturar docenas de eventos distintos y, para cada evento, se pueden capturar docenas de elementos de datos. SQL Server divide realmente la funcionalidad de seguimiento en dos componentes diferentes, aunque conectados. SQL Profiler es el servicio de traza en los clientes. Mediante SQL Profiler los usuarios pueden decidir guardar los datos capturados en un archivo o en una tabla, además de mostrarlos en la interfaz de usuario (UI) de Profiler. Esta herramienta muestra todos los eventos que cumplen el criterio del filtro en el momento en que se producen. Una vez que se han guardado los datos de la traza, SQL Profiler puede leer los datos guardados para mostrarlos o analizarlos.

En el lado del servidor está el servicio de traza de SQL, que gestiona las colas de los eventos generados por los productores de eventos. Una hebra (subproceso en DB2) consumidora lee los eventos desde las colas y los filtra antes de enviarlos al proceso que los ha solicitado. Los eventos son la unidad principal de actividad en lo que se refiere a la traza, y un evento puede ser cualquier cosa que suceda dentro de SQL Server o entre SQL Server y un cliente. Por ejemplo, la creación o eliminación de un objeto, la ejecución de un procedimiento almacenado, la adquisición o liberación de un bloqueo y el envío de un archivo de procesamiento por lotes de Transact-SQL desde un cliente a SQL Server son eventos. Hay un conjunto de procedimientos almacenados del sistema para definir los eventos que hay que seguir, los datos de cada evento que son interesantes y el lugar en el que se debe guardar la información recogida de los eventos. Los filtros aplicados a los eventos pueden reducir la cantidad de información recogida y almacenada.

SQL Server garantiza que se recoja siempre cierta información crítica, y se puede usar como un útil mecanismo de auditoría. SQL Server está certificado para el nivel C2 de seguridad y muchos de los eventos de los que se puede realizar una traza sólo están disponibles para cumplir los requisitos de la certificación C2.

29.1.2.3 Database Tuning Advisor

Las consultas y las actualizaciones se pueden ejecutar mucho más rápido si se dispone de un conjunto de índices adecuado. El diseño de los mejores índices posibles para las tablas de una base de datos de gran tamaño es una tarea compleja: no sólo exige un conocimiento completo del modo en que SQL Server usa los índices y de la manera en que el optimizador de consultas toma las decisiones, sino del modo en que las aplicaciones y las consultas interactivas usan realmente los datos. Database Tuning Advisor (DTA) de SQL Server (que sustituye al Asistente para optimización de índices de SQL Server 2000) es una potente herramienta para el diseño de los mejores índices y de las mejores vistas indexadas (materializadas) posibles, de acuerdo con las cargas de trabajo observadas de consultas y de actualizaciones.

DTA puede ajustar varias bases de datos a la vez y basa sus recomendaciones en una carga de trabajo que puede ser un archivo de eventos de traza capturados, un archivo con instrucciones de SQL o un archivo de datos de XML. SQL Profiler está diseñado para capturar todas las instrucciones de SQL remitidas por todos los usuarios en un periodo de tiempo determinado. DTA puede examinar luego los patrones de acceso a los datos de todos los usuarios, las aplicaciones y las tablas, y realizar recomendaciones equilibradas.

29.1.3 Management Studio de SQL Server

Además de proporcionar acceso a las herramientas de diseño de bases de datos y a las herramientas visuales para bases de datos, Management Studio de SQL Server, fácil de usar, soporta la administración

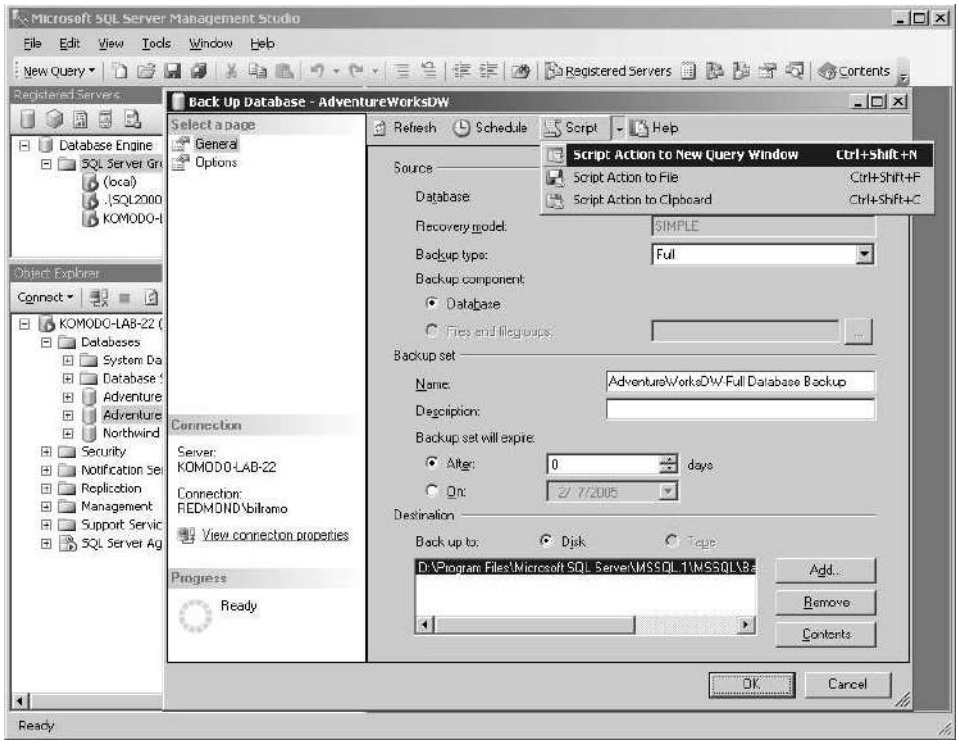


Figura 29.3 La interfaz de Management Studio de SQL Server.

centralizada de todos los aspectos de varias instalaciones de Database Engine, Analysis Services, Reporting Services, Integration Services y SQL Server Mobile, incluidos la seguridad, los eventos, las alertas, la programación, las copias de seguridad, la configuración del servidor, los ajustes, la búsqueda de texto completo y la duplicación. Management Studio de SQL Server permite que el administrador de la base de datos cree, modifique y copie los esquemas y los objetos de las base de datos de SQL Server como las tablas, las vistas y los desencadenadores. Debido a que se puede organizar en grupos varias instalaciones de SQL Server y tratarlas como una unidad, Management Studio de SQL Server puede gestionar cientos de servidores de manera simultánea.

Aunque se puede ejecutar en la misma computadora que el motor de SQL Server, Management Studio de SQL Server ofrece las mismas posibilidades de gestión cuando se ejecuta en cualquier máquina de Windows 2000 (o posterior). Además, la arquitectura eficiente cliente−servidor de SQL Server hace práctico el uso de las posibilidades de acceso remoto (acceso telefónico a redes) de Windows para la administración y la gestión.

Management Studio de SQL Server libera al administrador de la base de datos de tener que conocer los pasos y la sintaxis concretos necesarios para completar cada trabajo. Ofrece asistentes que guían a los administradores de bases de datos en el proceso de configuración y mantenimiento de las instalaciones de SQL Server. La interfaz de Management Studio se muestra en la Figura 29.3 e ilustra la manera en que se puede crear de manera directa un guión para las copias de seguridad de las bases de datos a partir de los cuadros de diálogo.

29.2 Variaciones y extensiones de SQL

SQL Server permite a los desarrolladores de aplicaciones escribir la lógica corporativa del lado del servidor usando Transact-SQL o algún lenguaje de programación .NET, como C#, Visual Basic, COBOL o J++. Transact-SQL es un lenguaje de programación de bases de datos completo que incluye instrucciones para la definición y la manipulación de los datos, instrucciones iterativas y condicionales, variables, procedimientos y funciones. Transact-SQL soporta la mayor parte de las instrucciones y estructuras de consulta y de modificación de datos del LDD **obligatorias** de la norma SQL-2003. Véase el Apartado 29.2.1 para

conocer la lista de tipos de datos de SQL-2003 soportados. Además de las características obligatorias, Transact-SQL también soporta muchas características **opcionales** de la norma de SQL-2003, como las consultas recursivas, las expresiones comunes de las tablas, las funciones definidas por los usuarios y los operadores relacionales como **intersect** y **except**, entre otros.

29.2.1 Tipos de datos

SQL Server soporta todos los tipos de datos escalares obligatorios de la norma de SQL-2003, excepto el de fecha (date) y el de hora (time). Soporta el tipo de datos de marca de tiempo (timestamp, también denominado datetime), que permite guardar los componentes de fecha y de hora. SQL Server también soporta la posibilidad de dar otros nombres a los tipos del sistema usando nombres facilitados por los usuarios; el uso de alias es parecido en funcionalidad a los diferentes tipos de SQL-2003, pero no cumple completamente con ellos.

Entre los tipos primitivos exclusivos de SQL Server se encuentran:

- Los tipos de cadenas de caracteres y binarias de gran tamaño de tamaño variable hasta los $2^{31} - 1$ bytes (**text/ntext/image**, **varchar/nvarchar/varbinary(max)**). Los tipos de datos text/ntext/image exigen el uso de un textptr especializado que actúe de manejador o un puntero a los valores LOB. Los tipos de datos varchar/nvarchar/varibinary(max) tienen la misma capacidad de bytes que text/ntext/image, pero el modelo de programación es parecido al de los tipos small character y byte string.

- El tipo XML, descrito en el Apartado 29.11, se usa para guardar datos de XML en las columnas de las tablas. El tipo XML puede tener, opcionalmente, una *colección de esquemas* de XML asociado que especifique la restricción de que los ejemplares de este tipo deben adherirse a uno de los tipos de XML definidos en la colección de esquemas.

- sql_variant es un tipo de datos escalares que puede contener valores de cualquier tipo escalar de SQL (excepto los tipos large character, large binary y sql_variant). Este tipo lo usan las aplicaciones que necesitan guardar datos cuyo tipo no se puede anticipar en el momento de definición de los datos. sql_variant también es el tipo de las columnas formadas a partir de la ejecución del operador relacional **unpivot** (véase el Apartado 29.2.2). Internamente, el sistema realiza un seguimiento del tipo original de los datos. Es posible filtar, reunir y ordenar las columnas sql_variant. La función del sistema **sql_variant_property** devuelve los detalles de los datos reales guardados en las columnas de tipo sql_variant, incluida la información sobre el tipo básico y sobre el tamaño.

Además, SQL Server soporta los tipos el tipo table y el tipo cursor, que no se pueden usar como columnas de las tablas, pero se pueden usar como variables del lenguaje Transact-SQL:

- El tipo **table** permite que una variable guarde un conjunto de filas. Los ejemplares de este tipo se usan, sobre todo, para guardar los resultados temporales de los procedimientos almacenados o como valor devuelto por funciones cuyo resultado es una tabla. Las variables table se comportan como variables locales. Tienen un ámbito bien definido, que es la función, el procedimiento almacenado o el procesamiento por lotes en que se declaran. Dentro de su ámbito las variables table se pueden utilizar como las tablas normales. Se pueden aplicar en cualquier lugar en que se utilicen tablas o expresiones de tabla en las instrucciones **select**, **insert**, **update** o **delete**.

- El tipo **cursor** permite las referencias a objetos cursor. El tipo cursor puede usarse para declarar variables, o argumentos de entrada/salida de las rutinas, para hacer referencia a cursores de unas llamadas a rutinas a otras.

29.2.2 Mejoras del lenguaje de consultas

Además de los operadores relacionales de SQL como la **reunión interna** y la **reunión externa**, SQL Server soporta los operadores relacionales **pivot**, **unpivot** y **apply**.

- **pivot** es un operador que transforma su conjunto de resultados de entrada con dos columnas, que representan pares nombre-valor, en varias columnas, una por cada nombre de la entrada. La columna de nombres de la entrada se denomina columna pivote. El usuario debe indicar los nombres que hay que trasponer de la entrada a las diferentes columnas de la salida. Considérese la tabla *VentasMensuales* (*IDProducto, Mes, CantidadVentas*). La consulta siguiente, que usa el operador **pivot**, devuelve la *CantidadVentas* de los meses de enero, febrero y marzo como columnas diferentes. Téngase en cuenta que el operador pivot también lleva a cabo una agregación implícita de todas las otras columnas de la tabla y una agregación explícita de la columna pivote.

  select *
  from *VentasMensuales* **pivot**(**sum**(*CantidadVentas*) **for** *mes* **in** ('Ene', 'Feb', 'Mar')) *T*

  La operación inversa a **pivot** es **unpivot**.

- El operador **apply** es un operador binario que toma dos entradas valoradas como tablas, de las que la parte derecha suele ser la llamada a una función que devuelve una tabla que toma como argumentos una o varias columnas de la parte izquierda. Las columnas generadas por este operador son la unión de las columnas de sus dos entradas. El operador **apply** se puede usar para valorar su entrada derecha para cada fila de su entrada izquierda y llevar a cabo una **union all** de las filas a lo largo de todas estas evaluaciones. Hay dos variedades del operador **apply** parecidas a las de **join**, es decir, **cross** y **outer**. Las dos variedades se diferencian en lo relativo al manejo el caso de que la entrada derecha produzca un conjunto de resultados vacío. En el caso de **cross apply**, hace que la fila correspondiente de la entrada izquierda no aparezca en el resultado. En el caso de **outer apply**, la fila de la entrada izquierda aparece con valores NULL para las columnas de la entrad derecha. Considérese una función que se valora como tabla denominada *HallarInformes*, que toma como entrada el ID de un empleado dado y devuelve el conjunto de empleados de la organización que informa directa o indirectamente a ese empleado. La consulta siguiente llama a esta función para el Jefe de cada departamento desde la tabla Departamentos:

  select *
  from *Departmentos D* **cross apply** *HallarInformes*(*D.IDJefe*)

29.2.3 Rutinas

Los usuarios pueden escribir rutinas que se ejecuten dentro del proceso servidor como funciones escalares o tabulares, como procedimientos almacenados o como disparadores usando Transact-SQL o algún lenguaje .NET. Todas estas rutinas se definen para la base de datos mediante la instrucción **create [function, procedure, trigger]** del LDD. Las funciones escalares se pueden usar en cualquier expresión escalar de las instrucciones LMD o LDD de SQL. Las funciones que devuelven tablas se pueden usar en cualquier parte en que se permitan tablas en las instrucciones **select**. Las funciones que devuelven tablas de Transact-SQL cuyo cuerpo contiene una sola instrucción **select** de SQL se tratan como vistas (expandidas en línea) en la consulta que hace referencia a la función. Dado que las funciones que devuelven tablas permiten argumentos de entrada, las funciones en línea valoradas como tablas se pueden considerar vistas parametrizadas.

29.2.3.1 Vistas indexadas

Además de las vistas tradicionales definidas en la norma ANSI de SQL, SQL Server soporta las vistas indexadas (materializadas). Las vistas indexadas pueden mejorar sustancialmente el rendimiento de las consultas complejas de ayuda a la toma de decisiones que recuperan gran número de filas y agregan grandes cantidades de información en sumas, recuentos y medias. SQL Server soporta la creación de

índices agrupados en cada vista y, en consecuencia, cualquier número de índices no agrupados. Una vez indexada una vista, el optimizador puede usar sus índices en consultas que hagan referencia a la vista o a sus tablas base. Las consultas ya existentes se pueden beneficiar de la eficiencia mejorada de recuperar los datos directamente de la vista indexada sin que haga falta reescribirlas para que hagan referencia a la vista. Las instrucciones de actualización de las tablas base de la vista se propagan automáticamente a las vistas indexadas.

29.2.3.2 Vistas actualizables y disparadores

Generalmente, las vistas puede ser objetivo de las instrucciones **update**, **delete** o **insert** si la modificación de los datos sólo se aplica a una de las tablas base de la vista. Las actualizaciones de las vistas divididas se pueden propagar a varias tablas base. Por ejemplo, la siguiente instrucción **update** incrementa los precios para el editor "0736" en un diez por ciento.

```
update vistatítulos
set precio = precio * 1.10
where id_editorial = '0736'
```

Para las modificaciones de los datos que afectan a más de una tabla base, la vista se puede actualizar si hay algún disparador **instead** definido para la operación: los disparadores **instead** para las operaciones **insert**, **update** o **delete** se pueden definir sobre una vista para especificar las actualizaciones que hay que ejecutar en las tablas base para implementar las modificaciones de la vista correspondientes.

Los disparadores son procedimientos de Transact-SQL o de .NET que se ejecutan automáticamente cuando se envía una instrucción de LMD (**update**, **insert** o **delete**) a una tabla base o a una vista. Los disparadores son mecanismos que posibilitan la aplicación de la lógica corporativa de forma automática al modificar los datos o al ejecutar instrucciones de LDD. Los disparadores pueden extender la lógica de comprobación de la integridad de las restricciones declarativas, de las predeterminadas y de las reglas, aunque las restricciones declarativas se deben usar preferentemente siempre que sean suficientes.

Los disparadores se pueden clasificar en LMD y en LDD, según el tipo de evento que los desencadene. Los disparadores LMD se definen contra tablas o vistas que se están modificando. Los disparadores LDD se definen contra bases de datos completas para una o varias instrucciones de LDD, como **create table**, **drop procedure**, etc.

Los disparadores se pueden clasificar en disparadores **after** e **instead**, según el momento en que se los invoca en relación con la acción que los desencadena. Los desencadenadores **after** se ejecutan después de la instrucción de disparo y luego se aplican restricciones declarativas. Los disparadores **instead** se ejecutan en lugar de la acción que los dispara. Se puede considerar que los disparadores **instead** son parecidos a los disparadores **before**, pero sustituyen realmente a la acción de disparo. En SQL Server, los disparadores **after** de LMD sólo se pueden definir sobre las tablas base, mientras que los disparadores **instead** de LMD se pueden definir sobre las tablas base o sobre las vistas. Los disparadores **instead** permiten hacer actualizable prácticamente cualquier vista. Los disparadores **instead** LDD se pueden definir sobre cualquier instrucción del LDD.

29.3 Almacenamiento e índices

En SQL Server cada base de datos hace referencia a un conjunto de archivos que contienen datos y están soportados por un único registro histórico de transacciones. La base de datos es la unidad principal de administración de SQL Server, y también proporciona un contenedor para estructuras físicas como las tablas y los índices y para estructuras lógicas como las restricciones, las vistas, etc.

29.3.1 Grupos de archivos

Con el fin de gestionar el espacio de la base de datos de forma efectiva, el conjunto de archivos de datos de la base de datos se divide en grupos denominados grupos de archivos. Cada grupo de archivos contiene uno o más archivos del sistema operativo.

Cada base de datos tiene, al menos, un grupo de archivos conocido como grupo de archivos principal. Este grupo de archivos contiene todos los metadatos de la base de datos en tablas del sistema. El grupo de archivos principal también puede contener datos de usuario.

Si se crean grupos de archivos definidos por el usuario adicionales, los usuarios pueden controlar de forma explícita la ubicación de cada tabla, de cada índice o de cada columna de objetos de gran tamaño de las tablas colocándolas en un grupo de archivos. Por ejemplo, el usuario puede decidir guardar una tabla en el grupo de archivos A, su índice no agrupado en el grupo de archivos B y las columnas de objetos de gran tamaño de la tabla en el grupo de archivos C. La ubicación de estas tablas y de estos índices en grupos de archivos diferentes permite al usuario controlar el uso de los recursos de hardware (esto es, de los discos y del subsistema de E/S). Siempre se considera a un grupo de archivos determinado el grupo de archivos predeterminado; inicialmente, el grupo de archivos predeterminado es el grupo de archivos principal, pero se puede otorgar la *propiedad predeterminado* a cualquier grupo de archivos definido por los usuarios. Si una tabla o índice no se ubica específicamente en un grupo de archivos, se crea en el grupo de archivos predeterminado.

29.3.2 Gestión del espacio en los grupos de archivos

Uno de los principales propósitos de los grupos de archivos es permitir una gestión efectiva del espacio. Todos los archivos de datos se dividen en unidades de tamaño fijo de ocho kilobytes denominadas **páginas**. El sistema de asignación es responsable de asignar esas páginas a las tablas y a los índices. El objetivo del sistema de asignación es minimizar la cantidad de espacio desperdiciado y, al tiempo, mantener el grado de fragmentación de la base de datos al mínimo para garantizar un buen rendimiento de exploración. Con el fin de lograr este objetivo, el administrador de asignación suele asignar y extraer todas las páginas en grupos de ocho páginas contiguas denominadas **extensiones**.

El sistema de asignación gestiona estas extensiones mediante varios mapas de bits. Estos mapas de bits permiten al sistema de asignación encontrar una página o una extensión para asignarla de forma rápida. Estos mapas de bits también se usan cuando se ejecuta una exploración de tabla completa o de índices. La ventaja de usar mapas de bits basados en la asignación para la exploración es que permite recorrer en el orden del disco todas las extensiones que pertenecen al nivel de hojas de la tabla o del índice, lo que mejora significativamente el rendimiento de la exploración.

Si hay más de un archivo en un grupo de archivos, el sistema de asignación asigna extensiones para cualquier objeto de ese grupo de archivos mediante un algoritmo de "relleno proporcional". Cada archivo se rellena en proporción a la cantidad de espacio libre de ese archivo en comparación con los demás archivos. Esto rellena todos los archivos del grupo de archivos aproximadamente al mismo ritmo, y permite al sistema usar por igual todos los archivos del grupo de archivos.

Una de las decisiones más importantes al configurar una base de datos es la determinación del tamaño que se desea que tenga. SQL Server permite que los archivos de datos cambien de tamaño después de la creación de la base de datos. El usuario puede, incluso, decidir que el archivo de datos crezca automáticamente si la base de datos se está quedando sin espacio. Por ello, el usuario puede configurar la base de datos con una aproximación razonable de su tamaño esperado y dejar que los archivos de la base de datos crezcan y se ajusten al patrón de uso, si la aproximación inicial es errónea. SQL Server permite que los archivos disminuyan de tamaño. Para disminuir el tamaño de un archivo de datos SQL Server traslada todos los datos desde el extremo físico del archivo a un punto más cercano al inicio del archivo y luego reduce realmente su tamaño, devolviendo el espacio liberado al sistema operativo.

29.3.3 Tablas

SQL Server soporta las organizaciones de las tablas en montones (o montículos—*heap*) y en agrupaciones (*clusters*). En las tablas organizadas en montones la ubicación de cada fila de la tabla la determina completamente el sistema y el usuario no la especifica en absoluto. Las filas de los montones tienen un identificador fijo conocido como identificador de la fila (RID, Row Identifier), y su valor no cambia nunca, a no ser que se reduzca el tamaño del archivo y la fila se traslade. Si la fila se hace tan grande que no cabe en la página en la que se insertó originalmente, el registro se traslada a un lugar distinto,

pero se deja un resguardo de entrega en la ubicación original, de modo que el registro todavía se pueda encontrar usando su RID original.

En la organización de índices agrupados de las tablas, las filas de la tabla se guardan en un árbol B^+ ordenado por la clave de agrupamiento del índice. La clave del índice agrupado también sirve como identificador único de cada fila. La clave del índice agrupado se puede definir como no única, en cuyo caso SQL Server agrega una columna oculta adicional para hacer que la clave sea única. El índice agrupado también sirve como estructura de búsqueda para identificar una fila de la tabla con una clave concreta o explorar un conjunto de filas de la tabla con las claves ubicadas dentro de un cierto rango. Los índices agrupados son el tipo más frecuente de organización de las tablas.

29.3.4 Índices

SQL Server también soporta los índices de árbol B^+ secundarios (no agrupados). Las consultas que sólo hacen referencia a las columnas que están disponibles mediante índices secundarios se procesan mediante la recuperación de las páginas desde el nivel hoja de los índices sin necesidad de recuperar los datos del índice agrupado o montón. Los índices no agrupados de las tablas con índices agrupados contienen las columnas clave del índice agrupado. Por tanto, las filas del índice agrupado se pueden trasladar a una página diferente (mediante divisiones, desfragmentaciones o recreaciones del índice) sin necesidad de modificaciones en los índices no agrupados.

SQL Server soporta la adición de columnas calculadas a las tablas. Las columnas calculadas son columnas cuyo valor es una expresión, normalmente basada en el valor de otras columnas de esa fila. SQL Server permite que el usuario construya índices secundarios en términos de las columnas calculadas.

29.3.5 Particiones

SQL Server soporta la división de las tablas y de los índices no agrupados. Los índices divididos están constituidos por varios árboles B^+, uno por partición. Las tablas divididas sin índices (montones) están constituidas por varios montones, uno por partición. Por brevedad, a partir de aquí sólo se hará referencia a los índices divididos (agrupados o sin agrupar) y se ignorarán los montones.

La división de índices de gran tamaño permite al administrador mayor flexibilidad en la gestión del almacenamiento del índice y puede mejorar el rendimiento de algunas consultas, ya que las particiones actúan como índices de grano grueso.

La división de los índices se especifica proporcionando una función y un esquema de partición. La función de partición asigna el dominio de la columna de partición (cualquier columna del índice) a las particiones numeradas de 1 a N. El esquema de la partición asigna los números de las particiones generadas por la función de partición a grupos de archivos concretos en los que se guardan las particiones.

29.3.6 Creación en línea de índices

La creación de índices nuevos y la reconstrucción de los ya existentes en una tabla se puede llevar a cabo en línea, es decir, mientras se están llevando a cabo las operaciones de selección, inserción, borrado o actualización en esa tabla. La creación del nuevo índice tiene lugar en tres fases. La primera fase no es más que la creación de un árbol B^+ vacío para el nuevo índice con el catálogo que muestre que el nuevo índice está disponible para operaciones de mantenimiento. Esto es, todas las operaciones posteriores de inserción, borrado o actualización deben mantener el nuevo índice, pero éste no se halla disponible para las consultas. La segunda fase consiste en la exploración de la tabla para recuperar las columnas del índice de cada fila, ordenar las filas e insertarlas en el nuevo árbol B^+. Estas inserciones deben tener cuidado al interactuar con las otras filas del nuevo árbol B^+ colocadas allí por las operaciones de mantenimiento del índice debidas a las actualizaciones de la tabla base. La exploración es de instantáneas que, sin bloqueos, garantiza que la exploración vea toda la tabla únicamente con los resultados de las transacciones comprometidas en el momento de comienzo de la exploración. Esto se consigue usando la tecnología de aislamiento de instantáneas descrita en el Apartado 29.5.1. La fase final de la creación del índice supone la actualización del catálogo para que indique que la creación del índice se ha completado y que éste se halla disponible para las consultas.

29.3.7 Exploraciones y lecturas anticipadas

La ejecución de las consultas en SQL Server puede involucrar varios modos de exploración diferentes de las tablas y de los índices subyacentes. Entre estos modos de exploración están las exploraciones ordenadas y las desordenadas, las exploraciones en serie y las paralelas, las unidireccionales y las bidireccionales, las exploraciones hacia delante y hacia atrás y la exploración de toda la tabla o de todo el índice y las exploraciones de rango o filtradas.

Cada uno de estos modos de exploración tiene un mecanismo de lectura anticipada que intenta que la exploración se anticipe a las necesidades de ejecución de la consulta, con el fin de reducir las sobrecargas de búsqueda y de latencia y de usar el tiempo de disco no ocupado. El algoritmo de lectura anticipada de SQL Server usa el conocimiento del plan de ejecución de la consulta con el fin de conducir la lectura anticipada y asegurarse de que solamente se lean los datos que la consulta necesita realmente. Además, la cantidad de lectura anticipada se dimensiona de forma automática según el tamaño de la memoria intermedia agrupada, el volumen de E/S que el subsistema del disco puede sostener y la velocidad a la que la ejecución de la consulta consume los datos.

29.4 Procesamiento y optimización de consultas

El procesador de consultas de SQL Server está basado en un entorno extensible que permite la rápida incorporación de nuevas técnicas de ejecución y de optimización. Cualquier consulta de SQL se puede expresar en forma de árbol de operadores del álgebra relacional extendida de SQL Server. Mediante la abstracción de los operadores de este álgebra en *iteradores*, la ejecución de la consulta encapsula los algoritmos de procesamiento de datos como unidades lógicas que se comunican entre sí usando la interfaz GetNextRow(). A partir del árbol inicial de la consulta, el optimizador de consultas de SQL Server genera alternativas usando transformaciones en árbol y estima su coste de ejecución teniendo en cuenta el comportamiento de los iteradores y los modelos estadísticos para realizar selecciones.

29.4.1 Visión general del proceso de optimización

Las consultas complejas presentan oportunidades significativas de optimización que exigen la ordenación de los operadores de unos bloques de consulta a otros y la selección de los planes con base únicamente en los costes estimados. Para aprovechar estas oportunidades, el optimizador de consultas de SQL Server se desvía de los enfoques tradicionales de la optimización de consultas usados en otros sistemas comerciales en favor de un entorno más general, puramente algebraico, que se basa en el prototipo de optimizador Cascades. La optimización de las consultas forma parte de su proceso de compilación, que consta de cuatro pasos:

- **Análisis/vinculación**. El analizador resuelve los nombres de tablas y columnas mediante los catálogos. SQL Server usa una caché de plan para evitar repetir la optimización de consultas idénticas o estructuralmente parecidas. Si no se dispone de plan guardado en la caché, se genera un árbol de operadores inicial. El árbol de operadores no es más que una combinación de operadores relacionales y no está restringido por conceptos como los bloques de consulta o las tablas derivadas, que suelen obstaculizar la optimización.
- **Simplificación/normalización.** El optimizador aplica las reglas de simplificación al árbol de operadores para obtener una forma normal y simplificada. Durante la simplificación, el optimizador determina y carga las estadísticas necesarias para la estimación de la cardinalidad.
- **Optimización basada en el coste.** El optimizador aplica las reglas de exploración y de implementación para generar alternativas, estimar el coste de ejecución y escoger el plan con el coste anticipado más bajo. Las reglas de exploración implementan la reordenación de un amplio conjunto de operadores, incluida la reordenación de la reunión y de la agregación. Las reglas de implementación introducen alternativas de ejecución como las reuniones por mezcla y las reuniones por asociación.
- **Preparación del plan.** El optimizador crea las estructuras del plan de ejecución para el plan seleccionado.

Para obtener mejores resultados, la optimización basada en el coste de SQL Server no se divide en fases que optimicen distintos aspectos de la consulta por separado; además, no está restringida a una sola dimensión, como puede ser la enumeración de reuniones. En vez de eso, un conjunto de reglas de transformación define el espacio de interés, y la estimación del coste se usa de manera uniforme para seleccionar un plan eficiente.

29.4.2 Simplificación de las consultas

Durante la simplificación sólo se aplican las transformaciones que garantizan la generación de sustitutos menos costosos. El optimizador envía las selecciones tan abajo del árbol de operadores como sea posible; comprueba los predicados en búsqueda de contradicciones, teniendo en cuenta las restricciones declaradas. Usa las contradicciones para identificar subexpresiones que se puedan eliminar del árbol. Una situación frecuente es la eliminación de las ramas **union** que recuperan los datos de las tablas con diferentes restricciones.

Una serie de reglas de simplificación son *dependientes del contexto*, es decir, la sustitución solamente es válida en el contexto de uso de la subexpresión. Por ejemplo, una reunión externa se puede simplificar en reunión interna si una operación de filtrado posterior elimina las reglas no coincidentes que se rellenaron con **null**. Otro ejemplo es la eliminación de las reuniones sobre las claves externas, que no hace falta ejecutar si no hay uso posterior de las columnas de la tabla a la que se hace referencia. Un tercer ejemplo es el contexto de insensibilidad a los duplicados, que especifica que la entrega de una o más copias de una fila no afecta al resultado de la consulta. Las subexpresiones bajo las semirreuniones y bajo **distinct** son insensibles a los duplicados, lo que permite cambiar **union** por **union all**.

Para la agrupación y la agregación se usa el operador *GbAgg*, que crea grupos y, opcionalmente, aplica una función agregada a cada grupo. La eliminación de duplicados, expresada en SQL mediante la palabra clave **distinct** es sencillamente un *GbAgg* sin funciones agregadas que calcular. Durante la simplificación, la información sobre las claves y sobre las dependencias funcionales se usa para reducir el agrupamiento de columnas.

Las subconsultas se normalizan mediante la eliminación de las especificaciones de consulta correlacionadas y el uso de algunas variantes de la reunión en su lugar. La eliminación de las correlaciones no es una "estrategia de ejecución de subconsultas", sino simplemente un paso de la normalización. Luego se considera una serie de estrategias de ejecución, durante la optimización basada en el coste.

29.4.3 Reordenación y optimización basadas en el coste

En SQL Server las transformaciones se integran completamente en la generación basada en el coste y en la selección de los planes de ejecución. El optimizador de consultas de SQL Server incluye alrededor de trescientas cincuenta reglas de transformación lógica y física. Además de la reordenación de la reunión interna, el optimizador de consultas usa transformaciones de reordenación para los operadores reunión externa, semirreunión y antisemirreunión del álgebra relacional estándar (con duplicados para SQL). También se reordena *GbAgg*, trasladándolo por debajo de las reuniones siempre que sea posible. La agregación parcial, esto es, la introducción de un nuevo *GbAgg* con agrupación sobre un superconjunto de las columnas de un *GbAgg* posterior, se considera por debajo de las reuniones y de **union all**, y también en los planes paralelos. Véanse las referencias dadas en las notas bibliográficas para obtener más detalles.

La ejecución correlacionada se considera durante la exploración del plan; el caso más simple es una reunión de búsqueda en el índice. SQL Server modela la ejecución correlacionada como un operador algebraico único, denominado **apply**, que opera sobre la tabla T y la expresión relacional parametrizada $E(t)$. **Apply** ejecuta E para cada fila de T, que proporciona los valores de los parámetros. La ejecución correlacionada se considera como una alternativa a la ejecución, independientemente del uso de subconsultas en la formulación original de SQL. Es una estrategia muy eficiente cuando la tabla T es muy pequeña y los índices soportan la ejecución parametrizada eficiente de $E(t)$. Además, se considera la reducción del número de ejecuciones de $E(t)$ cuando hay valores duplicados de los parámetros mediante dos técnicas: ordenar T según el valor de los parámetros, de forma que se reutilice un único resultado de $E(t)$ mientras que el valor del parámetro sigue siendo el mismo, o usar una tabla de asociación

que realice un seguimiento del resultado de $E(t)$ para (algún subconjunto de) los valores anteriores del parámetro.

Algunas aplicaciones seleccionan las filas según algún resultado agregado obtenido para su grupo. Por ejemplo "Hallar los clientes cuyo saldo sea mayor que el doble de la media de su segmento de mercado". La formulación de SQL exige una autorreunión. Durante la exploración se detecta este patrón y se considera la ejecución por segmentos como alternativa a la autorreunión.

También se considera el uso de vistas materializadas durante la optimización basada en el coste. Las interacciones de la coincidencia de vistas con la ordenación de operadores en ese uso puede que no resulte evidente hasta que se haya tenido lugar otra reordenación. Cuando se encuentra que una vista coincide con alguna subexpresión, la tabla que contiene el resultado de esa vista se agrega como alternativa a la expresión correspondiente. En función de la distribución de los datos y de los índices disponibles, puede que sea mejor que la expresión original—la selección se realizará en términos de la estimación del coste.

Para estimar el coste de ejecución del plan el modelo tiene en cuenta el número de filas que se espera procesar, denominado objetivo de filas, así como el número de veces que se ejecuta cada subexpresión. El número de filas puede ser menor que la estimación de la cardinalidad en casos tales como *Apply/semijoin*. *Apply/semijoin* devuelve la fila t de T tan pronto como $E(t)$ produce una fila (es decir, comprueba que $E(t)$ **existe**). Por tanto, el objetivo de filas del resultado de $E(t)$ es 1, y los objetivos de filas de los subárboles de $E(t)$ se calculan para este objetivo de filas de $E(t)$ y se usan para la estimación del coste.

29.4.4 Planes de actualización

Los planes de actualización optimizan el mantenimiento de índices, comprueban las restricciones, aplican las acciones en cascada y mantienen las vistas materializadas. Para el mantenimiento de los índices, en lugar de tomar cada fila y mantener todos sus índices, los planes de actualización aplican las modificaciones índice a índice, ordenando las filas y aplicando la operación **update** según el orden de la clave. Esto minimiza las operaciones aleatorias de E/S, especialmente cuando el número de filas que hay que actualizar es grande. Las restricciones las maneja un operador **assert**, que ejecuta un predicado y envía un mensaje de error si el resultado es **false**. Las restricciones de integridad referencial se definen mediante predicados **exist** que, a su vez, se convierten en semirreuniones y se optimizan considerando todos los algoritmos de ejecución.

El problema de Halloween se aborda usando elecciones basadas en el coste. El problema de Halloween hace referencia a la siguiente anomalía: supóngase que se lee un índice salarial en orden ascendente y se están subiendo los sueldos un diez por ciento. Como resultado de la actualización, las filas se desplazarán hacia arriba en el índice, se volverán a encontrar y se actualizarán de nuevo, lo que lleva a un bucle infinito. Una forma de abordar este problema es separar el procesamiento en dos fases: en primer lugar se leen todas las filas que se van a actualizar y se hace una copia de ellas en algún emplazamiento temporal, después se leen desde ese emplazamiento y se aplican todas las actualizaciones. Otra alternativa es leer desde un índice distinto donde las filas no se trasladen como consecuencia de la actualización. Algunos planes de ejecución proporcionan la separación de las fases de forma automática, si se ordena o crea una tabla de asociación con las filas que se van a actualizar. En el optimizador de SQL Server la protección contra Halloween se modela como una de las propiedades de los planes. Se generan varios planes que proporcionan la propiedad requerida y se selecciona uno en función del coste de ejecución estimado.

29.4.5 Análisis de los datos durante la optimización

SQL fue de los primeros en introducir técnicas para llevar a cabo la recogida de estadísticas como parte de las optimizaciones en proceso. El cálculo de las estimaciones de tamaño del resultado se basa en las estadísticas de las columnas usadas en una expresión dada. Estas estadísticas consisten en histogramas de diferencias máximas de los valores de las columnas y en varios contadores que capturan la densidad y el tamaño de las filas, entre otras cosas. Los administradores de las bases de datos pueden crear estadísticas de manera explícita mediante la sintaxis extendida de SQL.

Si no se dispone de estadísticas para una columna determinada, no obstante, el optimizador de SQL Server detiene la optimización en proceso y reúne las estadísticas necesarias. En cuanto se han calculado

las estadísticas, se reanuda la optimización original, que aprovechará las estadísticas recién creadas. La optimización de las consultas posteriores reúsa las estadísticas generadas anteriormente. Normalmente, tras un breve periodo de tiempo, ya se han creado las estadísticas de las columnas usadas con frecuencia y las interrupciones para la elaboración de estadísticas nuevas se hacen menos frecuentes. Mediante el seguimiento del número de filas modificadas en cada tabla se tiene una medida de la antigüedad de todas las estadísticas afectadas. Una vez que la antigüedad supera un cierto umbral, se vuelven a calcular esas estadísticas y los planes guardados en la caché se vuelven a compilar para que tengan en cuenta las distribuciones de datos modificadas.

SQL Server 2005 puede llevar a cabo el cálculo automático de las estadísticas de manera asíncrona. Esto evita los tiempos de compilación potencialmente largos debidos a la elaboración síncrona de las estadísticas. La optimización que desencadena el cálculo de las estadísticas usa estadísticas potencialmente antiguas. No obstante, las consultas posteriores pueden aprovechar las estadísticas recalculadas. Esto permite lograr un equilibrio aceptable entre el tiempo empleado en la optimización y la calidad del plan de consultas resultante.

29.4.6 Búsquedas parciales y heurísticas

Los optimizadores basados en el coste se enfrentan con el problema de la explosión del espacio de búsqueda, puesto que las aplicaciones realizan consultas que implican a docenas de tablas. Para abordar esto, SQL Server usa varias etapas de optimización, cada una de los cuales usa transformaciones de la consulta para explorar regiones sucesivamente mayores del espacio de búsqueda.

Hay transformaciones sencillas y completas diseñadas para la optimización exhaustiva, así como transformaciones inteligentes que implementan varias heurísticas. Las transformaciones inteligentes generan planes que están muy lejos entre sí en el espacio de búsqueda, mientras que las transformaciones sencillas exploran zonas vecinas. Las etapas de optimización aplican una mezcla de ambas clases de transformaciones, poniendo en primer lugar el énfasis en las transformaciones inteligentes y pasando luego a las transformaciones sencillas. Se conservan los resultados óptimos de los subárboles, de forma que las etapas posteriores puedan aprovechar los resultados generados con anterioridad. Cada etapa debe equilibrar técnicas de generación de planes opuestas:

- **Generación exhaustiva de alternativas.** Para generar el espacio completo el optimizador usa transformaciones completas, locales, no redundantes—una regla de transformación equivalente a una secuencia de transformaciones más primitivas solamente introduce una sobrecarga adicional.
- **Generación heurística de candidatos.** Es probable que una serie de candidatos interesantes (seleccionados en términos del coste estimado) estén lejos entre sí en términos de las reglas de transformación primitivas. En este caso, las transformaciones deseadas son incompletas, globales y redundantes.

La optimización se puede terminar en cualquier momento tras la generación del primer plan. Esta terminación se basa en el coste estimado del mejor plan encontrado y en el tiempo ya empleado en la optimización. Por ejemplo, si una consulta sólo necesita buscar unas cuantas filas de algunos índices, se producirá rápidamente un plan muy barato en las primeras etapas, lo que terminará la optimización. Este enfoque permite agregar fácilmente nuevas heurísticas en el transcurso del tiempo, sin comprometer la selección de planes basada en el coste ni la exploración exhaustiva del espacio de búsqueda, cuando resulta conveniente.

29.4.7 Ejecución de las consultas

Los algoritmos de ejecución soportan tanto el procesamiento basado en la ordenación como el basado en la asociación, y sus estructuras de datos se diseñan para optimizar el uso de la caché del procesador. Las operaciones de asociación soportan la agregación y la reunión básicas, con una serie de optimizaciones, extensiones y ajustes dinámicos del sesgo de datos. La operación **flow–distinct** es una variante diferente de la asociación (hash distinct), en la que las filas se devuelven antes, tan pronto como se encuentra un valor diferente nuevo, en lugar de esperar a procesar todos los datos de entrada. Este operador es

efectivo para las consultas que usan **distinct** y sólo piden unas cuantas filas, por ejemplo, cuando se usa el constructor **top n**. Los planes correlacionados especifican la ejecución de $E(t)$, e incluyen a menudo varias búsquedas en el índice basadas en el parámetro, para cada fila t de la tabla T. La *captura previa asíncrona* permite la solicitud al motor de almacenamiento de varias solicitudes de búsqueda en el índice. Se implementa de esta manera: se realiza una solicitud de búsqueda en el índice sin bloqueo para la fila t de T, luego t se coloca en la cola de captura previa. Las filas se sacan de la cola y **apply** las usa para ejecutar $E(t)$. La ejecución de $E(t)$ no necesita que los datos se hallen ya en la memoria intermedia agrupada, pero tener buenas operaciones de captura previa maximiza el uso del hardware e incrementa el rendimiento. El tamaño de la cola se determina dinámicamente como función de los aciertos de la caché. Si no se necesita ninguna ordenación de las filas de salida de **apply**, las filas de esa cola se pueden tomar sin prestar atención al orden, para minimizar la espera en las operaciones de E/S.

La ejecución en paralelo la implementa el operador **exchange**, que gestiona varias hebras, realiza particiones o difunde datos y proporciona los datos a varios procesos. El optimizador de consultas decide la ubicación de **exchange** según el coste estimado. El grado de paralelismo se determina dinámicamente en el momento de la ejecución, en función del uso del sistema en ese momento.

Los planes de índices están constituidos por los fragmentos que se han descrito anteriormente. Por ejemplo, se considera el uso de una reunión de índices para resolver las conjunciones de predicados (o una unión de índices para las disyunciones), en términos de su coste. Esta reunión se puede realizar en paralelo, usando cualquiera de los algoritmos de reunión de SQL Server. También se consideran reuniones de índices con el único propósito de ensamblar una fila con el conjunto de columnas necesario en una consulta, lo cual, a veces, es más rápido que explorar una tabla base. Tomar los identificadores de registros de un índice secundario y localizar la fila correspondiente de la tabla base es equivalente, en efecto, a ejecutar una reunión de búsqueda de índices. Para ello se usan las técnicas genéricas de ejecución correlacionada, como la captura previa asíncrona.

La comunicación con el motor de almacenamiento se realiza mediante OLE-DB, lo que permite el acceso a otros proveedores de datos que también implementan esa interfaz. OLE-DB es el mecanismo usado para las consultas distribuidas y para las remotas, que maneja directamente el procesador de consultas. Los proveedores de datos se clasifican según el rango de funcionalidad que proporcionan, desde simples proveedores de conjuntos de filas sin capacidades de indexado a proveedores con soporte completo de SQL.

29.5 Concurrencia y recuperación

Los subsistemas de transacciones, registro histórico, bloqueos y recuperación de SQL Server hacen que se cumplan las propiedades ACID esperadas de los sistemas de bases de datos.

29.5.1 Transacciones

El control de concurrencia basado en los bloqueos es el predeterminado para SQL Server. SQL Server también ofrece el control de concurrencia optimista para los cursores. El control de concurrencia optimista se basa en la suposición de que los conflictos de recursos entre varios usuarios son poco probables (aunque no imposibles) y permite que las transacciones se ejecuten sin bloquear ningún recurso. Sólo cuando se intenta modificar los datos, SQL Server comprueba los recursos para determinar si se ha producido algún conflicto. Si se produce algún conflicto, la aplicación debe leer los datos e intentar el cambio de nuevo. Las aplicaciones pueden elegir si se detectan los cambios comparando los valores o comprobando la columna especial rowversion de cada fila.

SQL Server soporta los niveles de aislamiento de SQL de lectura no comprometida, lectura comprometida, lectura repetible y serializable. La lectura comprometida es el nivel predeterminado. Además, SQL Server soporta dos niveles de aislamiento basados en las instantáneas.

- **Instantánea.** Especifica que los datos leídos por cualquier instrucción de la transacción son la versión consistente transaccionalmente de los datos que existían al comienzo de esa transacción. La transacción sólo puede ver las modificaciones de los datos que se comprometieron antes de su comienzo. Las modificaciones de los datos llevadas a cabo por otras transacciones tras el comienzo de esta transacción no son visibles para las instrucciones que se ejecutan en la transacción

actual. El efecto es como si las instrucciones de la transacción vieran una instantánea de los datos comprometidos tal y como eran al principio de la transacción.

- **Instantánea de lectura comprometida.** Especifica que cada instrucción ejecutada en la transacción ve una instantánea transaccionalmente consistente de los datos tal y como eran al comienzo de la instrucción. Las modificaciones de los datos llevadas a cabo por otras transacciones tras el comienzo de la instrucción no son visibles para ella. Esto es diferente de lo que ocurre con el aislamiento de lectura comprometida, en el que la instrucción puede ver las actualizaciones comprometidas de las transacciones que se comprometen mientras se ejecuta.

29.5.2 Bloqueos

El bloqueo es el principal mecanismo de los usados para hacer cumplir la semántica de los niveles de aislamiento. Todas las actualizaciones adquieren los bloqueos exclusivos suficientes, mantenidos toda la duración de la transacción, para evitar que se produzcan actualizaciones que entren en conflicto entre sí. Los bloqueos compartidos se mantienen en duraciones diferentes para proporcionar los diversos niveles de aislamiento de SQL para las consultas.

SQL Server proporciona bloqueos de varias granularidades que permiten que cada transacción bloquee diferentes tipos de recursos (véase la Figura 29.4, en la que los recursos se relacionan en orden creciente de granularidad). Para minimizar el coste de los bloqueos, SQL Server bloquea de manera automática los recursos con la granularidad apropiada para cada tarea. El bloqueo con una granularidad menor, como pueden ser las filas, aumenta la concurrencia, pero tiene una sobrecarga mayor, ya que hay que realizar más bloqueos si se bloquean muchas filas.

Los modos de bloqueos fundamentales de SQL Server son el compartido (S, shared), el de actualización (U, update) y el exclusivo (X, exclusive); los bloqueos intencionales (intent) se usan para evitar una forma frecuente de interbloqueo que se produce cuando varias sesiones leen, bloquean y, potencialmente, actualizan posteriormente los recursos. Los modos adicionales de bloqueo—denominados bloqueos de rango de claves—sólo se toman en el nivel de aislamiento serializable para bloquear el rango entre dos filas de un índice.

29.5.2.1 Bloqueos dinámicos

Los bloqueos de granularidad fina pueden mejorar la concurrencia a cambio de ciclos adicionales de CPU y de memoria extra para adquirir y mantener muchos bloqueos. Para muchas consultas, una granularidad de bloqueo más gruesa proporciona mejor rendimiento sin pérdida (o con una pérdida mínima) de la concurrencia . Los sistemas de base de datos han exigido tradicionalmente sugerencias de consulta y opciones de tabla para que las aplicaciones especificaran la granularidad del bloqueo. Además, hay parámetros de configuración (frecuentemente estáticos) para la cantidad de memoria que se debe dedicar al administrador de bloqueos.

En SQL Server la granularidad del bloqueo se optimiza automáticamente para un rendimiento y una concurrencia óptimos de cada índice de la consulta. Además, la memoria dedicada al administrador de bloqueos se ajusta dinámicamente según la realimentación de las demás partes del sistema, incluidas otras aplicaciones de la máquina.

La granularidad del bloqueo se optimiza antes de la ejecución de las consultas para cada tabla y para cada índice usados en esa consulta. El proceso de optimización del bloqueo tiene en cuenta el nivel de

Recurso	*Descripción*
RID	Identificador de fila, usado para bloquear una sola fila de la tabla.
Clave	Bloqueo de fila en un índice; protege rangos de la clave en transacciones secuenciables.
Página	Página de tabla o de índice de ocho kilobytes.
Extensión	Grupo contiguo de ocho páginas de datos o de índices.
Tabla	Tabla completa, incluidos todos los datos y todos los índices.
BD	Base de datos.

Figura 29.4 Recursos bloqueables.

aislamiento (esto es, el tiempo que se mantienen los bloqueos), el tipo de exploración (rango, sonda o toda la tabla), el número estimado de filas que hay que explorar, la selectividad (porcentaje de filas visitadas que son aceptables para la consulta) la densidad de las filas (número de filas por página), el tipo de operación (exploración, actualización), los límites del usuario sobre la granularidad y la memoria de sistema disponible.

Una vez se está ejecutando la consulta, la granularidad de bloqueo se dimensiona automáticamente hasta el nivel de las tablas si el sistema adquiere significativamente más bloqueos que los esperados por el optimizador, o si la cantidad de memoria disponible baja y no se puede soportar el número de bloqueos necesario.

29.5.2.2 Detección de los interbloqueos

SQL Server detecta de forma automática los interbloqueos que involucran tanto a bloqueos como a otros recursos. Por ejemplo, si la transacción A mantiene un bloqueo sobre Tabla1 y está esperando que haya memoria disponible y la transacción B tiene algo de memoria que no puede liberar hasta que adquiera un bloqueo sobre Tabla1, las transacciones sufrirán un interbloqueo. Las hebras y las memorias intermedias de comunicación también pueden estar implicados en los interbloqueos. Cuando SQL Server detecta un interbloqueo, elige como víctima del interbloqueo la transacción que es menos costosa de hacer retroceder, considerando la cantidad de trabajo que la transacción ya ha realizado.

La detección frecuente de interbloqueos puede perjudicar al rendimiento del sistema. SQL Server ajusta automáticamente la frecuencia de detección de los interbloqueos a la frecuencia con la que se producen. Si los interbloqueos no son frecuentes, el algoritmo de detección se ejecuta cada cinco segundos. Si son frecuentes, comenzará a comprobar si hay alguno cada vez que una transacción espere un bloqueo.

29.5.2.3 Versiones de las filas para el aislamiento de instantáneas

Los dos niveles de aislamiento basados en las instantáneas usan las versiones de las filas para conseguir el aislamiento de las consultas sin bloquear las consultas que se hallan tras las actualizaciones, y viceversa. Bajo el aislamiento de instantáneas las operaciones de actualización y de borrado generan versiones de las filas afectadas y las guardan en una base de datos temporal. El sistema se deshace de estas versiones cuando no hay ninguna transacción activa que pueda necesitarlas. Por tanto, las consultas ejecutadas bajo el aislamiento de instantáneas no necesita adquirir bloqueos y, en vez de eso, puede leer las versiones más antiguas de cualquier registro que otra transacción actualice o borre. Las versiones de las filas se usan también para proporcionar instantáneas de tablas para las operaciones de creación de índices en línea.

29.5.3 Recuperación y disponibilidad

SQL Server está diseñado para recuperarse de los fallos del sistema y de los medios, y el sistema de recuperación se puede adaptar a máquinas con grupos de memorias intermedias de tamaño muy grande (cien gigabytes) y millares de unidades de disco.

29.5.3.1 Recuperación de caídas

El registro histórico es, desde un punto de vista lógico, una corriente potencialmente infinita de registros históricos identificados por los números de secuencia del registro histórico (Log Sequence Numbers, LSNs). Desde un punto de vista físico, parte de la corriente se almacena en los archivos del registro histórico. Los registros históricos se guardan en los archivos del registro histórico hasta que se realiza una copia de seguridad y el sistema ya no los necesita para el retroceso o para la duplicación. Los archivos del registro histórico aumentan y disminuyen de tamaño para acomodarse a los registros que hay que almacenar. Se pueden añadir más archivos del registro histórico a la base de datos (en nuevos discos, por ejemplo) mientras que el sistema se está ejecutando y sin bloquear ninguna operación actual, y todos los registros históricos se tratan como si fueran un archivo continuo.

El sistema de recuperación de SQL Server tiene muchos aspectos en común con el algoritmo de recuperación ARIES (véase el Apartado 17.8.6), y en este apartado se muestran algunas de las diferencias fundamentales.

SQL Server tiene una opción de configuración denominada **intervalo de recuperación**, que permite que el administrador limite el tiempo que SQL Server debe tardar en recuperarse después de una caída. El servidor ajusta dinámicamente la frecuencia de los puntos de comprobación para reducir el tiempo de recuperación a valores comprendidos dentro del intervalo de recuperación. Los puntos de comprobación eliminan todas las páginas desfasadas de la memoria intermedia agrupada y se ajustan a las capacidades del sistema de E/S y a su carga de trabajo actual para eliminar de forma efectiva cualquier efecto sobre las transacciones que se estén ejecutando.

En el inicio, después de una caída, el sistema inicia varias hebras (dimensionadas automáticamente al número de CPUs) para iniciar la recuperación de varias bases de datos en paralelo. La primera fase de la recuperación es una pasada de análisis del registro histórico, que crea una tabla de páginas desfasadas y una lista de transacciones activas. La siguiente fase es una pasada de recreación que se inicia desde el último punto de comprobación y rehace todas las operaciones. Durante la fase de recreación se usa la tabla de páginas desfasadas para leer anticipadamente las páginas de datos. La fase final es una fase de destrucción en la que se retroceden las transacciones incompletas. La fase de destrucción se divide realmente en dos partes, puesto que SQL Server usa un esquema de recuperación de dos niveles. Las transacciones del primer nivel (aquéllas que implican operaciones internas como la asignación de espacio y las divisiones de páginas) retroceden en primer lugar, seguidas por las transacciones de los usuarios. Una vez que las transacciones del primer nivel han retrocedido, la base de datos se conecta al exterior y queda disponible para que comiencen nuevas transacciones de los usuarios mientras se llevan a cabo las últimas operaciones de retroceso. Esto se consigue haciendo que la pasada de recreación vuelva a adquirir los bloqueos para todas las transacciones de usuario incompletas que retrocederán en la fase de destrucción.

29.5.3.2 Recuperación de los medios

Las capacidades de copia de seguridad y de restauración de SQL Server permiten que se recupere de muchos fallos, incluidos la pérdida o corrupción de los medios de disco, los errores del usuario y la pérdida permanente de servidores. Además, la realización de copias de seguridad y de restauraciones de las bases de datos es útil para otros fines, como la copia de bases de datos de un servidor a otro y el mantenimiento de sistemas en espera.

SQL Server tiene tres modelos de recuperación diferentes entre los que los usuarios pueden elegir para cada base de datos. Mediante la especificación del modelo de recuperación, el administrador declara el tipo de capacidades de recuperación necesarias (como la restauración en un momento determinado y el envío de registros históricos) y las copias de seguridad necesarias para conseguirlas. Se pueden realizar copias de seguridad de las bases de datos, de los archivos, de los grupos de archivos y del registro histórico de transacciones. Todas las copias de seguridad son difusas y se realizan completamente en línea; es decir, no bloquean ninguna operación de LMD ni de LDD mientras se ejecutan. Las recuperaciones también se pueden llevar a cabo en línea, de modo que sólo se deje desconectada la parte de la base de datos que se está recuperando (por ejemplo, un bloque de disco corrupto). Las operaciones de copia de seguridad y de restauración están muy optimizadas y sólo quedan limitadas por la velocidad de los medios a los que se dirige la copia de seguridad. SQL Server puede realizar copias de seguridad tanto en dispositivos de disco como en los de cinta (hasta sesenta y cuatro en paralelo) y tiene APIs de gran rendimiento para usarlas con productos de copia de seguridad de otros fabricantes.

29.5.3.3 Copias exactas de las bases de datos

El uso de copias exactas de las bases de datos supone la reproducción inmediata de todas las actualizaciones de una de las bases de datos (la base de datos principal) en una copia diferente y completa de la base de datos (la base de datos copia exacta) que se suele ubicar en otra máquina. En caso de desastre en el servidor principal, o simplemente por labores de mantenimiento, el sistema puede recurrir de manera inmediata a la copia exacta en cuestión de segundos. Se consigue un estrecho acoplamiento entre la base

de datos principal y la copia exacta mediante el envío de bloques de registros históricos de transacciones a la copia exacta a medida que se generan en la base de datos principal y mediante la reconstrucción de los registros del registro histórico en la copia exacta. En el modo de seguridad completa las transacciones no se pueden comprometer hasta que los registros del registro histórico de la transacción han llegado al disco de la copia exacta. La biblioteca de comunicaciones usadas por las aplicaciones es consciente de la realización de la copia exacta y se vuelve a conectar a ella de manera automática en caso de conmutación debida a un error.

29.6 Arquitectura del sistema

Cada ejemplar de SQL Server es un único proceso del sistema operativo, que es también un punto de referencia para las solicitudes de ejecución de SQL. Las aplicaciones interactúan con SQL Server mediante diferentes bibliotecas del lado del cliente (como ODBC y OLE-DB) con el fin de ejecutar SQL.

29.6.1 Agrupación de hebras en el servidor

Para minimizar el cambio de contextos en el servidor y para controlar el grado de multiprogramación, el proceso de SQL Server mantiene un grupo de hebras que ejecutan las solicitudes del cliente. Cuando llegan las solicitudes del cliente se les asigna una hebra en el que ejecutarse. La hebra ejecuta las instrucciones de SQL enviadas por el cliente y le devuelve el resultado. Una vez completada la solicitud del usuario, la hebra se devuelve al grupo de hebras. Además de las solicitudes de los usuarios, el grupo de hebras también se usa para asignar hebras para tareas internas que se ejecutan en segundo plano como:

- **Escritor diferido** (lazywriter). Esta hebra se dedica a garantizar que una cierta cantidad del grupo de memorias intermedias está libre y disponible en todo momento para su asignación por el sistema. Esta hebra también interactúa con el sistema operativo para determinar la cantidad óptima de memoria que debe consumir el proceso de SQL Server.
- **Punto de comprobación** (checkpoint). Esta hebra comprueba de forma periódica todas las bases de datos para mantener un intervalo de recuperación breve para el inicio de las bases de datos al reiniciar el servidor.
- **Monitor de interbloqueo** (deadlock monitor). Esta hebra supervisa otras hebras, buscando interbloqueos en el sistema. Es responsable de la detección de interbloqueos y también selecciona una víctima para permitir que el sistema progrese.

Cuando el procesador de consultas elige un plan paralelo para ejecutar una consulta determinada puede asignar varias hebras que trabajen en nombre de la hebra principal para ejecutar la consulta. Puesto que la familia de sistemas operativos de Windows NT proporciona soporte nativo de las hebras, SQL Server usa las hebras de NT para su ejecución. No obstante, SQL Server se puede configurar para que ejecute hebras en modo usuario además de las hebras del núcleo en sistemas de prestaciones muy elevadas para evitar el coste de un cambio de contexto del núcleo en los intercambios de hebras.

29.6.2 Gestión de la memoria

Hay muchos usos distintos de memoria en el proceso de SQL Server:

- **Grupo de memorias intermedias**. El mayor consumidor de memoria del sistema es el grupo de memorias intermedias. El grupo de memorias intermedias mantiene una caché de las páginas de la base de datos usadas más recientemente. Usa un algoritmo de sustitución de reloj con una política de robo sin fuerza; esto es, las páginas de la memoria intermedia con actualizaciones no comprometidas se pueden sustituir (“robar”), y no se fuerza su envío al disco cuando se compromete la transacción. Las memorias intermedias también obedecen el protocolo del registro histórico de escritura anticipada para garantizar la corrección de la recuperación de las caídas y de los medios.

- **Asignación de la memoria dinámica**. Se trata de la memoria que se asigna de forma dinámica para ejecutar solicitudes remitidas por el usuario.
- **Caché de planes y de ejecución**. Esta caché almacena los planes compilados para varias consultas que los usuarios han ejecutado previamente en el sistema. Esto permite que varios usuarios compartan el mismo plan (lo cual ahorra memoria) y también ahorra tiempo de compilación de la consulta para consultas parecidas.
- **Concesiones de grandes cantidades de memoria**. Para los operadores de consulta que consumen grandes cantidades de memoria, como las reuniones por asociación y las ordenaciones.

SQL Server usa un esquema elaborado de gestión de la memoria para dividir su memoria entre los varios usos que se han descrito. Un solo administrador de memoria gestiona de forma centralizada toda la memoria usada por SQL Server. El administrador de memoria es responsable de realizar de forma dinámica la división y la redistribución de la memoria entre los diversos consumidores de memoria del sistema. Distribuye esa memoria de acuerdo con un análisis de la relación entre costes y beneficios de la memoria para cualquier uso concreto. Hay disponible un mecanismo generalizado con infraestructura LRU para todos los componentes. Esta infraestructura de caché no sólo realiza un seguimiento del tiempo de vida de los datos guardados en la caché, sino también de los costes relativos de CPU y de E/S necesarios para crearlos y guardarlos en la caché. Esta información se usa para determinar los costes relativos de los diferentes datos guardados en la caché. El administrador de memoria se centra en la expulsión de los datos guardados en la caché que no han sido tocados recientemente y que eran baratos de guardar en la caché. Como ejemplo, es más probable que permanezcan en la memoria los planes de consulta complejos que necesitan segundos de CPU para compilarse que los planes triviales para frecuencias de acceso equivalentes.

El administrador de memoria interactúa con el sistema operativo para decidir de forma dinámica la cantidad de memoria que se debe consumir de la cantidad total de memoria del sistema. Esto permite que SQL Server sea bastante agresivo en el uso de la memoria del sistema, pero también que pueda devolver memoria al sistema cuando otros programas la necesiten sin causar excesivos fallos de página.

29.6.3 Seguridad

SQL Server ofrece mecanismos de seguridad generales y directivas para la autenticación, la autorización y el cifrado. Dos cosas son todavía más importantes para la seguridad de los usuarios: (1) la calidad de todo el código base y (2) la posibilidad de que los usuarios determinen si han protegido adecuadamente el sistema.

La calidad del código base se mejora haciendo que todos los desarrolladores y probadores del producto reciban formación de seguridad. Siempre que es posible, SQL Server usa las características de seguridad subyacentes del sistema operativo en vez de implementar las suyas propias. Además, se usan numerosas herramientas internas para analizar el código base, en búsqueda de posibles fallos de seguridad.

Varias de estas características se proporcionan para ayudar a los usuarios a proteger el sistema adecuadamente. Una de ellas es una directiva fundamental denominada "deshabilitado de forma predeterminada", por la que se deshabilitan de forma predeterminada muchos componentes usados escasamente de entre los que necesitan una preocupación adicional por la seguridad. Otra característica es un "analizador de buenas prácticas", que avisa a los usuarios sobre las configuraciones de los parámetros del sistema que pueden provocar vulnerabilidades de seguridad.

29.7 Acceso a los datos

SQL Server soporta las siguientes interfaces de programación de aplicaciones (Application Programming Interface, API) para la creación de aplicaciones intensivas en datos:

- **ODBC** (Conectividad abierta de bases de datos, Open Database Connectivity). Se trata de la implementación de Microsoft de la interfaz del nivel de llamadas (call−level interface, CLI) de la norma SQL:1999. Incluye los modelos de objetos—Remote Data Objects, RDOs (objetos de datos

remotos) y Data Access Objects, DAOs (objetos de acceso a datos)— que facilitan la programación de las aplicaciones de bases de datos multicapa a partir de lenguajes de programación como Visual Basic.

- **OLE-DB** (Object Linking and Embedding—Database, Vinculación e incrustación de objetos para bases de datos). Se trata de una API de bajo nivel orientada a sistemas diseñada para los programadores que crean componentes de bases de datos. La interfaz está construida de acuerdo con el modelo de objetos componentes (Component Object Model, COM) de Microsoft, y permite la encapsulación de servicios de bajo nivel de la base de datos como los proveedores de conjuntos de filas, los proveedores ISAM y los motores de consultas. OLE-DB se usa en SQL Server para integrar el procesador de consultas relacionales y el motor de almacenamiento y permitir la duplicación y el acceso distribuido a SQL y a otros orígenes externos de datos. Al igual que ODBC, OLE-DB incluye un modelo de objetos de nivel superior denominado Objetos de datos ActiveX (ActiveX Data Objects, ADO) para facilitar la programación de aplicaciones de bases de datos desde Visual Basic.
- **ADO.NET**. Se trata de una API más nueva diseñada para las aplicaciones escritas en lenguajes .NET como C# y Visual Basic.NET. Esta interfaz simplifica algunos patrones frecuentes de acceso a datos soportados por ODBC y por OLE-DB. Además, proporciona un nuevo modelo de *conjuntos de datos* (data set) para permitir las aplicaciones de acceso a datos desconectadas y sin estado.
- **DB-Lib**. La biblioteca de bases de datos para la API de C que se desarrolló específicamente para usarla con versiones anteriores de SQL Server anteriores a la norma SQL-92.
- **HTTP/SOAP**. Las aplicaciones pueden usar las solicitudes HTTP/SOAP para invocar las consultas y los procedimientos de SQL Server. Las aplicaciones pueden usar URLs que especifiquen raíces virtuales de Internet Information Server (IIS, Servidor de información de Internet) que hagan referencia a ejemplares de SQL Server. El URL puede contener una consulta XPath, una instrucción de Transact-SQL o una plantilla de XML.

29.8 Procesamiento de consultas heterogéneas distribuidas

La posibilidad de realizar consultas distribuidas heterogéneas de SQL Server permite formular consultas transaccionales y actualizaciones de gran variedad de orígenes relacionales y no relacionales mediante proveedores de datos OLE-DB que se ejecutan en una o más computadoras. SQL Server soporta dos métodos para hacer referencia a los orígenes de datos OLE-DB heterogéneos en las instrucciones Transact-SQL. El método de los nombres de servidor vinculados usa procedimientos almacenados del sistema para asociar el nombre de un servidor con cada origen de datos OLE-DB. Se puede hacer referencia a los objetos de estos servidores vinculados en las instrucciones de Transact-SQL usando el convenio de nombres de cuatro partes que se describe más adelante. Por ejemplo, si el nombre de un servidor vinculado de *DeptSQLSrvr* se define en otra copia de SQL Server, la siguiente instrucción hace referencia a una tabla de ese servidor:

```
select *
from DeptSQLSrvr.Northwind.dbo.Employees
```

En SQL Server los orígenes de datos OLE-DB se registran como servidores vinculados. Una vez que se define un servidor vinculado, se puede tener acceso a sus datos usando el nombre de cuatro partes

<servidor_vinculado>.<catálogo>.<esquema>.<objeto>

El ejemplo siguiente establece un servidor vinculado a un servidor Oracle mediante un proveedor OLE-DB para Oracle:

```
exec sp_addlinkedserver OraSvr, 'Oracle 7.3', 'MSDAORA', 'OracleServer'
```

Una consulta a este servidor vinculado se expresa como:

```
select *
from OraSvr.CORP.ADMIN.VENTAS
```

Además, SQL Server soporta funciones intrínsecas parametrizadas de tipo tabla denominadas **openrowset** y **openquery**, que permiten enviar consultas no interpretadas a proveedores o a servidores vinculados, respectivamente, en el dialecto soportado por cada proveedor. La siguiente consulta combina la información almacenada en un servidor de Oracle y en un servidor de Microsoft Index Server. Relaciona todos los documentos que contienen las palabras Datos y Acceso, junto con sus autores, ordenadas por el departamento y el nombre del autor.

```
select e.dept, f.AutorDoc, f.NombreArchivo
from OraSvr.Corp.Admin.Empleado e,
openquery(ArchivosEmp,
          'select AutorDoc, NombreArchivo
           from scope("c:\EmpDocs")
           where contains(' ' "Datos" near() "Acceso" ' ')>0') as f
where e.nombre = f.AutorDoc
order by e.dept, f.AutorDoc
```

El motor relacional usa las interfaces OLE-DB para abrir los conjuntos de filas de los servidores vinculados, capturar las filas y gestionar las transacciones. Para cada origen de datos OLE-DB al que se tiene acceso como servidor vinculado debe estar presente un proveedor OLE-DB en el servidor en el que se ejecuta SQL Server. El conjunto de operaciones de Transact-SQL que se pueden usar en un origen de datos OLE-DB concreto depende de las capacidades del proveedor OLE-DB. Siempre que sea efectivo en el coste, SQL Server envía las operaciones relacionales como reuniones, restricciones, proyecciones, ordenaciones y agrupaciones al origen de datos OLE-DB. SQL Server usa el coordinador de transacciones distribuidas de Microsoft (Microsoft Distributed Transaction Coordinator) y las interfaces de transacciones OLE-DB del proveedor para garantizar la atomicidad de las transacciones que abarcan varios orígenes de datos.

29.9 Duplicación

La duplicación (réplica) de SQL Server son un conjunto de tecnologías para la copia y distribución de datos y de objetos de las bases de datos entre unas bases de datos y otras, el seguimiento de las modificaciones y la sincronización entre las bases de datos para conservar la consistencia. Las versiones más recientes de la duplicación de SQL Server también ofrecen la duplicación en línea de la mayor parte de las modificaciones de los esquemas de las bases de datos sin necesidad de interrupciones ni reconfiguraciones.

Los datos se suelen duplicar para aumentar su disponibilidad. La duplicación pueden reunir datos corporativos desde sitios geográficamente dispersos con destino a informes y diseminar datos a usuarios remotos de redes de área local o a usuarios itinerantes de conexiones de acceso telefónico a redes o de Internet. La duplicación de Microsoft SQL Server también mejora el rendimiento de las aplicaciones mediante su dimensionado para mejorar el rendimiento total de lectura entre duplicados, como es habitual al proporcionar servicios de caché de datos de capa intermedia para sitios Web.

29.9.1 Modelo de duplicación

SQL Server introdujo la metáfora *Publicar–Suscribir* para la duplicación de las bases de datos y extiende esta metáfora de la industria editorial a sus herramientas de administración y supervisión de los duplicados.

El **publicador** es un servidor que pone los datos a disposición de otros servidores para su duplicación. El publicador puede tener una o más publicaciones, cada una de las cuales representa un conjunto

de datos y de objetos de la base de datos relacionados lógicamente. Los objetos discretos de cada publicación, incluidas las tablas, los procedimientos almacenados, las funciones definidas por el usuario, las vistas, las vistas materializadas, etc., se denominan **artículos**. Añadir un artículo a una publicación permite la personalización extensiva de la forma en que se duplica ese objeto, por ejemplo, las restricciones sobre los usuarios que pueden suscribirse para recibir esos datos y sobre la manera en que ese conjunto de datos debe filtrarse de acuerdo con la proyección o la selección de una tabla, mediante un filtro "horizontal" o "vertical", respectivamente.

Los **suscriptores** son servidores que reciben los datos duplicados de los publicadores. Los suscriptores se pueden suscribir, como resulte oportuno, sólo a las publicaciones que necesiten de uno o varios publicadores, independientemente del número o del tipo de opciones de duplicación que implemente cada uno. Dependiendo del tipo de opciones de duplicación seleccionadas, el suscriptor se puede usar como duplicado de sólo lectura o bien se pueden realizar modificaciones en los datos que se propagan automáticamente al publicador y, por consiguiente, al resto de duplicados. Los suscriptores también pueden volver a publicar los datos a los que se suscriben, dando soporte a una topología de duplicación tan flexible como la empresa necesite.

El **distribuidor** es un servidor que desempeña varios roles, en función de las opciones de duplicación escogidas. Como mínimo, se usa como repositorio de la información histórica y de los estados de error. En otros casos se usa también como cola intermedia de almacenamiento y entrega para redimensionar la entrega de la carga de duplicación a todos los suscriptores.

29.9.2 Opciones de duplicación

La duplicación de Microsoft SQL Server ofrece un amplio espectro de opciones tecnológicas. Para decidir sobre las opciones de duplicación adecuadas que se pueden usar, el diseñador de bases de datos debe determinar los requisitos de la aplicación con respecto a la operación autónoma del sitio implicado y el grado de consistencia transaccional necesario.

La **duplicación instantánea** copia y distribuye los datos y los objetos de la base de datos exactamente como aparecen en un momento dado. La duplicación instantánea no exige un seguimiento continuo de las modificaciones, puesto que los cambios no se propagan a los suscriptores de forma incremental. Los suscriptores se actualizan de manera periódica con una renovación completa del conjunto de datos definido por la publicación. Las opciones disponibles para la duplicación instantánea pueden filtrar los datos publicados y permitir que los suscriptores modifiquen los datos duplicados y propaguen esos cambios al publicador. Este tipo de duplicación es más indicado para datos de pequeño tamaño y cuando las actualizaciones suelen afectar a suficientes datos como para que la duplicación de una renovación completa de los datos resulte eficiente.

Con la **duplicación transaccional** el publicador propaga una instantánea inicial de los datos a los suscriptores y luego envía las modificaciones incrementales de esos datos a los suscriptores en forma de transacciones discretas y comandos. El seguimiento del cambio incremental se produce dentro del motor principal de SQL Server, que marca las transacciones que afectan a los objetos duplicados en el registro histórico de transacciones de la base de datos publicadora. Un proceso de duplicación denominado **agente de lector del registro** (histórico) lee estas transacciones del registro histórico de transacciones de la base de datos, aplica un filtro opcional y las almacena en la base de datos de distribución, que actúa como cola fiable que soporta el mecanismo de almacenamiento y entrega de la duplicación transaccional (el concepto de cola fiable es el mismo que el de colas duraderas descrito en el Apartado 25.1.1). Otro proceso de duplicación, denominado **agente de distribución**, envía luego las modificaciones a cada suscriptor. Al igual que la duplicación instantánea, la duplicación transaccional ofrece a los suscriptores la opción de realizar actualizaciones que utilicen un compromiso de dos fases que refleje esas modificaciones de forma consistente en el publicador o de colocarlas en cola en el suscriptor para su recuperación asíncrona por un proceso de duplicación que propague posteriormente la modificación al publicador. Este tipo de duplicación resulta adecuada cuando hay que conservar los estados intermedios entre varias actualizaciones.

La **duplicación por mezcla** permite que cada duplicado de la empresa funcione con total autonomía tanto en conexión como sin conexión. El sistema realiza un seguimiento de los metadatos según las modificaciones de los objetos publicados en los publicadores y en los suscriptores de todas las bases

de datos duplicadas y el agente de duplicación mezcla esas modificaciones de los datos durante la sincronización entre los pares duplicados y asegura la convergencia de los datos mediante la detección y resolución automática de los conflictos. El agente de duplicación usado en el proceso de sincronización incorpora numerosas opciones de políticas de resolución de conflictos, y se puede escribir una política de resolución de conflictos personalizada mediante el uso de procedimientos almacenados o de una interfaz extensible del **modelo de objetos componentes** (**Component Object Model**, **COM**). Este tipo de duplicación no duplica todos los estados intermedios, sino sólo el estado actual de los datos en el momento de la sincronización. Resulta adecuado cuando los duplicados necesitan la posibilidad de llevar a cabo actualizaciones autónomas mientras no se hallan conectadas a ninguna red.

29.10 Programación de servidores en .NET

SQL Server soporta el motor común en tiempo de ejecución de lenguajes .NET (.NET Common Language Runtime, CLR) en el proceso de SQL Server para permitir que los programadores de bases de datos escriban la lógica corporativa en forma de funciones, procedimientos almacenados, disparadores, tipos de datos y agregados. La posibilidad de ejecutar el código de las aplicaciones dentro de la base de datos añade flexibilidad al diseño de las arquitecturas de las aplicaciones que necesita que la lógica corporativa se ejecute cerca de los datos y no puede permitirse el coste de enviar los datos a un proceso de una capa intermedia para llevar a cabo el cálculo fuera de la base de datos.

CLR es un entorno de tiempo de ejecución con un lenguaje intermedio de tipos fuertes que ejecuta varios lenguajes de programación modernos como C#, Visual Basic, C++, COBOL y J++, entre otros, tiene memoria con recogida de basura, hebras con derecho preferente, servicios de metadatos (reflexión de tipos), comprobación de los tipos y seguridad de acceso al código. El tiempo de ejecución usa los metadatos para localizar y cargar las clases, dejar los ejemplares en la memoria, resolver las invocaciones a los métodos, generar código nativo, hacer que se cumpla la seguridad y definir las fronteras de contexto en el tiempo de ejecución.

El código de las aplicaciones se implanta en la base de datos mediante ensamblados, que son las unidades de empaquetado, implantación y versiones del código de las aplicaciones en .NET. La implantación del código de las aplicaciones en la base de datos proporciona una manera uniforme de administrar, realizar copias de seguridad de las aplicaciones de bases de datos completas y restaurarlas (el código y los datos). Una vez se ha registrado un ensamblado en la base de datos, los usuarios pueden mostrar los puntos de entrada del ensamblado mediante las instrucciones de LDD de SQL, que pueden actuar como funciones escalares o de tablas, procedimientos, disparadores, tipos y agregados mediante el uso de contratos de extensión bien definidos, que se hacen cumplir durante la ejecución de esas sentencias de LDD. Los procedimientos almacenados, los disparadores y las funciones suelen necesitar ejecutar consultas y actualizaciones de SQL. Esto se consigue mediante un componente que implementa el API de acceso a los datos ADO.NET para su uso dentro del proceso de la base de datos.

29.10.1 Conceptos básicos de .NET

En el entorno .NET los programadores escriben el código de los programas en lenguajes de programación de alto nivel que implementan clases que definen su estructura (por ejemplo, los campos o las propiedades de las clases) y sus métodos. Algunos de esos métodos pueden ser funciones estáticas. La compilación del programa genera un archivo, denominado *ensamblado*, que contiene el código compilado en el *lenguaje intermedio de Microsoft (Microsoft Intermediate Language, MSIL)* y un *manifiesto* que contiene todas las referencias a los ensamblados dependientes. El manifiesto es parte integral de cada ensamblado y permite que se describa a sí mismo. El manifiesto del ensamblado contiene los metadatos del ensamblado, que describen todas las estructuras, campos, propiedades, clases, relaciones de herencia, funciones y métodos definidos en el programa. El manifiesto establece la identidad del ensamblado, especifica los archivos que conforman su implementación, especifica los tipos y recursos que lo constituyen, divide en elementos las dependencias del momento de la compilación respecto de otros ensamblados y especifica el conjunto de permisos necesarios para que el ensamblado se ejecute correctamente. Esta información se usa en el momento de la ejecución para resolver las referencias, hacer que se cumpla la directiva de vinculación de versiones y validar la integridad de los ensamblados cargados. El entorno .NET sopor-

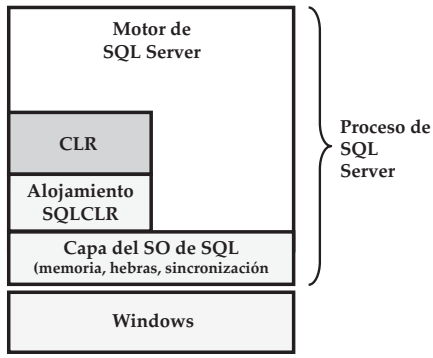


Figura 29.5 Integración de CLR con los servicios de sistema operativo de SQL Server.

ta un mecanismo marginal denominado *atributos personalizados* para anotar las clases, las propiedades, las funciones y los métodos con información adicional o con facetas que la aplicaciones puede desear capturar en los metadatos. Todos los compiladores .NET consumen estas anotaciones sin interpretarlas y las guardan en los metadatos del ensamblado. Todas esas anotaciones se pueden examinar del mismo modo que se examina cualquier otro metadato, mediante un conjunto común de APIs de reflexión. La expresión *código gestionado* hace referencia al MSIL ejecutado en CLR en lugar del ejecutado directamente por el sistema operativo. Las aplicaciones de código gestionado obtienen servicios en el momento de la ejecución de lenguajes comunes como la recogida automática de basura, la comprobación de tipos durante la ejecución y el soporte de seguridad. Estos servicios ayudan a ofrecer un comportamiento de las aplicaciones de código gestionado uniforme e independiente de las plataformas y de los lenguajes. Durante la ejecución un compilador sobre la marcha (JIT, just−in−time) traduce el MSIL a código nativo (por ejemplo, código de Intel X86). Durante esa traducción el código debe pasar un proceso de comprobación que examina el MSIL y los metadatos para averiguar si el código se puede considerar de tipos seguros.

29.10.2 CLR en SQL

SQL Server y CLR son dos motores de ejecución diferentes con modelos internos de hebras, programación y gestión de la memoria diferentes. SQL Server soporta un modelo de hebras cooperativo sin derecho preferente en el que las hebras del SGBD entregan voluntariamente la ejecución de manera periódica o cuando esperan por los bloqueos o en la cola de E/S, mientras que CLR soporta un modelo de hebras preventivo. Si el código del usuario que se ejecuta en el SGBD puede llamar directamente a las primitivas de multienhebramiento (subprocesamiento en DB2) del sistema operativo (SO), no se integra bien con el programador de tareas de SQL Server y puede degradar la capacidad de redimensionamiento del sistema. CLR no distingue entre la memoria virtual y la física, mientras que SQL Server gestiona directamente la memoria física y se le exige que utilice la memoria física dentro de unos límites configurables.

Los diferentes modelos de multienhebramiento, programación y gestión de la memoria suponen un reto de integración para los SGBDs que se redimensionan para soportar millares de sesiones de usuario concurrentes. SQL Server resuelve este reto pasando a ser el sistema operativo de CLR cuando se alberga en el proceso de SQL Server. CLR llama a las primitivas de bajo nivel implantadas por SQL Server para multienhebramiento, programación, sincronización y gestión de la memoria (véase la Figura 29.5). Este enfoque proporciona las ventajas de redimensionamiento y fiabilidad siguientes.

Multienhebramiento, programación y sincronización comunes. CLR llama a las APIs de SQL Server para crear hebras, tanto para ejecutar el código del usuario como para su propio uso interno, como las hebras del recogedor de basura y del destructor de clases. Para sincronizar varios hebras, CLR llama a los objetos de sincronización de SQL Server. Esto permite que el programador de SQL Server programe otras tareas mientras una hebra espera en un objeto de sincronización. Por ejemplo, cuando CLR inicia la recogida de basura, todas sus hebras esperan a que acabe la recogida de basura. Dado que las hebras de CLR y los objetos de sincronización en los que esperan son conocidos por el programador de SQL Server, puede programar hebras que estén ejecutando otras tareas de la base de datos que no impliquen

a CLR. Además, esto permite que SQL Server detecte los interbloqueos que implican a los bloqueos adoptados por los objetos de sincronización de CLR y a usar técnicas tradicionales para su eliminación. El programador de SQL Server tiene la posibilidad de detectar y de detener las hebras que no hayan aportado resultados durante un periodo de tiempo significativo. La posibilidad de vincular las hebras de CLR con las hebras de SQL Server implica que el programador de SQL Server pueda identificar las hebras fuera de control que se ejecutan en CLR y gestionar su prioridad, de modo que no consuman recursos significativos de la CPU y afecten, por tanto, al flujo del sistema. Esas hebras fuera de control se suspenden y se vuelven a poner en la cola. A los infractores reincidentes no se les permiten intervalos de tiempo que sean injustos para otros trabajadores que estén ejecutando instrucciones. Si un infractor toma cincuenta veces la cantidad permitida, es castigado durante cincuenta "rondas" antes de que se le vuelva a permitir ejecutar instrucciones, ya que el programador no puede decidir si un cálculo largo está fuera de control o es legítimo.

Gestión de la memoria común. CLR llama a las primitivas de SQL Server para la asignación y la desasignación de memoria. Dado que la memoria usada por CLR cuenta para el uso total de memoria del sistema, SQL Server puede seguir dentro de los límites de memoria del sistema configurados y garantizar que CLR y SQL Server no compitan entre sí por memoria. Además, SQL Server puede rechazar las solicitudes de memoria de CLR mientras el sistema está restringido y pedir al CLR que reduzca su uso de la memoria cuando otras tareas la necesiten.

29.10.3 Contratos para la extensión

Todo el código gestionado por el usuario que se ejecuta en el proceso de SQL Server interactúa con los componentes del SGBD como extensión. Entre las extensiones actuales están las funciones escalares, las tabulares, los procedimientos, los disparadores, los tipos escalares y los agregados escalares. Para cada extensión hay un contrato mutuo que define las propiedades o servicios que el código de usuario debe implementar para actuar como tal extensión, y los servicios que la extensión puede esperar del SGBD cuando se llame al código gestionado. CLR de SQL aprovecha la clase y la información de los atributos personalizados ya guardada en los metadatos del ensamblado para hacer que se cumpla que el código de usuario implemente esos contratos de extensión. Todos los ensamblados de usuario se guardan en la base de datos. Todos los metadatos relacionales y de los ensamblados se procesan en el motor de SQL mediante un conjunto uniforme de interfaces y de estructuras de datos. Cuando se procesan las instrucciones del lenguaje de definición de datos que registran una función, tipo o agregado de extensión dado, el sistema garantiza que el código de usuario implemente el contrato correspondiente mediante el análisis de los metadatos de su ensamblado. Si el contrato se implementa, la instrucción del DDL tiene éxito; en caso contrario, falla. Los subapartados siguientes describen aspectos fundamentales de los contratos concretos que SQL Server hace cumplir actualmente.

29.10.3.1 Rutinas

Las funciones escalares, los procedimientos y los disparadores se clasifican genéricamente como rutinas. Las rutinas, implementadas como métodos de clase estática, pueden especificar las propiedades siguientes mediante los atributos personalizados.

- **IsPrecise**. Si esta propiedad binaria es **false**, indica que el cuerpo de la rutina comprende cálculos imprecisos como las operaciones de coma flotante. Las expresiones que implican funciones imprecisas no se pueden indexar.
- **UserDataAccess**. Si el valor de esta propiedad es **read**, la rutina lee las tablas de datos de usuario. En caso contrario, el valor de la propiedad es None, lo que indica que la rutina no tiene acceso a los datos. Las consultas que no tienen acceso a las tablas de usuario (directamente o de manera indirecta mediante las vistas y las funciones) no se considera que tengan acceso a los datos de usuario.
- **SystemDataAccess**. Si el valor de esta propiedad es **read**, la rutina lee los catálogos del sistema o las tablas virtuales del sistema.

- **IsDeterministic**. Si esta propiedad es **true**, se da por supuesto que la rutina produce el mismo resultado, dados valores de entrada, estado de la base de datos local y contexto de ejecución iguales.
- **IsSystemVerified**. Esto indica si SQL Server puede asegurarse de las propiedades de determinismo y de precisión o hacerlas cumplir (por ejemplo, funciones predefinidas y de Transact-SQL) o son tal y como las especifique el usuario (por ejemplo, funciones de CLR).
- **HasExternalAccess**. Si el valor de esta propiedad es **true**, la rutina tiene acceso a los recursos externos a SQL Server, como los archivos, la red, el acceso Web y el registro.

29.10.3.2 Funciones que devuelven tablas

Las clases que implementan funciones que se valoran como tablas deben implementar la interfaz IEnumerable para permitir la iteración sobre las filas devueltas por la función, un método para describir el esquema de la tabla devuelta (es decir, las columnas, los tipos), un método para describir las columnas que pueden ser claves únicas y un método para insertar filas en la tabla.

29.10.3.3 Tipos

Las clases que implementan tipos definidos por los usuarios se anotan con el atributo SQLUserDefinedType(), que especifica las propiedades siguientes:

- **Format**. SQL Server soporta tres formatos de almacenamiento: nativo, definido por el usuario y serialización .NET.
- **MaxByteSize**. Se trata del tamaño máximo de la representación binaria serializada de los ejemplares de los tipos en bytes.
- **IsFixedLength**. Se trata de una propiedad booleana que especifica si los ejemplares del tipo tienen longitud fija o variable.
- **IsByteOrdered**. Se trata de una propiedad booleana que indica si la representación binaria serializada de los ejemplares del tipo tiene ordenación binaria. Cuando esta propiedad es **true**, el sistema puede realizar directamente comparaciones con esta representación sin necesidad de hacer que los ejemplares de los tipos sean objetos.
- **Nullability**. Todos los UDTs del sistema deben ser capaces de guardar el valor NULL mediante el soporte de la interfaz INullable que contiene el método booleano IsNull.
- **Type conversions**. Todos los UDTs deben implementar las conversiones de tipo directas e inversas con las cadenas de caracteres mediante los métodos ToString y Parse.

29.10.3.4 Agregados

Además de soportar los contratos de los tipos, los agregados definidos por los usuarios deben implementar cuatro métodos exigidos por el motor de ejecución de consultas para inicializar el cálculo de los ejemplares agregados, para acumular valores de entrada en la función proporcionada por el agregado, para mezclar cálculos parciales del agregado y para recuperar el resultado final del agregado. Los agregados pueden declarar propiedades adicionales mediante los atributos personalizados en su definición de clase; esas propiedades las usa el optimizador de consultas para obtener planes alternativos para el cálculo del agregado.

- **IsInvariantToDuplicates**. Si esta propiedad es **true**, el cálculo que entrega los datos al agregado puede modificarse descartando o introduciendo nuevas operaciones para la eliminación de duplicados.
- **IsInvariantToNulls**. Si esta propiedad es **true**, se pueden descartar las filas NULL de la entrada en algunos casos, pero hay que tener cuidado para no descartar grupos enteros.

- **IsInvariantToOrder**. Si esta propiedad es **true**, el procesador de consultas puede ignorar las cláusulas **order by** y explorar los planes que evitan tener que ordenar los datos.

29.11 Soporte de XML en SQL Server 2005

Los sistemas de bases de datos relacionales han adoptado XML de muchas maneras diferentes durante el último par de años. La primera generación del soporte de XML en los sistemas de bases de datos relacionales estaba relacionada sobre todo con la exportación de datos relacionales en forma de XML ("publicación XML") y en importar datos relacionales en forma de marcas de XML de nuevo a una representación relacional ("fragmentación de XML"). La situación en que más se usan estos sistemas es el intercambio de información en contextos en los que XML se usa como "formato cable" y los esquemas relacional y de XML suelen estar predefinidos de manera independiente entre sí. Para abordar esta situación, SQL Server 2005 de Microsoft ofrece y extiende la amplia funcionalidad introducida por primera vez en SQL Server 2000, como el agregador de conjuntos de filas para edición **for xml** el proveedor de conjuntos de filas OpenXML y la tecnología de vistas de XML basada en los esquemas anotados. Véanse las notas bibliográficas para obtener referencias que ofrezcan más información sobre las características de XML en SQL Server 2000.

La fragmentación de los datos XML en esquemas relacionales puede resultar bastante difícil o ineficiente para el almacenamiento de los datos semiestructurados cuya estructura puede variar con el tiempo y para almacenar documentos. Para soportar esas aplicaciones, SQL Server 2005 añade el soporte nativo de XML basado en el tipo de datos **xml** de SQL:2003. La Figura 29.6 ofrece un diagrama arquitectónico de alto nivel del soporte nativo de XML en las bases de datos de SQL Server. Consiste en la posibilidad de almacenar de manera nativa XML, restringir y tipificar los datos de XML almacenados con conjuntos de esquemas de XML y consultar y actualizar los datos de XML. Para proporcionar ejecuciones de consultas eficientes se proporcionan varios tipos de índices específicos de XML. Finalmente, el soporte nativo de XML también se integra con la "fragmentación" y la "publicación" directas e inversas de los datos relacionales.

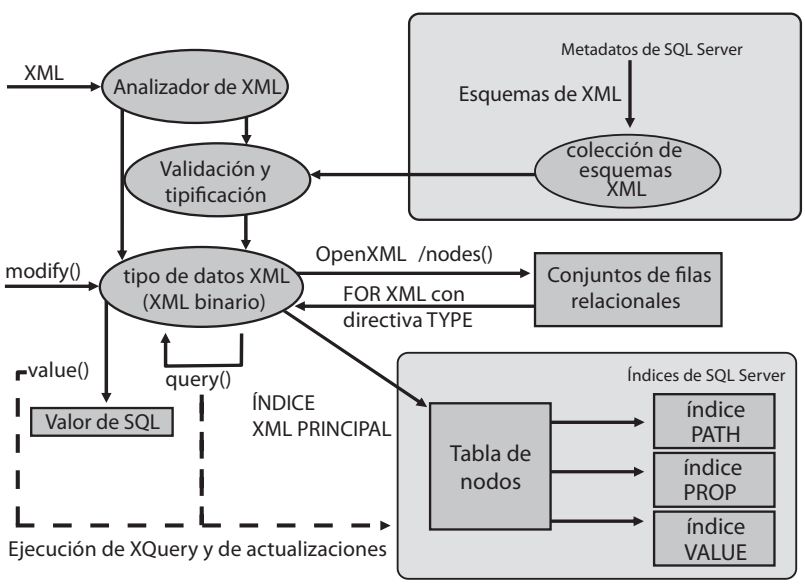


Figura 29.6 Visión general de la arquitectura del soporte nativo de XML en SQL Server.

29.11.1 Almacenamiento y organización nativos de XML

El tipo de datos **xml** puede guardar documentos de XML y contener fragmentos de contenido (varios nodos de texto o de elementos en la parte superior) y se define en términos del modelo de datos XQuery 1.0/XPath 2.0. Este tipo de datos se puede usar para parámetros de procedimientos almacenados, para variables y como tipo de columna. Por ejemplo, la siguiente instrucción de SQL crea una tabla de valores en la que una de las columnas es del tipo XML:

create table *InformesViajes*(*id* **int**,
fechaviaje **datetime**,
Informe **xml**)

El sistema de bases de datos puede decidir guardar un ejemplar del tipo **xml** en gran variedad de formatos, como objeto binario o carácter de gran tamaño, descompuesto en tablas o en una mezcla de formatos. SQL Server 2005 guarda los datos del tipo **xml** en un formato binario interno como **blob** y proporciona mecanismos de indexado para la ejecución de consultas. El formato binario interno proporciona una recuperación y una reconstrucción eficientes del documento de XML original, además de algún ahorro de espacio (en promedio, un veinte por ciento). Los índices soportan un mecanismo eficiente de consulta que puede usar el motor y el optimizador de consultas relacionales; se ofrecen detalles más adelante, en el Apartado 29.11.4.

29.11.2 Validación y tipificación de tipos de datos XML

SQL Server ofrece la posibilidad de restringir el tipo de datos de XML a un conjunto de esquemas de XML. SQL Server 2005 proporciona el concepto de metadatos de las base de datos denominado **conjunto de esquemas de XML**, que asocia un identificador de SQL a un conjunto de componentes de esquemas de uno o varios espacios de nombres objetivos. Por ejemplo, la expresión

create xml schema collection *E1* **as** *@e*

crea el conjunto de esquemas XML de SQL Server de nombre *E1*, que consiste en los esquemas de XML contenidos en la variable de SQL *@e*. Cada uno de estos conjuntos contiene toda la información necesaria para llevar a cabo la validación y tipificación y se guarda en los metadatos del esquema de la base de datos.

SQL Server 2005 permite asociar un conjunto de esquemas con un tipo de datos de XML y especificar si el tipo sólo puede contener documentos (de manera predeterminada puede contener fragmentos de documentos). El ejemplo siguiente muestra la definición de una tabla que restringe los ejemplares de la columna XML *Informe* a un documento bien formado (es decir, sólo puede contener un único elemento de nivel superior) que sea válido de acuerdo con el conjunto de esquemas *EsquemaInformes* de la misma base de datos:

create table *InformesViajes*(*id* **int**,
fechaviaje **datetime**,
Informe **xml**(**document** *EsquemaInforme*))

La omisión de la palabra clave **document** permitiría que el contenido guardara varios fragmentos.

Los conjuntos de esquemas también se pueden usar para validar datos de XML previamente no tipificados durante la ejecución mediante coerción de tipos:

cast (*@x* **as xml**(*E1*))

29.11.3 Consultas y actualizaciones sobre el tipo de datos XML

SQL Server 2005 ofrece varias posibilidades de consultas y de modificación basada en XQuery sobre el tipo de datos XML. Estas posibilidades de consulta y de modificación se soportan mediante los métodos definidos para el tipo de datos**xml**. Algunos de estos métodos se describen en el resto de este apartado.

Cada método toma el valor literal de una cadena de caracteres como cadena de caracteres de la consulta y, potencialmente, otros argumentos. El tipo de datos XML (al que se le aplica el método) ofrece el elemento de contexto para las expresiones de ruta y rellena las definiciones de esquemas correspondientes al ámbito correspondiente con toda la información de tipos proporcionada por el conjunto de esquemas de XML asociado (si no se proporciona ningún conjunto, se da por supuesto que los datos de XML no tienen ningún tipo). La implementación de XQuery de SQL Server 2005 tiene tipos estáticos, por lo que soporta la detección temprana de los errores de escritura de las expresiones de ruta, de los errores de tipos y de la no coincidencia de cardinalidades, así como algunas optimizaciones adicionales.

El método *query* toma expresiones de XQuery y devuelve ejemplares de tipos de datos de XML sin tipo (que luego se pueden dirigir a un conjunto de esquemas objetivo si hace falta tipificar los datos). En la terminología de la especificación de XQuery se ha definido el modo de creación como "strip". El ejemplo siguiente muestra una expresión simple de XQuery que resume un elemento Cliente complejo de un documento de informe de viaje que contiene, entre otras informaciones, un nombre, un atributo ID y posibilidades de ventas que se contienen en las notas marcadas del informe de viaje real. El resumen muestra el nombre y la información sobre ventas de los elementos Cliente que tienen posibilidades de ventas.

```
select Informe.query('
    declare namespace c = "urn:ejemplo/cliente";
        for $cli in /c:doc/c:cliente
        where $cli/c:notas//c:posiblesventas
        return
            <id_cliente ="$cli/@id"> {
                $cli/c:nombre,
                $cli/c:notas//c:posiblesventas
            }</cliente>')
from InformesViajes
```

La consulta XQuery anterior se ejecuta para el valor XML guardado en el atributo *doc* de cada fila de la tabla *InformesViajes*. Cada fila del resultado de la consulta de SQL contiene el resultado de ejecutar la consulta de XQuery con los datos de una fila como entrada.

El método *value* toma una expresión de XQuery y el nombre de un tipo de SQL, extrae un solo valor atómico del resultado de la expresión de XQuery y envía su forma léxica al tipo de SQL especificado. Si la expresión de XQuery da lugar a un nodo, el valor tipificado del nodo se extrae de manera implícita como valor atómico que se envía al tipo de SQL (en la terminología de XQuery, el nodo se "atomiza"; el resultado se envía a SQL). Téngase en cuenta que el método *value* lleva a cabo una comprobación estática de tipos comprobando que se devuelve, como máximo, un valor. Dado que el tipo estático de la expresión de ruta suele poder inferir un tipo estático más amplio, aunque la semántica dinámica sólo devuelve un valor, se recomienda usar el predicado posicional para recuperar, como máximo, un valor. El ejemplo siguiente muestra una expresión sencilla de XQuery que cuenta los elementos de pistas de ventas de cada ejemplar del tipo de datos XML y lo devuelve como valor entero de SQL:

```
select Informe.value(
        'declare namespace c = "urn:ejemplo/cliente";
        count(/c:doc/c:cliente//c:posiblesventas)', 'int')
from InformesViajes
```

El método *exist* toma una expresión de XQuery y devuelve un uno si la expresión genera un resultado no vacío y un cero en caso contrario. La expresión siguiente recupera todas las filas de la tabla *InformesViajes*, en la que el documento contiene, como mínimo, un cliente con una posibilidad de ventas:

```
select Informe
from InformesViajes
where 1 = Informe.exist('declare namespace c = "urn:ejemplo/cliente";
                         /c:doc/c:cliente//c:posiblesventas')
```

Hasta ahora, las expresiones siempre se aplican de un ejemplar de tipo de datos XML a un valor resultado por fila relacional. A veces, no obstante, se desea dividir un ejemplar de XML en varios subárboles, de los que cada subárbol se halla en una fila propia, para posterior procesamiento relacional y de XQuery. Esta funcionalidad la aporta el método *nodes*, que toma una expresión de XQuery y genera una tabla que contiene filas con una sola columna, con una fila por cada nodo que la expresión devuelve. Cada fila contiene una referencia a uno de los nodos. Dado que el tipo resultante es un tipo de referencia que no existe en SQL Server fuera del contexto de cada consulta, hay que aplicar uno de los métodos de consulta para materializar el resultado. El método *query* se aplica como cualquier otro tipo de datos de XML, con la diferencia de que el elemento de contexto para las expresiones de ruta no se halla en el documento raíz del tipo de datos de XML, sino en el nodo al que se hace referencia. El ejemplo siguiente extrae para cada pedido de cliente de la columna de XML una fila que contiene la representación de XML del cliente, el nombre del cliente, el identificador del pedido y el identificador del documento que contiene al cliente:

```
select N1.cliente.query('.') as Cliente,
        N1.cliente.value(
            'declare namespace c = "urn:ejemplo/cliente";
            c:nombre[1]', 'nvarchar(20)') as NombreCliente,
        N2."pedido".value('@id', 'int') as IDPedido,
        N1.cliente.value('../@id', 'nvarchar(5)') as IDDoc
from InformesViajes cross apply
        InformesViajes.Informe.nodes(
            'declare namespace c = "urn:ejemplo/cliente";
            /c:doc/c:cliente') as N1(cliente)
        cross apply N1.cliente.nodes(
            'declare namespace c = "urn:ejemplo/cliente";
            ./c:pedido') as N2("pedido")
```

Obsérvese que esto es parecido a la funcionalidad de OpenXML que proporcionan tanto SQL Server 2000 como SQL Server 2005, con la diferencia de que la expresión del método *nodes* está integrada en el procesamiento de XQuery.

Para tener acceso a los datos de SQL del interior de las expresiones de XQuery, SQL Server 2005 proporciona dos funciones denominadas *sql:variable($variable as xs:string)* y *sql:column($columna as xs:string)*. Cada una de estas funciones toma el valor literal de una cadena de caracteres constante para hacer referencia a la variable de SQL o al valor de la columna correlacionada.

Finalmente, el método *modify* proporciona un mecanismo para modificar el valor de XML en el nivel de subárboles. SQL Server 2005 permite insertar subárboles nuevos en ubicaciones concretas de cada árbol, modificar el valor de un elemento de un atributo y borrar subárboles. El ejemplo siguiente borra todos los elementos posiblesventas de los años anteriores al año proporcionado por una variable o por un parámetro de SQL de nombre @año:

```
update InformesViajes
set Informe.modify(
    'declare namespace c = "urn:ejemplo/cliente";
    delete /c:doc/c:cliente//c:posiblesventas[@año < sql:variable("@año")]')
```

29.11.4 Ejecución de expresiones XQuery

Como ya se ha mencionado antes, los datos XML se guardan en una representación binaria interna. Sin embargo, para poder ejecutar las expresiones de XQuery, el tipo de datos XML se transforma internamente en una tabla denominada de nodos. La tabla de nodos interna usa básicamente una fila para

representar cada nodo. Cada nodo recibe un identificador OrdPath como identificador de nodo (los identificadores OrdPath son esquemas numéricos decimales Dewey modificados; véanse las notas bibliográficas para obtener referencias de más información sobre OrdPath). Cada nodo contiene también información de las claves para volver a apuntar a la fila de SQL original a la que pertenece el nodo, información sobre el nombre y el tipo (en forma simbólica), los valores, etc. Dado que OrdPath codifica tanto el orden de los documentos como la información de las jerarquías, la tabla de nodos se agrupa en términos de la información de las claves y de OrdPath, de modo que se pueda conseguir una expresión de ruta o la recomposición de un subárbol con la mera exploración de una tabla.

Todas las expresiones de XQuery y de actualización se traducen luego en un árbol de operadores algebraicos con la tabla de nodos interna; el árbol usa los operadores relacionales habituales y algunos operadores diseñados de manera específica para la algebrización de XQuery. El árbol resultante se injerta en el árbol algebraico de la expresión relacional de modo que, al final, el motor de ejecución de consultas recibe un solo árbol de ejecución que puede optimizar y ejecutar. Para evitar las costosas transformaciones en el momento de la ejecución los usuarios pueden materializar anticipadamente la tabla de nodos usando el índice principal de XML. SQL Server 2005, además, proporciona tres índices secundarios de XML, de modo que la ejecución de las consultas pueda aprovecharse más aún de las estructuras de índices:

- El índice *path* (ruta) ofrece soporte para los tipos sencillos de expresiones de ruta.
- El índice *properties* (propiedades) ofrece soporte para la situación frecuente de las comparaciones entre valores de las propiedades.
- El índice *value* (valor) es idóneo si la consulta usa comodines en las comparaciones.

Véanse las notas bibliográficas para hallar referencias a más información sobre el indexado de XML y el procesado de consultas de XML en SQL Server 2005.

29.12 Service Broker de SQLServer

Service Broker ayuda a los desarrolladores a crear aplicaciones distribuidas de acoplamiento laxo al proporcionar soporte para la mensajería encolada y de confianza en SQL Server. Muchas aplicaciones de bases de datos usan el procesamiento asíncrono para mejorar la posibilidad de redimensionamiento y los tiempos de respuesta de las sesiones interactivas. Un enfoque habitual del procesamiento asíncrono es el uso de tablas de trabajos. En lugar de llevar a cabo todo el trabajo de un proceso corporativo en una sola transacción de la base de datos, la aplicación realiza una modificación que indica que hay presente trabajo destacado y luego inserta un registro del trabajo que hay que llevar a cabo en una tabla de trabajos. Cuando los recursos disponibles lo permiten, la aplicaciones procesa la tabla de trabajos y completa el proceso corporativo. Service Broker forma parte del servidor de bases de datos que soporta directamente este enfoque del desarrollo de aplicaciones. El lenguaje Transact-SQL incluye instrucciones de LDD y de LMD para Service Broker.

Las tecnologías anteriores de encolado de mensajes se concentraban en cada uno de los mensajes. Con Service Broker la unidad básica de comunicación es la *conversación*—un flujo de mensajes persistente, digno de confianza, de tipo dúplex completo. SQL Server garantiza que los mensajes de cada conversación se entregan a la aplicación exactamente una vez y por su orden. Cada conversación forma parte de un *grupo de conversaciones*. Las conversaciones relacionadas se pueden asociar al mismo grupo de conversaciones. Las conversaciones tienen lugar entre dos servicios. Cada *servicio* es un punto final con nombre de una conversación.

Las conversaciones y los mensajes tienen tipos fuertes. Cada mensaje tiene un tipo concreto. SQL Server puede, opcionalmente, validar que los mensajes constituyen XML bien formado, que los mensajes están vacíos o que un mensaje dado está conforme con un esquema de XML. Un *contrato* define los tipos de mensaje admisibles para cada conversación y los participantes en esa conversación que pueden enviar mensajes de ese tipo. SQL Server ofrece un contrato y un tipo de mensajes predeterminados para las aplicaciones que sólo necesitan un flujo digno de confianza.

SQL Server guarda los mensajes en tablas internas. Estas tablas no son accesibles directamente; en vez de esto, SQL Server muestra las *colas* como vistas de esas tablas internas. Las aplicaciones reciben mensajes de las colas. La operación receive (recibir) devuelve uno o varios mensajes del mismo grupo de conversaciones. Al controlar el acceso a la tabla subyacente, SQL Server puede hacer que se cumplan de manera eficiente la ordenación de los mensajes, la correlación de los mensajes relacionados y los bloqueos. Como las colas son tablas internas, las colas no necesitan ningún tratamiento especial para las copias de seguridad, las restauraciones, los relevos ni las copias exactas de las bases de datos. Tanto las tablas de las aplicaciones como los mensajes encolados asociados se incluyen en las copias de seguridad, se restauran y se relevan con la base de datos. Las conversaciones de Broker que se producen en las bases de datos con copias exactas continúan cuando parten al completarse el relevo de la copia exacta —aunque la conversación se produjera entre dos servicios ubicados en bases de datos diferentes.

La granularidad de los bloqueos de las operaciones de Service Broker es el grupo de conversaciones, en vez de una conversación concreta o los diferentes mensajes. Al hacer que se cumplan los bloqueos de los grupos de conversación, Service Broker ayuda de manera automática a que las aplicaciones eviten problemas de concurrencia al procesar los mensajes. Cuando una cola contiene varias conversaciones, SQL Server garantiza que sólo un lector de colas pueda procesar los mensajes que pertenecen a un grupo de conversación dado a la vez. Esto elimina la necesidad de que las propias aplicaciones incluyan lógica para evitar los interbloqueos—una fuente de errores habitual en muchas aplicaciones de mensajería. Otro efecto lateral positivo de esta semántica de bloqueos es que las aplicaciones pueden decidir usar el grupo de conversaciones como clave para guardar y recuperar el estado de las aplicaciones. Estas ventajas del modelo de programación no son más que dos ejemplos de las ventajas que se obtienen de la decisión de formalizar la conversación como primitiva de la comunicación en lugar de la primitiva de los mensajes atómicos usada en los sistemas de encolado de mensajes tradicionales.

SQL Server puede activar de manera automática los procedimientos almacenados cuando una cola contiene mensajes que hay que procesar. Para adecuar el número de procesos almacenados en ejecución al tráfico entrante, la lógica de activación supervisa la cola para ver si hay trabajo útil para otro lector de colas. SQL Server considera tanto la velocidad a la que los lectores existentes reciben los mensajes como el número de grupos de conversaciones disponibles para decidir si debe iniciar otro lector de colas. El procedimiento almacenado que se debe activar, el contexto de seguridad del procedimiento almacenado y el número máximo de ejemplares que se pueden iniciar se configuran para una cola dada. SQL Server también muestra un evento que pueden usar las aplicaciones externas para iniciar los lectores de colas.

Como extensión lógica de la mensajería asíncrona dentro del ejemplar, Service Broker también ofrece mensajería digna de confianza entre ejemplares de SQL Server para permitir que los desarrolladores creen aplicaciones distribuidas con facilidad. Las conversaciones pueden tener lugar dentro de un solo ejemplar de SQL Server o entre dos ejemplares de SQL Server. Las conversaciones locales y las remotas usan el mismo modelo de programación.

La seguridad y el encaminamiento se configuran de manera declarativa, sin necesidad de modificaciones de los lectores de colas. SQL Server usa *rutas* para asignar un nombre de servicio a la dirección de red del otro participante en la conversación. SQL Server también puede llevar a cabo entregas de mensajes y equilibrios de carga sencillos para las conversaciones. SQL Server ofrece entregas dignas de confianza, sólo una vez y en su orden, independientemente del número de ejemplares por los que tengan que viajar los mensajes. Las conversaciones que abarcan varios ejemplares de SQL Server se pueden proteger tanto en el nivel de la red (de punto a punto) como en el nivel de la conversación (de extremo a extremo). Cuando se usa la seguridad de extremo a extremo, el contenido de los mensajes permanece cifrado hasta que llega a su destino final, mientras que las cabeceras están disponibles para cada ejemplar de SQL Server por el que viaje cada mensaje. Los permisos estándar de SQL Server se aplican dentro de cada ejemplar. El cifrado se produce cuando los mensajes dejan un ejemplar.

SQL Server usa un protocolo binario para el envío de mensajes entre ejemplares. El protocolo fragmenta los mensajes de gran tamaño y permite que se intercalen los fragmentos de varios mensajes. La fragmentación permite que SQL Server transmita rápidamente los mensajes más pequeños, aun en el caso de que se esté transmitiendo un mensaje de gran tamaño. El protocolo binario no usa las transacciones distribuidas ni el compromiso de dos fases. En vez de eso, exige que un receptor reconozca los fragmentos de los mensajes. SQL Server se limita a volver a intentar enviar periódicamente los fragmentos de los mensajes hasta que el receptor los reconoce. Los reconocimientos se incluyen muy frecuentemente como

parte de la cabecera de los mensajes devueltos, aunque se utilicen mensajes de vuelta exclusivos si no se dispone de ningún otro.

29.13 Almacenes de datos e inteligencia de negocio

El componente de almacenamiento de datos e inteligencia de negocio de SQL Server contiene tres componentes:

- Los servicios de integración de SQL Server (SQL Server Integration Services, SSIS), antes conocidos como Servicios de transformación de datos (Data Transformation Services, DTS), que ofrecen los medios para integrar datos de varios orígenes, llevan a cabo transformaciones relacionadas con la limpieza de los datos y su transformación a un formato común y la carga de los datos en el sistema de bases de datos.
- Los servicios de análisis de SQL Server (SQL Server Analysis Services, SSAS), que proporcionan las capacidades OLAP y de minería de datos.
- Los servicios de informes de SLQ Server (SQL Server Reporting Services, SSRS).

Los servicios de integración, los de análisis y los de informes se implementan en servidores diferentes y se pueden instalar de manera independiente en la misma máquina o en máquinas diferentes. Pueden conectar con gran variedad de orígenes de datos, como los archivos planos, las hojas de cálculo o gran variedad de sistemas de bases de datos relacionales, mediante conectores nativos, OLE-DB o controladores ODBC.

En conjunto ofrecen una solución integrada para la extracción, transformación y carga de los datos, el modelado y la adición de capacidad analítica posteriores a los datos y, finalmente, la creación y distribución de informes de los datos. Los diversos componentes del servidor de análisis pueden integrarse entre sí y aprovechar las capacidades de los demás. A continuación se muestran unas cuantas situaciones que aprovechan diferentes combinaciones de componentes:

- Creación de un paquete SSIS que limpie los datos, usando patrones generados por la minería de datos de SSAS.
- El uso de SSIS para cargar datos en un cubo de SSAS, su procesamiento y la ejecución de informes relativos al cubo de SSAS.
- Creación de un informe de SSRS para publicar los hallazgos de un modelo de minería o los datos contenidos en un componente OLAP de SSAS.

Los apartados siguientes aportan una visión general de las posibilidades y de la arquitectura de estos componentes de los servidores.

29.13.1 SQL Server Integration Services

SQL Server 2005 Integration Services (SSIS) de Microsoft es una solución corporativa para la transformación y la integración de datos que se puede usar para extraer, transformar, agregar y consolidar datos de diferentes orígenes y trasladarlos a uno o a varios destinos. Se puede usar SSIS para llevar a cabo las tareas siguientes:

- Mezclar datos de almacenes de datos heterogéneos
- Refrescar los datos de los almacenes y puestos de datos.
- Limpiar los datos antes de cargarlos en sus destinos.
- Realizar cargas masivas de datos en bases de datos de procesamiento de transacciones en línea (Online Transaction Processing, OLTP) y de procesamiento analítico en línea (Online Analytical Processing, OLAP).
- Enviar notificaciones.

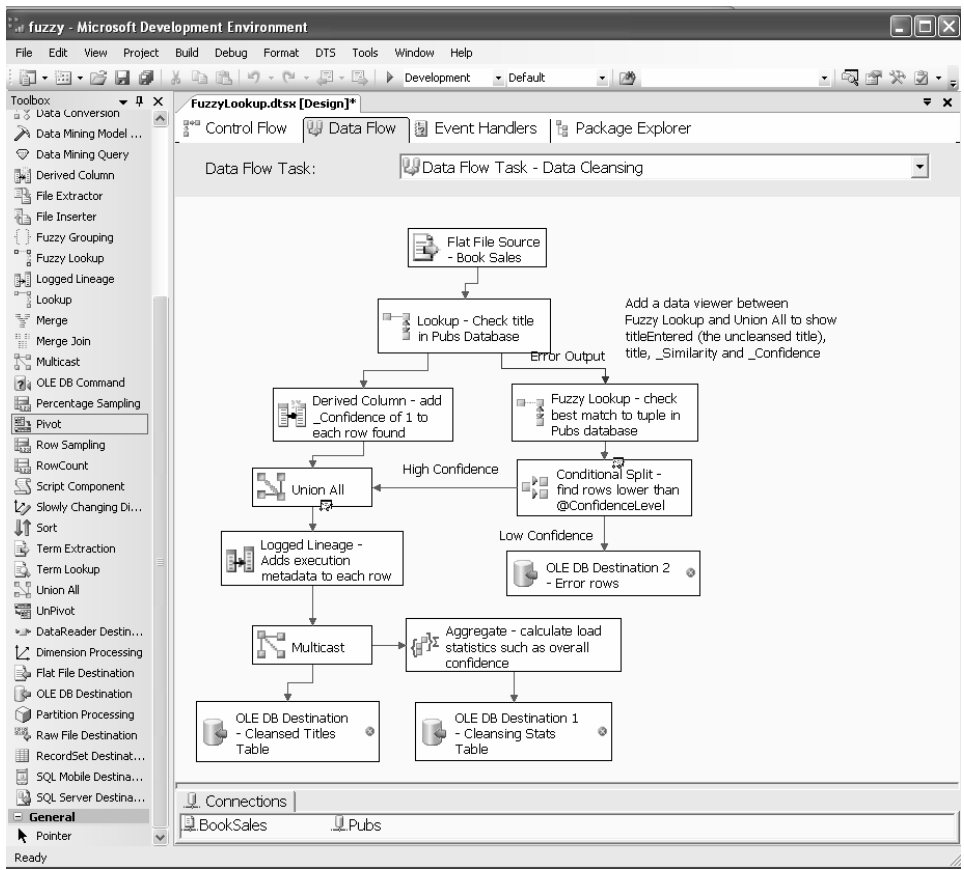


Figura 29.7 Carga de datos usando búsquedas difusas.

- Incluir inteligencia de negocio en el proceso de transformación de los datos.
- Automatizar las funciones administrativas.

SSIS ofrece un conjunto completo de servicios, herramientas gráficas, objetos programables y APIs para estas tareas. Todo ello ofrece la posibilidad de crear soluciones de transformación de los datos de gran tamaño, robustas y complejas sin necesidad de programación personalizada. No obstante, se dispone de APIs y de objetos programables cuando son necesarios para crear elementos personalizados o para integrar las posibilidades de transformación de los datos en aplicaciones personalizadas.

El motor de flujo de datos de SSIS ofrece las memorias intermedias ubicadas en la memoria que trasladan los datos desde su origen hasta su destino y llama a los adaptadores de orígenes que extraen los datos de los archivos y de las bases de datos relacionales. El motor también proporciona las transformaciones que modifican los datos y los adaptadores de destinos que cargan los datos en los almacenes de datos. La eliminación de duplicados basada en la coincidencia difusa (aproximada) es un ejemplo de transformación proporcionada por SSIS. Los usuarios pueden programar sus propias transformaciones si hace falta. La Figura 29.7 muestra un ejemplo de combinación de varias transformaciones para la limpieza y carga de información sobre ventas de libros; los títulos de los libros de los datos de venta se comparan con la base de datos de publicaciones y, en caso de que no coincidan, se lleva a cabo una búsqueda difusa para manejar los títulos con errores sin importancia (como los de escritura). La información sobre la confianza y la correlación de los datos se guarda con los datos limpios.

29.13.2 SQL Server Analysis Services

El componente de servicios de análisis proporciona la funcionalidad de procesamiento analítico en línea (On−Line Analytical Processing, OLAP) y de minería de datos para las aplicaciones de inteligencia de

negocio. Los servicios de análisis soportan una arquitectura de cliente reducida. El motor de cálculo se halla en el servidor, por lo que las consultas se resuelven allí, lo que evita la necesidad de transferir grandes cantidades de datos entre el cliente y el servidor.

29.13.2.1 SQL Server Analysis Services: OLAP

Los servicios de análisis (Analysis Services) de SQL Server 2005 introducen un modelo dimensional unificado (Unified Dimensional Model, UDM) que salva el hueco entre los informes relacionales tradicionales y el análisis OLAP ad hoc. El papel del modelo dimensional unificado es ofrecer un puente entre el usuario y los orígenes de datos. Los UDMs se crean sobre uno o varios orígenes físicos de datos y luego el usuario final formula consultas al UDM, usando gran variedad de herramientas de cliente, como Microsoft Excel.

Además de ser una capa de modelado dimensional de los esquemas DataSource, el UDM proporciona un rico entorno para la definición de lógica, reglas y definiciones semánticas corporativas potentes pero exhaustivas. Los usuarios pueden examinar y generar informes de los datos del UDM en su idioma nativo (por ejemplo, francés o hindi) mediante la definición de la traducción al lenguaje local del catálogo de metadatos y de los datos dimensionales. El servidor de análisis define dimensiones temporales complejas (fiscal, informes, fabricación, etc.) y permite la definición de lógica corporativa multidimensional potente (crecimiento interanual, en el año en curso) mediante el lenguaje de expresiones multidimensionales (MDX). El UDM permite que los usuarios definan perspectivas orientadas al negocio, cada una de las cuales sólo presenta el subconjunto concreto del modelo (medidas, dimensiones, atributos, reglas corporativas, etc.) que es relevante para un grupo determinado de usuarios. Las empresas suelen definir indicadores de rendimiento cruciales (key performance indicators, KPIs), que son métricas importantes que se usan para medir la prosperidad del negocio. Ejemplos de esos KPIs son las ventas, los ingresos por empleado y la tasa de retención de los clientes. El UDM permite definir esos KPIs, lo que permite una agrupación y presentación de los datos mucho más comprensibles.

29.13.2.2 SQL Server Analysis Services: minería de datos

SQL Server 2005 proporciona gran variedad de técnicas de minería, con una rica interfaz gráfica para ver los resultados de la minería. Algunos de los algoritmos soportados son:

- Las reglas de asociación (útiles para las aplicaciones de ventas cruzadas).
- Las técnicas de clasificación y predicción, como los árboles de decisión, los árboles de regresión, las redes neuronales y la lógica de Bayes ingenua.
- La predicción mediante series temporales.
- Las técnicas de agrupación como la maximización de esperanzas y las medias k (acopladas con técnicas de agrupación de secuencias).

SQL Server También soporta las *Extensiones de minería de datos* (*Data–Mining Extensions*, DMX) de SQL. DMX es el lenguaje usado para interactuar con los modelos de minería de datos, igual que SQL se usa para interactuar con las tablas y con las vistas. Con DMX se pueden crear, entrenar y almacenar modelos en bases de datos de los servicios de análisis. El modelo se puede examinar luego para buscar patrones o, mediante una sintaxis especial de **reunión de predicciones** (**prediction join)**, aplicada a los datos nuevos, llevar a cabo predicciones. El lenguaje DMX soporta funciones y estructuras para determinar con facilidad las clases predichas, junto con su confianza, predecir una lista de elementos asociados, como en los motores de recomendaciones, o, incluso, devolver información y hechos que apoyen las predicciones. La minería de datos en SQL Server 2005 se puede usar con datos guardados en bases de datos relacionales o en orígenes de datos multidimensionales. Se soportan también otros orígenes de datos mediante tareas y transformaciones especializadas, lo que permite la minería de datos directamente en el cauce de datos operativo de los servicios de integración. Los resultados de la minería de datos se pueden exponer en controles gráficos, dimensiones especiales de minería de datos de los cubos OLAP o, simplemente, en los informes de los servicios de informes.

29.13.3 SQL Server Reporting Services

Los servicios de informes (Reporting Services) son una nueva plataforma de informes basada en servidor que se puede usar para crear y gestionar informes tabulares, matriciales, gráficos y de formato libre que contengan datos de orígenes de datos relacionales y multidimensionales. Los informes creados se pueden ver y gestionar mediante conexiones basadas en Web. Los informes matriciales pueden resumir datos de revisiones de alto nivel, mientras que ofrecen detalles de apoyo en los informes de minería. Los informes parametrizados se pueden usar para filtrar datos en términos de los valores que se proporcionen en el momento de la ejecución. Los usuarios pueden escoger entre gran variedad de formatos de vistas para presentar los informes sobre la marcha en los formatos preferidos para el tratamiento de datos o para la impresión. También se dispone de una API para extender o integrar capacidades de elaboración de informes en soluciones personalizadas. Los informes basados en los servidores ofrecen una manera de centralizar el almacenamiento y la gestión de los informes, definir políticas y proteger el acceso a los informes y a las carpetas, controlar la manera en que se procesan y distribuyen los informes y normalizar el modo en que se usan los informes en la empresa.

Notas bibliográficas

En www.microsoft.com/Downloads/Release.asp?ReleaseID=25503 se dispone de información detallada sobre el uso de sistemas certificados C2 con SQL Server.

El entorno de optimización de SQL Server se basa en el prototipo de optimizador Cascades, que propuso Graefe [1995]. Simmen et al. [1996] estudian el esquema para la reducción de las columnas de agrupación. Galindo-Legaria y Joshi [2001] presentan la gran variedad de estrategias de ejecución que SQL Server considera durante la optimización basada en costes. Chaudhuri et al. [1999] estudian información adicional sobre los aspectos de ajuste automático de SQL Server. Chaudhuri y Shim [1994] y Yan y Larson [1995] estudian la agregación parcial.

Chatziantoniou y Ross [1997] y Galindo-Legaria y Joshi [2001] propusieron la alternativa usada por SQL Server para las consultas de SQL que necesitan una autorreunión. Según este esquema, el optimizador detecta el patrón y considera la ejecución segmento por segmento. Pellenkoft et al. [1997] estudian el esquema de optimización para la generación del espacio completo de búsqueda mediante un conjunto de transformaciones que son completas, locales y no redundantes. Graefe et al. [1998] ofrecen la discusión relativa a las operaciones de asociación que soportan la agregación y la reunión básicas, con varias optimizaciones, extensiones y ajustes dinámicos del sesgo de los datos. Graefe et al. [1998] presentan la idea de reunir los índices con el único propósito de ensamblar una fila con el conjunto de columnas necesario para la consulta. Arguyen que esto a veces resulta más rápido que explorar la tabla base.

Blakeley [1996] y Blakeley y Pizzo [2001] estudian la comunicación con el motor de almacenamiento mediante OLE-DB. Blakeley et al. [2005] detallan la implementación de las capacidades de consulta distribuida y heterogénea de SQL Server. Acheson et al. [2004] proporcionan detalles sobre la integración de CLR de .NET dentro del proceso de SQL Server.

La norma SQL:2003 se define en SQL/XML [2004]. Rys [2001] proporciona más detalles sobre la funcionalidad XML de SQL Server 2000 XML. Rys [2004] ofrece una visión general de las extensiones de la agregación **for xml**. Para obtener más información sobre las capacidades XML que se pueden usar en el lado del cliente o dentro de CLR, consúltese la colección de libros blancos de MSDN: XML Developer Center (Centro de desarrollo XML de MSDN). El modelo de datos XQuery 1.0/XPath 2.0 se define en Walsh et al. [2004]. Rys [2003] proporciona una visión general de las técnicas de implementación de XQuery en el contexto de las bases de datos relacionales. El esquema de numeración OrdPath se describe en O'Neil et al. [2004]; Pal et al. [2004] y Baras et al. [2005] ofrecen más información sobre el indexado de XML, la algebrización de XQuery y la optimización en SQL Server 2005.

Bibliografía

[**Abiteboul et al. 1995**] S. Abiteboul, R. Hull y V. Vianu, *Foundations of Databases*, Addison Wesley (1995).

[**Abiteboul et al. 2003**] S. Abiteboul, R. Agrawal, P. A. Bernstein, M. J. Carey et al. "The Lowell Database Research Self Assessment". CoRR cs.DB/0310006 (2003).

[**Acharya et al. 1995**] S. Acharya, R. Alonso, M. Franklin y S. Zdonik, "Broadcast Disks: Data Management for Asymmetric Communication Environments", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), páginas 19 – 210. También aparece en Imielinski y Korth [1996], Capítulo 12.

[**Acheson et al. 2004**] A. Acheson, M. Bendixen, J. A. Blakeley, I. P. Carlin, E. Ersan, J. Fang, X. Jiang, C. Kleinerman, B. Rathakrishnan, G. Schaller, B. Sezgin, R. Venkatesh y H. Zhang, "Hosting the .NET Runtime in Microsoft SQL Server", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004), páginas 860–865.

[**Adali et al. 1996**] S. Adali, K. S. Candan, Y. Papakonstantinou y V. S. Subrahmanian, "Query Caching and Optimization in Distributed Mediator Systems", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996), páginas 137–148.

[**Agarwal et al. 1996**] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan y S. Sarawagi, "On the Computation of Multidimensional Attributes", *Proc. of the International Conf. on Very Large Databases*, Bombay, India (1996), páginas 506–521.

[**Agrawal et al. 1992**] R. Agrawal, S. P. Ghosh, T. Imielinski, B. R. Iyer y A. N. Swami, "An Interval Classifier for Database Mining Applications", *Proc. of the International Conf. on Very Large Databases* (1992), páginas 560–573.

[**Agrawal et al. 1993a**] R. Agrawal, T. Imielinski y A. Swami, "Mining Association Rules between Sets of Items in Large Databases", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993).

[**Agrawal et al. 1993b**] R. Agrawal, T. Imielinski y A. N. Swami, "Database Mining: A Performance Perspective", *IEEE Transactions on Knowledge and Data Engineering*, volumen 5, número 6 (1993), páginas 914–925.

[**Agrawal et al. 2000**] S. Agrawal, S. Chaudhuri y V. R. Narasayya, "Automated Selection of Materialized Views and Indexes in SQL Databases", *Proc. of the International Conf. on Very Large Databases* (2000), páginas 496–505.

[**Agrawal et al. 2002**] S. Agrawal, S. Chaudhuri y G. Das, "DBXplorer: A System for Keyword-Based Search over Relational Databases", *Proc. of the International Conf. on Data Engineering* (2002).

[**Agrawal et al. 2004**] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya y M. Syamala, "Database Tuning Advisor for Microsoft SQL Server 2005", *Proc. of the International Conf. on Very Large Databases* (2004).

[**Agrawal y Srikant 1994**] R. Agrawal y R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases", *Proc. of the International Conf. on Very Large Databases* (1994), páginas 487–499.

[**Aho et al. 1979a**] A. V. Aho, C. Beeri y J. D. Ullman, "The Theory of Joins in Relational Databases", *ACM Transactions on Database Systems*, volumen 4, número 3 (1979), páginas 297–314.

[**Aho et al. 1979b**] V. Aho, Y. Sagiv y J. D. Ullman, "Equivalences among Relational Expressions", *SIAM Journal of Computing*, volumen 8, número 2 (1979), páginas 218–246.

[**Aho et al. 1986**] A. V. Aho, R. Sethi y J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley (1986).

[**Ailamaki et al. 2001**] A. Ailamaki, D. J. DeWitt, M. D. Hill y M. Skounakis, "Weaving Relations for Cache Performance", *Proc. of the International Conf. on Very Large Databases* (2001), páginas 169–180.

[**Alonso y Korth 1993**] R. Alonso y H. F. Korth, "Database System Issues in Nomadic Computing", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993), páginas 388–392.

[**Amer-Yahia et al. 2004**] S. Amer-Yahia, C. Botev y J. Shanmugasundaram, "TeXQuery: A Full-Text Search Extension to XQuery", *Proc. of the International World Wide Web Conf.* (2004).

[**Anderson et al. 1992**] D. P. Anderson, Y. Osawa y R. Govindan, "A File System for Continuous Media", *ACM Transactions on Database Systems*, volumen 10, número 4 (1992), páginas 311–337.

[**Anderson et al. 1998**] T. Anderson, Y. Breitbart, H. F. Korth y A. Wool, "Replication, Consistency and Practicality: Are These Mutually Exclusive?", *Proc. of the ACM SIGMOD Conf. on Management of Data*, Seattle, WA (1998).

[**ANSI 1986**] *American National Standard for Information Systems: Database Language SQL*. American National Standards Institute. FDT, ANSI X3,135–1986 (1986).

[**ANSI 1989**] *Database Language SQL with Integrity Enhancement, ANSI X3, 135–1989*. American National Standards Institute, New York. Also availabe as ISO/IEC Document 9075:1989 (1989).

[**ANSI 1992**] *Database Language SQL, ANSI X3,135–1992*. American National Standards Institute, New York. Also available as ISO/IEC Document 9075:1992 (1992).

[**Antoshenkov 1995**] G. Antoshenkov, "Byte-aligned Bitmap Compression (poster abstract)", *IEEE Data Compression Conf.* (1995).

[**Appelt y Israel 1999**] D. E. Appelt y D. J. Israel, "Introduction to Information Extraction Technology", *IJCAI* (1999). Tutorial presented at IJCAI-99, available at http://www.ai.sri.com/ãppelt/ie-tutorial/IJCAI99.pdf.

[**Apt y Pugin 1987**] K. R. Apt y J. M. Pugin, "Maintenance of Stratified Database Viewed as a Belief Revision System", *Proc. of the ACM Symposium on Principles of Database Systems* (1987), páginas 136–145.

[**Armstrong 1974**] W. W. Armstrong, "Dependency Structures of Data Base Relationships", *Proc. of the 1974 IFIP Congress* (1974), páginas 580–583.

[**Astrahan et al. 1976**] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade y V. Watson, "System R, A Relational Approach to Data Base Management", *ACM Transactions on Database Systems*, volumen 1, número 2 (1976), páginas 97–137.

[**Atreya et al. 2002**] M. Atreya, B. Hammond, S. Paine, P. Starrett y S. Wu, *Digital Signatures*, RSA Press (2002).

[**Atzeni y Antonellis 1993**] P. Atzeni y V. D. Antonellis, *Relational Database Theory*, Benjamin Cummings, Menlo Park (1993).

[**Baeza-Yates y Ribeiro-Neto 1999**] R. Baeza-Yates y B. Ribeiro-Neto, *Modern Information Retrieval*, Addison Wesley (1999).

[**Bancilhon et al. 1989**] F. Bancilhon, S. Cluet y C. Delobel, "A Query Language for the O_2 Object-Oriented Database", *Proc. of the Second Workshop on Database Programming Languages* (1989).

[**Bancilhon y Buneman 1990**] F. Bancilhon y P. Buneman, *Advances in Database Programming Languages*, ACM Press, New York (1990).

[**Banerjee et al. 2000**] S. Banerjee, V. Krishnamurthy, M. Krishnaprasad y R. Murthy, "Oracle 8i - The XML Enabled Data Management System", *Proc. of the International Conf. on Data Engineering* (2000), páginas 561–568.

[**Baras et al. 2005**] A. Baras, D. Churin, I. Cseri, T. Grabs, E. Kogan, S. Pal, M. Rys y O. Seeliger. "Implementing XQuery in a Relational Database System". to be published (2005).

[**Barbará y Imielinski 1994**] D. Barbará y T. Imielinski, "Sleepers and Workaholics: Caching Strategies in Mobile Environments", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1994), páginas 1–12.

[**Baru et al. 1995**] C. Baru et al., "DB2 Parallel Edition", *IBM Systems Journal*, volumen 34, número 2 (1995), páginas 292–322.

[**Bassiouni 1988**] M. Bassiouni, "Single-site and Distributed Optimistic Protocols for Concurrency Control", *IEEE Transactions on Software Engineering*, volumen SE-14, número 8 (1988), páginas 1071–1080.

[**Batini et al. 1992**] C. Batini, S. Ceri y S. Navathe, *Database Design: An Entity-Relationship Approach*, Benjamin Cummings, Redwood City (1992).

[**Bayer 1972**] R. Bayer, "Symmetric Binary B-trees: Data Structure and Maintenance Algorithms", *Acta Informatica*, volumen 1, número 4 (1972), páginas 290–306.

[**Bayer et al. 1978**] R. Bayer, R. M. Graham y G. Seegmuller, editores, *Operating Systems: An Advanced Course*, Springer Verlag (1978).

[**Bayer et al. 1980**] R. Bayer, H. Heller y A. Reiser, "Parallelism and Recovery in Database Systems", *ACM Transactions on Database Systems*, volumen 5, número 2 (1980), páginas 139–156.

[**Bayer y McCreight 1972**] R. Bayer y E. M. McCreight, "Organization and Maintenance of Large Ordered Indices", *Acta Informatica*, volumen 1, número 3 (1972), páginas 173–189.

[**Bayer y Schkolnick 1977**] R. Bayer y M. Schkolnick, "Concurrency of Operating on B-trees", *Acta Informatica*, volumen 9, número 1 (1977), páginas 1–21.

[**Bayer y Unterauer 1977**] R. Bayer y K. Unterauer, "Prefix B-trees", *ACM Transactions on Database Systems*, volumen 2, número 1 (1977), páginas 11–26.

[**Beckmann et al. 1990**] N. Beckmann, H. P. Kriegel, R. Schneider y B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), páginas 322–331.

[**Beeri et al. 1977**] C. Beeri, R. Fagin y J. H. Howard, "A Complete Axiomatization for Functional and Multivalued Dependencies", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1977), páginas 47–61.

[**Bello et al. 1998**] R. G. Bello, K. Dias, A. Downing, J. F. Jr., W. D. Norcott, H. Sun, A. Witkowski y M. Ziauddin, "Materialized Views in Oracle", *Proc. of the International Conf. on Very Large Databases* (1998), páginas 659–664.

[**Bentley 1975**] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching", *Communications of the ACM*, volumen 18, número 9 (1975), páginas 509–517.

[**Berenson et al. 1995**] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil y P. O'Neil, "A Critique of ANSI SQL Isolation Levels", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), páginas 1–10.

[**Bernstein et al. 1983**] P. A. Bernstein, N. Goodman y M. Y. Lai, "Analyzing Concurrency Control when User and System Operations Differ", *IEEE Transactions on Software Engineering*, volumen SE-9, número 3 (1983), páginas 233–239.

[**Bernstein et al. 1987**] A. Bernstein, V. Hadzilacos y N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley (1987).

[**Bernstein et al. 1998**] P. Bernstein, M. Brodie, S. Ceri, D. DeWitt, M. Franklin, H. Garcia-Molina, J. Gray, J. Held, J. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. Naughton, H. Pirahesh, M. Stonebraker y J. Ullman, "The Asilomar Report on Database Research", *ACM SIGMOD Record*, volumen 27, número 4 (1998).

[**Bernstein y Goodman 1980**] P. A. Bernstein y N. Goodman, "Timestamp-based Algorithms for Concurrency Control in Distributed Database Systems", *Proc. of the International Conf. on Very Large Databases* (1980), páginas 285–300.

[**Bernstein y Goodman 1981**] P. A. Bernstein y N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Survey*, volumen 13, número 2 (1981), páginas 185–221.

[**Bernstein y Goodman 1982**] P. A. Bernstein y N. Goodman, "A Sophisticate's Introduction to Distributed Database Concurrency Control", *Proc. of the International Conf. on Very Large Databases* (1982), páginas 62–76.

[**Bernstein y Newcomer 1997**] P. A. Bernstein y E. Newcomer, *Principles of Transaction Processing*, Morgan Kaufmann (1997).

[**Berson et al. 1995**] S. Berson, L. Golubchik y R. R. Muntz, "Fault Tolerant Design of Multimedia Servers", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), páginas 364–375.

[**Bhalotia et al. 2002**] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti y S. Sudarshan, "Keyword Searching and Browsing in Databases using BANKS", *Proc. of the International Conf. on Data Engineering* (2002).

[**Bharat y Henzinger 1998**] K. Bharat y M. R. Henzinger, "Improved Algorithms for Topic Distillation in a Hyperlinked Environment", *Proc. of the ACM SIGIR Conf. on Research and Development in Information Retrieval* (1998), páginas 104–111.

[**Bhattacharjee et al. 2003**] B. Bhattacharjee, S. Padmanabhan, T. Malkemus, T. Lai, L. Cranston y M. Huras, "Efficient Query Processing for Multi-Dimensionally Clustered Tables in DB2", *Proc. of the International Conf. on Very Large Databases* (2003), páginas 963–974.

[**Biliris y Orenstein 1994**] A. Biliris y J. Orenstein. "Object Storage Management Architectures", In *Dogac et al. [1994]*, páginas 185–200 (1994).

[**Biskup et al. 1979**] J. Biskup, U. Dayal y P. A. Bernstein, "Synthesizing Independent Database Schemas", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1979), páginas 143–152.

[**Bitton et al. 1983**] D. Bitton, D. J. DeWitt y C. Turbyfill, "Benchmarking Database Systems: A Systematic Approach", *Proc. of the International Conf. on Very Large Databases* (1983).

[**Bitton y Gray 1988**] D. Bitton y J. N. Gray, "Disk Shadowing", *Proc. of the International Conf. on Very Large Databases* (1988), páginas 331–338.

[**Bjork 1973**] L. A. Bjork, "Recovery Scenario for a DB/DC System", *Proc. of the ACM Annual Conf.* (1973), páginas 142–146.

[**Blakeley 1996**] J. A. Blakeley, "Data Access for the Masses through OLE DB", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996), páginas 161–172.

[**Blakeley et al. 1986**] J. A. Blakeley, P. Larson y F. W. Tompa, "Efficiently Updating Materialized Views", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1986), páginas 61–71.

[**Blakeley et al. 1989**] J. Blakeley, N. Coburn y P. Larson, "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates", *ACM Transactions on Database Systems*, volumen 14, número 3 (1989), páginas 369–400.

[**Blakeley et al. 2005**] J. A. Blakeley, C. Cunningham, N. Ellis, B. Rathakrishnan y M.-C. Wu, "Distributed/Heterogeneous Query Processing in Microsoft SQL Server", *Proc. of the International Conf. on Data Engineering* (2005).

[**Blakeley y Pizzo 2001**] J. A. Blakeley y M. Pizzo, "Enabling Component Databases with OLE DB", K. R. Dittrich y A. Geppert, editores, *Component Database Systems*, Morgan Kaufmann Publishers (2001), páginas 139–173.

[**Blasgen y Eswaran 1976**] M. W. Blasgen y K. P. Eswaran, "On the Evaluation of Queries in a Relational Database System", *IBM Systems Journal*, volumen 16, (1976), páginas 363–377.

[**Boral et al. 1990**] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith y P. Valduriez, "Prototyping Bubba, a Highly Parallel Database System", *IEEE Transactions on Knowledge and Data Engineering*, volumen 2, número 1 (1990).

[**Boyce et al. 1975**] R. Boyce, D. D. Chamberlin, W. F. King y M. Hammer, "Specifying Queries as Relational Expressions", *Communications of the ACM*, volumen 18, número 11 (1975), páginas 621–628.

[**Breese et al. 1998**] J. Breese, D. Heckerman y C. Kadie, "Empirical Analysis of Predictive Algorithms for Collaborative Filtering", *Procs. Conf. on Uncertainty in Artificial Intelligence*, Morgan Kaufmann (1998).

[**Breitbart et al. 1999a**] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri y A. Silberschatz, "Update Propagation Protocols For Replicated Databases", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1999), páginas 97–108.

[**Breitbart et al. 1999b**] Y. Breitbart, H. Korth, A. Silberschatz y S. Sudarshan, "Distributed Databases", *In Encyclopedia of Electrical and Electronics Engineering*, John Wiley and Sons (1999).

[**Brin y Page 1998**] S. Brin y L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine", *Proc. of the International World Wide Web Conf.* (1998).

[**Brinkhoff et al. 1993**] T. Brinkhoff, H.-P. Kriegel y B. Seeger, "Efficient Processing of Spatial Joins Using R-trees", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993), páginas 237–246.

[**Bruno et al. 2002**] N. Bruno, S. Chaudhuri y L. Gravano, "Top-k Selection Queries Over Relational Databases: Mapping Strategies and Performance Evaluation", *ACM Transactions on Database Systems*, volumen 27, número 2 (2002), páginas 153–187.

[**Buckley y Silberschatz 1983**] G. Buckley y A. Silberschatz, "Obtaining Progressive Protocols for a Simple Multiversion Database Model", *Proc. of the International Conf. on Very Large Databases* (1983), páginas 74–81.

[**Buckley y Silberschatz 1984**] G. Buckley y A. Silberschatz, "Concurrency Control in Graph Protocols by Using Edge Locks", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1984), páginas 45–50.

[**Buckley y Silberschatz 1985**] G. Buckley y A. Silberschatz, "Beyond Two-Phase Locking", *Journal of the ACM*, volumen 32, número 2 (1985), páginas 314–326.

[**Bulmer 1979**] M. G. Bulmer, *Principles of Statistics*, Dover Publications (1979).

[**Burkhard 1976**] W. A. Burkhard, "Hashing and Trie Algorithms for Partial Match Retrieval", *ACM Transactions on Database Systems*, volumen 1, número 2 (1976), páginas 175–187.

[**Burkhard 1979**] W. A. Burkhard, "Partial-match Hash Coding: Benefits of Redundancy", *ACM Transactions on Database Systems*, volumen 4, número 2 (1979), páginas 228–239.

[**Cannan y Otten 1993**] S. Cannan y G. Otten, *SQL—The Standard Handbook*, McGraw-Hill (1993).

[**Carey 1983**] M. J. Carey, "Granularity Hierarchies in Concurrency Control", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1983), páginas 156–165.

[**Carey et al. 1991**] M. Carey, M. Franklin, M. Livny y E. Shekita, "Data Caching Tradeoffs in Client-Server DBMS Architectures", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1991).

[**Carey et al. 1993**] M. J. Carey, D. DeWitt y J. Naughton, "The OO7 Benchmark", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993).

[**Carey et al. 1999**] M. J. Carey, D. D. Chamberlin, S. Narayanan, B. Vance, D. Doole, S. Rielau, R. Swagerman y N. Mattos, "O-O, What Have They Done to DB2?", *Proc. of the International Conf. on Very Large Databases* (1999), páginas 542–553.

[**Carey y Kossmann 1998**] M. J. Carey y D. Kossmann, "Reducing the Braking Distance of an SQL Query Engine", *Proc. of the International Conf. on Very Large Databases* (1998), páginas 158–169.

[**Cattell 2000**] R. Cattell, editor, *The Object Database Standard: ODMG 3.0*, Morgan Kaufmann (2000).

[**Cattell y Skeen 1992**] R. Cattell y J. Skeen, "Object Operations Benchmark", *ACM Transactions on Database Systems*, volumen 17, número 1 (1992).

[**Ceri y Owicki 1983**] S. Ceri y S. Owicki, "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases", *Proc. of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks* (1983).

[**Ceri y Pelagatti 1984**] S. Ceri y G. Pelagatti, *Distributed Databases: Principles and Systems*, McGraw-Hill (1984).

[**Chakrabarti 1999**] S. Chakrabarti, "Recent Results in Automatic Web Resource Discovery", *ACM Computing Surveys*, volumen 31, número 4 (1999).

[**Chakrabarti 2000**] S. Chakrabarti, "Data Mining for Hypertext: A Tutorial Survey", *SIGKDD Explorations*, volumen 1, número 2 (2000), páginas 1–11.

[**Chakrabarti 2002**] S. Chakrabarti, *Mining the Web: Discovering Knowledge from HyperText Data*, Morgan Kaufmann (2002).

[**Chakrabarti et al. 1998**] S. Chakrabarti, S. Sarawagi y B. Dom, "Mining Surprising Patterns Using Temporal Description Length", *Proc. of the International Conf. on Very Large Databases* (1998), páginas 606–617.

[**Chakrabarti et al. 1999**] S. Chakrabarti, M. van den Berg y B. Dom, "Focused Crawling: A New Approach to Topic Specific Web Resource Discovery", *Proc. of the International World Wide Web Conf.* (1999).

[**Chakravarthy et al. 1990**] U. S. Chakravarthy, J. Grant y J. Minker, "Logic-Based Approach to Semantic Query Optimization", *ACM Transactions on Database Systems*, volumen 15, número 2 (1990), páginas 162–207.

[**Chamberlin 1996**] D. Chamberlin, *Using the New DB2: IBM's Object-Relational Database System*, Morgan Kaufmann (1996).

[**Chamberlin 1998**] D. D. Chamberlin, *A Complete Guide to DB2 Universal Database*, Morgan Kaufmann (1998).

[**Chamberlin et al. 1976**] D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner y B. W. Wade, "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control", *IBM Journal of Research and Development*, volumen 20, número 6 (1976), páginas 560–575.

[**Chamberlin et al. 1981**] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade y R. A. Yost, "A History and Evaluation of System R", *Communications of the ACM*, volumen 24, número 10 (1981), páginas 632–646.

[**Chamberlin et al. 2000**] D. D. Chamberlin, J. Robie y D. Florescu, "Quilt: An XML Query Language for Heterogeneous Data Sources", *Proc. of the International Workshop on the Web and Databases (WebDB)* (2000), páginas 53–62.

[**Chamberlin y Boyce 1974**] D. D. Chamberlin y R. F. Boyce, "SEQUEL: A Structured English Query Language", *ACM SIGMOD Workshop on Data Description, Access, and Control* (1974), páginas 249–264.

[**Chan y Ioannidis 1998**] C.-Y. Chan y Y. E. Ioannidis, "Bitmap Index Design and Evaluation", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1998).

[**Chan y Ioannidis 1999**] C.-Y. Chan y Y. E. Ioannidis, "An Efficient Bitmap Encoding Scheme for Selection Queries", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1999).

[**Chandra y Harel 1982**] A. K. Chandra y D. Harel, "Structure and Complexity of Relational Queries", *Journal of Computer and System Sciences*, volumen 15, número 10 (1982), páginas 99–128.

[**Chandrasekaran et al. 2003**] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss y M. A. Shah, "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World", *Proc. First Biennial Conference on Innovative Data Systems Research* (2003).

[**Chandy et al. 1975**] K. M. Chandy, J. C. Browne, C. W. Dissley y W. R. Uhrig, "Analytic Models for Rollback and Recovery Strategies in Database Systems", *IEEE Transactions on Software Engineering*, volumen SE-1, número 1 (1975), páginas 100–110.

[**Chatziantoniou y Ross 1997**] D. Chatziantoniou y K. A. Ross, "Groupwise Processing of Relational Queries", *Proc. of the International Conf. on Very Large Databases* (1997), páginas 476–485.

[**Chaudhuri et al. 1995**] S. Chaudhuri, R. Krishnamurthy, S. Potamianos y K. Shim, "Optimizing Queries with Materialized Views", *Proc. of the International Conf. on Data Engineering*, Taipei, Taiwan (1995).

[**Chaudhuri et al. 1999**] S. Chaudhuri, E. Christensen, G. Graefe, V. Narasayya y M. Zwilling, ""Self Tuning Technology in Microsoft SQL Server", *IEEE Data Engineering Bulletin*, volumen 22, número 2 (1999).

[**Chaudhuri et al. 2003**] S. Chaudhuri, K. Ganjam, V. Ganti y R. Motwani, "Robust and Efficient Fuzzy Match for Online Data Cleaning", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2003).

[**Chaudhuri y Narasayya 1997**] S. Chaudhuri y V. Narasayya, "An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server", *Proc. of the International Conf. on Very Large Databases* (1997).

[**Chaudhuri y Shim 1994**] S. Chaudhuri y K. Shim, "Including Group-By in Query Optimization", *Proc. of the International Conf. on Very Large Databases* (1994).

[**Chen 1976**] P. P. Chen, "The Entity-Relationship Model: Toward a Unified View of Data", *ACM Transactions on Database Systems*, volumen 1, número 1 (1976), páginas 9–36.

[**Chen et al. 1994**] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz y D. A. Patterson, "RAID: High-Performance, Reliable Secondary Storage", *ACM Computing Survey*, volumen 26, número 2 (1994).

[**Chen y Patterson 1990**] P. Chen y D. Patterson, "Maximizing Performance in a Striped Disk Array", *Proc. of the Seventeenth Annual International Symposium on Computer Architecture* (1990).

[**Chomicki 1995**] J. Chomicki, "Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding", *ACM Transactions on Database Systems*, volumen 20, número 2 (1995), páginas 149–186.

[**Chou y Dewitt 1985**] H. T. Chou y D. J. Dewitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems", *Proc. of the International Conf. on Very Large Databases* (1985), páginas 127–141.

[**Cochrane et al. 1996**] R. Cochrane, H. Pirahesh y N. M. Mattos, "Integrating Triggers and Declarative Constraints in SQL Database Sytems", *Proc. of the International Conf. on Very Large Databases* (1996).

[**Codd 1970**] E. F. Codd, "A Relational Model for Large Shared Data Banks", *Communications of the ACM*, volumen 13, número 6 (1970), páginas 377–387.

[**Codd 1972**] E. F. Codd. "Further Normalization of the Data Base Relational Model", In *Rustin [1972]*, páginas 33–64 (1972).

[**Codd 1979**] E. F. Codd, "Extending the Database Relational Model to Capture More Meaning", *ACM Transactions on Database Systems*, volumen 4, número 4 (1979), páginas 397–434.

[**Codd 1982**] E. F. Codd, "The 1981 ACM Turing Award Lecture: Relational Database: A Practical Foundation for Productivity", *Communications of the ACM*, volumen 25, número 2 (1982), páginas 109–117.

[**Codd 1990**] E. F. Codd, *The Relational Model for Database Management: Version 2*, Addison Wesley (1990).

[**Comer 1979**] D. Comer, "The Ubiquitous B-tree", *ACM Computing Survey*, volumen 11, número 2 (1979), páginas 121–137.

[**Comer y Droms 2003**] D. E. Comer y R. E. Droms, *Computer Networks and Internets*, 4ª edición, Prentice Hall (2003).

[**Cook 1996**] M. A. Cook, *Building Enterprise Information Architecture: Reengineering Information Systems*, Prentice Hall (1996).

[**Cook et al. 1999**] J. Cook, R. Harbus y T. Shirai, *The DB2 Universal Database V6.1 Certification Guide: For UNIX, Windows, and OS/2*, Prentice Hall (1999).

[**Cormen et al. 1990**] T. Cormen, C. Leiserson y R. Rivest, *Introduction to Algorithms*, MIT Press (1990).

[**Cruanes et al. 2004**] T. Cruanes, B. Dageville y B. Ghosh, "Parallel SQL Execution in Oracle 10g", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004), páginas 850–854.

[**Dageville et al. 2004**] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait y M. Ziauddin, "Automatic SQL Tuning in Oracle 10g", *Proc. of the International Conf. on Very Large Databases* (2004), páginas 1098–1109.

[**Dageville y Zait 2002**] B. Dageville y M. Zait, "SQL Memory Management in Oracle 9i", *Proc. of the International Conf. on Very Large Databases* (2002), páginas 962–973.

[**Dalvi et al. 2001**] N. N. Dalvi, S. K. Sanghai, P. Roy y S. Sudarshan, "Pipelining in Multi-Query Optimization", *Proc. of the ACM Symposium on Principles of Database Systems* (2001).

[**Daniels et al. 1982**] D. Daniels, P. G. Selinger, L. M. Haas, B. G. Lindsay, C. Mohan, A. Walker y P. F. Wilms. "An Introduction to Distributed Query Compilation in R*", In *Schneider [1982]* (1982).

[**Dar et al. 1996**] S. Dar, H. V. Jagadish, A. Levy y D. Srivastava', "Answering Queries with Aggregation Using Views", *Proc. of the International Conf. on Very Large Databases* (1996).

[**Dashti et al. 2003**] A. Dashti, S. H. Kim, C. Shahabi y R. Zimmermann, *Streaming Media Server Design*, Prentice Hall (2003).

[**Date 1986**] C. J. Date, *Relational Databases: selected Writings*, Addison Wesley (1986).

[**Date 1989**] C. Date, *A Guide to DB2*, Addison Wesley (1989).

[**Date 1990**] C. J. Date, *Relational Database Writings, 1985-1989*, Addison Wesley (1990).

[**Date 1993a**] C. J. Date, "How SQL Missed the Boat", *Database Programming and Design*, volumen 6, número 9 (1993).

[**Date 1993b**] C. J. Date, "The Outer Join", *IOCOD-2*, John Wiley and Sons (1993), páginas 76–106.

[**Date 2003**] C. J. Date, *An Introduction to Database Systems*, 8th edición, Addison Wesley (2003).

[**Date y Darwen 1997**] C. J. Date y G. Darwen, *A Guide to the SQL Standard*, 4ª edición, Addison Wesley (1997).

[**Davies 1973**] C. T. Davies, "Recovery Semantics for a DB/DC System", *ACM Annual Conference* (1973), páginas 136–141.

[**Davis et al. 1983**] C. Davis, S. Jajodia, P. A. Ng y R. Yeh, editores, *Entity-Relationship Approach to Software Engineering*, North Holland (1983).

[**Davison y Graefe 1994**] D. L. Davison y G. Graefe, "Memory-Contention Responsive Hash Joins", *Proc. of the International Conf. on Very Large Databases* (1994).

[**Dayal 1987**] U. Dayal, "Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Subqueries, Aggregates and Quantifiers", *Proc. of the International Conf. on Very Large Databases* (1987), páginas 197–208.

[**Dayal et al. 1982**] U. Dayal, N. Goodman y R. H. Katz, "An Extended Relational Algebra with Control over Duplicate Elimination", *Proc. of the ACM Symposium on Principles of Database Systems* (1982).

[**DB2 Online documentation**] DB2 Online documentation. http://www.software.ibm.com/db2/pubs.

[**Deshpande y Larson 1992**] V. Deshpande y P. A. Larson, "The Design and Implementation of a Parallel Join Algorithm for Nested Relations on Shared-Memory Multiprocessors", *Proc. of the International Conf. on Data Engineering* (1992).

[**Deutsch et al. 1999**] A. Deutsch, M. Fernandez, D. Florescu, A. Levy y D. Suciu, "A Query Language for XML", *Proc. of the International World Wide Web Conf.* (1999). (XML-QL also submitted to the World Wide Web Consortium http://www.w3.org/TR/1998/NOTE-xml-ql-19980819).

[**DeWitt 1990**] D. DeWitt, "The Gamma Database Machine Project", *IEEE Transactions on Knowledge and Data Engineering*, volumen 2, número 1 (1990).

[**DeWitt et al. 1986**] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar y M. Muralikrishna, "A High Performance Dataflow Database Machine", *Proc. of the International Conf. on Very Large Databases* (1986).

[**DeWitt et al. 1992**] D. DeWitt, J. Naughton, D. Schneider y S. Seshadri, "Practical Skew Handling in Parallel Joins", *Proc. of the International Conf. on Very Large Databases* (1992).

[**DeWitt y Gray 1992**] D. DeWitt y J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems", *Communications of the ACM*, volumen 35, número 6 (1992), páginas 85–98.

[**Dias et al. 1989**] D. Dias, B. Iyer, J. Robinson y P. Yu, "Integrated Concurrency-Coherency Controls for Multisystem Data Sharing", *Software Engineering*, volumen 15, número 4 (1989), páginas 437–448.

[**Dogac et al. 1994**] A. Dogac, M. T. Ozsu, A. Biliris y T. Selis, *Advances in Object-Oriented Database Systems*, volumen 130, Springer Verlag, Computer and Systems Sciences, NATO ASI Series F (1994).

[**Donahoo y Speegle 2005**] M. J. Donahoo y G. D. Speegle, *SQL: Practical Guide for Developers*, Morgan Kaufmann (2005).

[**Douglas y Douglas 2003**] K. Douglas y S. Douglas, *PostgreSQL*, Sam's Publishing (2003).

[**Dubois y Thakkar 1992**] M. Dubois y S. Thakkar, editores, *Scalable Shared Memory Multiprocessors*, Kluwer Academic Publishers (1992).

[**Duncan 1990**] R. Duncan, "A Survey of Parallel Computer Architectures", *IEEE Computer*, volumen 23, número 2 (1990), páginas 5–16.

[**Eisenberg et al. 2004**] A. Eisenberg, J. Melton, K. G. Kulkarni, J.-E. Michels y F. Zemke, "SQL:2003 Has Been Published", *ACM SIGMOD Record*, volumen 33, número 1 (2004), páginas 119–126.

[**Eisenberg y Melton 1999**] A. Eisenberg y J. Melton, "SQL:1999, formerly known as SQL3", *ACM SIGMOD Record*, volumen 28, número 1 (1999).

[**Eisenberg y Melton 2004a**] A. Eisenberg y J. Melton, "Advancements in SQL/XML", *ACM SIGMOD Record*, volumen 33, número 3 (2004), páginas 79–86.

[**Eisenberg y Melton 2004b**] A. Eisenberg y J. Melton, "An Early Look at XQuery API for Java (XQJ)", *ACM SIGMOD Record*, volumen 33, número 2 (2004), páginas 105–111.

[**Ellis 1987**] C. S. Ellis, "Concurrency in Linear Hashing", *ACM Transactions on Database Systems*, volumen 12, número 2 (1987), páginas 195–217.

[**Elmasri y Navathe 2003**] R. Elmasri y S. B. Navathe, *Fundamentals of Database Systems*, 4^a edición, Addison Wesley (2003).

[**Eppinger et al. 1991**] J. L. Eppinger, L. B. Mummert y A. Z. Spector, *Camelot and Avalon: A Distributed Transaction Facility*, Morgan Kaufmann (1991).

[**Epstein et al. 1978**] R. Epstein, M. R. Stonebraker y E. Wong, "Distributed Query Processing in a Relational Database System", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1978), páginas 169–180.

[**Epstein y Stonebraker 1980**] R. Epstein y M. R. Stonebraker, "Analysis of Distributed Database Processing Strategies", *Proc. of the International Conf. on Very Large Databases* (1980), páginas 92–110.

[**Escobar-Molano et al. 1993**] M. Escobar-Molano, R. Hull y D. Jacobs, "Safety and Translation of Calculus Queries with Scalar Functions", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993), páginas 253–264.

[**Eswaran et al. 1976**] K. P. Eswaran, J. N. Gray, R. A. Lorie y I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", *Communications of the ACM*, volumen 19, número 11 (1976), páginas 624–633.

[**Fagin 1977**] R. Fagin, "Multivalued Dependencies and a New Normal Form for Relational Databases", *ACM Transactions on Database Systems*, volumen 2, número 3 (1977), páginas 262–278.

[**Fagin 1979**] R. Fagin, "Normal Forms and Relational Database Operators", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1979).

[**Fagin 1981**] R. Fagin, "A Normal Form for Relational Databases That Is Based on Domains and Keys", *ACM Transactions on Database Systems*, volumen 6, número 3 (1981), páginas 387–415.

[**Fagin et al. 1979**] R. Fagin, J. Nievergelt, N. Pippenger y H. R. Strong, "Extendible Hashing — A Fast Access Method for

Dynamic Files", *ACM Transactions on Database Systems*, volumen 4, número 3 (1979), páginas 315–344.

[**Faloutsos y Lin 1995**] C. Faloutsos y K.-I. Lin, "Fast Map: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), páginas 163–174.

[**Fayyad et al. 1995**] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth y R. Uthurusamy, *Advances in Knowledge Discovery and Data Mining*, MIT Press (1995).

[**Fekete et al. 1990**] A. Fekete, N. Lynch, M. Merritt y W. Weihl, "Commutativity-Based Locking for Nested Transactions", *Journal of Computer and System Science* (1990), páginas 65–156.

[**Fekete et al. 2005**] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil y D. Shasha, "Making Snapshot Isolation Serializable", *ACM Transactions on Database Systems*, volumen 30, número 2 (2005).

[**Finkel y Bentley 1974**] R. A. Finkel y J. L. Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys", *Acta Informatica*, volumen 4, (1974), páginas 1–9.

[**Finkelstein 1982**] S. Finkelstein, "Common Expression Analysis in Database Applications", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1982), páginas 235–245.

[**Fischer 2001**] L. Fischer, editor, *Workflow Handbook 2001*, Future Strategies (2001).

[**Fischer 2004**] L. Fischer, editor, *The Workflow Handbook 2004*, Future Strategies Inc. (2004).

[**Florescu et al. 2000**] D. Florescu, D. Kossmann y I. Monalescu, "Integrating Keyword Search into XML Query Processing", *Proc. of the International World Wide Web Conf.* (2000), páginas 119–135. También aparece en una edición especial de Computer Networks.

[**Florescu y Kossmann 1999**] D. Florescu y D. Kossmann, "Storing and Querying XML Data Using an RDBMS", *IEEE Data Engineering Bulletin (Special Issue on XML)* (1999), páginas 27–35.

[**Franklin et al. 1992**] M. J. Franklin, M. J. Zwilling, C. K. Tan, M. J. Carey y D. J. DeWitt, "Crash Recovery in Client-Server EXODUS", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1992), páginas 165–174.

[**Franklin et al. 1993**] M. J. Franklin, M. Carey y M. Livny, "Local Disk Caching for Client-Server Database Systems", *Proc. of the International Conf. on Very Large Databases* (1993).

[**Fredkin 1960**] E. Fredkin, "Trie Memory", *Communications of the ACM*, volumen 4, número 2 (1960), páginas 490–499.

[**Freedman y DeWitt 1995**] C. S. Freedman y D. J. DeWitt, "The SPIFFI Scalable Video-on-Demand Server", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), páginas 352–363.

[**Funderburk et al. 2002a**] J. E. Funderburk, G. Kiernan, J. Shanmugasundaram, E. Shekita y C. Wei, "XTABLES: Bridging Relational Technology and XML", *IBM Systems Journal*, volumen 41, número 4 (2002), páginas 616–641.

[**Funderburk et al. 2002b**] J. E. Funderburk, S. Malaika y B. Reinwald, "XML Programming with SQL/XML and XQuery", *IBM Systems Journal*, volumen 41, número 4 (2002), páginas 642–665.

[**Fushimi et al. 1986**] S. Fushimi, M. Kitsuregawa y H. Tanaka, "An Overview of the Systems Software of a Parallel Relational Database Machine: GRACE", *Proc. of the International Conf. on Very Large Databases* (1986).

[**Galindo-Legaria 1994**] C. Galindo-Legaria, "Outerjoins as Disjunctions", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1994).

[**Galindo-Legaria et al. 2004**] C. Galindo-Legaria, S. Stefani y F. Waas, "Query Processing for SQL Updates", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004), páginas 844–849.

[**Galindo-Legaria y Joshi 2001**] C. A. Galindo-Legaria y M. M. Joshi, "Orthogonal Optimization of Subqueries and Aggregation", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2001).

[**Galindo-Legaria y Rosenthal 1992**] C. Galindo-Legaria y A. Rosenthal, "How to Extend a Conventional Optimizer to Handle One- and Two-Sided Outerjoin", *Proc. of the International Conf. on Data Engineering* (1992), páginas 402–409.

[**Ganguly 1998**] S. Ganguly, "Design and Analysis of Parametric Query Optimization Algorithms", *Proc. of the International Conf. on Very Large Databases*, New York City, New York (1998).

[**Ganguly et al. 1992**] S. Ganguly, W. Hasan y R. Krishnamurthy, "Query Optimization for Parallel Execution", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1992).

[**Ganguly et al. 1996**] S. Ganguly, P. Gibbons, Y. Matias y A. Silberschatz, "A Sampling Algorithm for Estimating Join Size", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[**Ganski y Wong 1987**] R. A. Ganski y H. K. T. Wong, "Optimization of Nested SQL Queries Revisited", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1987).

[**Garcia-Molina 1982**] H. Garcia-Molina, "Elections in Distributed Computing Systems", *IEEE Transactions on Computers*, volumen C-31, número 1 (1982), páginas 48–59.

[**Garcia-Molina et al. 2001**] H. Garcia-Molina, J. D. Ullman y J. D. Widom, *Database Systems: The Complete Book*, Prentice Hall (2001).

[**Garcia-Molina y Salem 1987**] H. Garcia-Molina y K. Salem, "Sagas", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1987), páginas 249–259.

[**Garcia-Molina y Salem 1992**] H. Garcia-Molina y K. Salem, "Main Memory Database Systems: An Overview", *IEEE Transactions on Knowledge and Data Engineering*, volumen 4, número 6 (1992), páginas 509–516.

[**Gawlick 1998**] D. Gawlick, "Messaging/Queuing in Oracle8", *Proc. of the International Conf. on Data Engineering* (1998), páginas 66–68.

[**Georgakopoulos et al. 1994**] D. Georgakopoulos, M. Rusinkiewicz y A. Seth, "Using Tickets to Enforce the Serializability of Multidatabase Transactions", *IEEE Transactions on Knowledge and Data Engineering*, volumen 6, número 1 (1994), páginas 166–180.

[**Gifford 1979**] D. K. Gifford, "Weighted Voting for Replicated Data", *Proc. the ACM SIGOPS Symposium on Operating Systems Principles* (1979), páginas 150–162.

[**Graefe 1990**] G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), páginas 102–111.

[**Graefe 1993**] G. Graefe, "Query Evaluation Techniques for Large Databases", *ACM Computing Survey*, volumen 25, número 2 (1993), páginas 73–170.

[**Graefe 1995**] G. Graefe, "The Cascades Framework for Query Optimization", *Data Engineering Bulletin*, volumen 18, número 3 (1995), páginas 19–29.

[**Graefe et al. 1998**] G. Graefe, R. Bunker y S. Cooper, "Hash Joins and Hash Teams in Microsoft SQL Server", *Proc. of the International Conf. on Very Large Databases* (1998), páginas 86–97.

[**Graefe y McKenna 1993**] G. Graefe y W. McKenna, "The Volcano Optimizer Generator", *Proc. of the International Conf. on Data Engineering* (1993), páginas 209–218.

[**Gray 1978**] J. Gray. "Notes on Data Base Operating System", In *Bayer et al. [1978]*, páginas 393–481 (1978).

[**Gray 1981**] J. Gray, "The Transaction Concept: Virtues and Limitations", *Proc. of the International Conf. on Very Large Databases* (1981), páginas 144–154.

[**Gray 1991**] J. Gray, *The Benchmark Handbook for Database and Transaction Processing Systems*, 2^a edición, Morgan Kaufmann (1991).

[**Gray et al. 1975**] J. Gray, R. A. Lorie y G. R. Putzolu, "Granularity of Locks and Degrees of Consistency in a Shared Data Base", *Proc. of the International Conf. on Very Large Databases* (1975), páginas 428–451.

[**Gray et al. 1976**] J. Gray, R. A. Lorie, G. R. Putzolu y I. L. Traiger, *Granularity of Locks and Degrees of Consistency in a Shared Data Base*, Nijssen (1976).

[**Gray et al. 1981**] J. Gray, P. R. McJones y M. Blasgen, "The Recovery Manager of the System R Database Manager", *ACM Computing Survey*, volumen 13, número 2 (1981), páginas 223–242.

[**Gray et al. 1990**] J. Gray, B. Horst y M. Walker, "Parity Striping of Disc Arrays: Low-Cost Reliable Storage with Acceptable Throughput", *Proc. of the International Conf. on Very Large Databases* (1990), páginas 148–161.

[**Gray et al. 1995**] J. Gray, A. Bosworth, A. Layman y H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab and Sub-Totals", informe técnico, Microsoft Research (1995).

[**Gray et al. 1996**] J. Gray, P. Helland y P. O'Neil, "The Dangers of Replication and a Solution", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996), páginas 173–182.

[**Gray et al. 1997**] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow y H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals", *Data Mining and Knowledge Discovery*, volumen 1, número 1 (1997), páginas 29–53.

[**Gray y Edwards 1995**] J. Gray y J. Edwards, "Scale Up with TP Monitors", *Byte*, volumen 20, número 4 (1995), páginas 123–130.

[**Gray y Graefe 1997**] J. Gray y G. Graefe, "The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb", *SIGMOD Record*, volumen 26, número 4 (1997), páginas 63–68.

[**Gray y Putzolu 1987**] J. Gray y G. R. Putzolu, "The 5 Minute Rule for Trading Memory for Disk Accesses and the 10 Byte Rule for Trading Memory for CPU Time", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1987), páginas 395–398.

[**Gray y Reuter 1993**] J. Gray y A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann (1993).

[**Gregersen y Jensen 1999**] H. Gregersen y C. S. Jensen, "Temporal Entity-Relationship Models-A Survey", *IEEE Transactions on Knowledge and Data Engineering*, volumen 11, número 3 (1999), páginas 464–497.

[**Griffin y Libkin 1995**] T. Griffin y L. Libkin, "Incremental Maintenance of Views with Duplicates", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995).

[**Grossman y Frieder 2004**] D. A. Grossman y O. Frieder, *Information Retrieval: Algorithms and Heuristics*, 2ª edición, Springer Verlag (2004).

[**Guo et al. 2003**] L. Guo, F. Shao, C. Botev y J. Shanmugasundaram, "XRANK: Ranked Keyword Search over XML Documents", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2003).

[**Gupta 1997**] H. Gupta, "Selection of Views to Materialize in a Data Warehouse", *Proc. of the International Conf. on Database Theory* (1997).

[**Gupta y Mumick 1995**] A. Gupta y I. S. Mumick, "Maintenance of Materialized Views: Problems, Techniques and Applications", *IEEE Data Engineering Bulletin*, volumen 18, número 2 (1995).

[**Guttman 1984**] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1984), páginas 47–57.

[**H. Lu y Tan 1991**] M. S. H. Lu y K. Tan, "Optimization of Multi-Way Join Queries for Parallel Execution", *Proc. of the International Conf. on Very Large Databases* (1991).

[**Haas et al. 1989**] L. M. Haas, J. C. Freytag, G. M. Lohman y H. Pirahesh, "Extensible Query Processing in Starburst", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1989), páginas 377–388.

[**Haas et al. 1990**] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. G. Lindsay, H. Pirahesh, M. J. Carey y E. J. Shekita, "Starburst Mid-Flight: As the Dust Clears", *IEEE Transactions on Knowledge and Data Engineering*, volumen 2, número 1 (1990), páginas 143–160.

[**Haerder y Reuter 1983**] T. Haerder y A. Reuter, "Principles of Transaction-Oriented Database Recovery", *ACM Computing Survey*, volumen 15, número 4 (1983), páginas 287–318.

[**Haerder y Rothermel 1987**] T. Haerder y K. Rothermel, "Concepts for Transaction Recovery in Nested Transactions", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1987), páginas 239–248.

[**Hall 1976**] P. A. V. Hall, "Optimization of a Single Relational Expression in a Relational Database System", *IBM Journal of Research and Development*, volumen 20, número 3 (1976), páginas 244–257.

[**Halsall 1996**] F. Halsall, *Data Communications, Computer Networks, and Open Systems*, 4ª edición, Addison Wesley (1996).

[**Han y Kamber 2000**] J. Han y M. Kamber, *Data Mining: Concepts and Techniques*, Morgan Kaufmann (2000).

[**Harinarayan et al. 1996**] V. Harinarayan, J. D. Ullman y A. Rajaraman, "Implementing Data Cubes Efficiently", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[**Haritsa et al. 1990**] J. Haritsa, M. Carey y M. Livny, "On Being Optimistic about Real-Time Constraints", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990).

[**Harizopoulos y Ailamaki 2004**] S. Harizopoulos y A. Ailamaki, "STEPS towards Cache-resident Transaction Processing", *Proc. of the International Conf. on Very Large Databases* (2004), páginas 660–671.

[**Hellerstein et al. 1995**] J. M. Hellerstein, J. F. Naughton y A. Pfeffer, "Generalized Search Trees for Database Systems", *Proc. of the International Conf. on Very Large Databases* (1995), páginas 562–573.

[**Hellerstein y Stonebraker 2005**] J. M. Hellerstein y M. Stonebraker, editores, *Readings in Database Systems*, 4ª edición, Morgan Kaufmann (2005).

[**Hennessy et al. 2002**] J. L. Hennessy, D. A. Patterson y D. Goldberg, *Computer Architecture: A Quantitative Approach*, 3ª edición, Morgan Kaufmann (2002).

[**Hevner y Yao 1979**] A. R. Hevner y S. B. Yao, "Query Processing in Distributed Database Systems", *IEEE Transactions on Software Engineering*, volumen SE-5, número 3 (1979), páginas 177–187.

[**Heywood et al. 2002**] I. Heywood, S. Cornelius y S. Carver, *An Introduction to Geographical Information Systems, Second Edition*, Prentice Hall (2002).

[**Hollingsworth 1994**] D. Hollingsworth, *The Workflow Reference Model*, Workflow Management Coalition, TC00-1003 (1994).

[**Hollingsworth 2004**] D. Hollingsworth. "The Workflow Reference Model 10 Years On", In Fischer [2004] (2004).

[**Hong et al. 1993**] D. Hong, T. Johnson y S. Chakravarthy, "Real-Time Transaction Scheduling: A Cost Conscious Approach", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993).

[**Hong y Stonebraker 1991**] W. Hong y M. Stonebraker, "Optimization of Parallel Query Execution Plans in XPRS", *Proc. of the International Conf. on Parallel and Distributed Information Systems* (1991), páginas 218–225.

[**Howes et al. 1999**] T. A. Howes, M. C. Smith y G. S. Good, *Understanding and Deploying LDAP Directory Services*, Macmillan Publishing, New York (1999).

[**Hristidis y Papakonstantinou 2002**] V. Hristidis y Y. Papakonstantinou, "DISCOVER: Keyword Search in Relational Databases", *Proc. of the International Conf. on Very Large Databases* (2002).

[**Huang y Garcia-Molina 2001**] Y. Huang y H. Garcia-Molina, "Exactly-once Semantics in a Replicated Messaging System", *Proc. of the International Conf. on Data Engineering* (2001), páginas 3–12.

[**Hulgeri y Sudarshan 2003**] A. Hulgeri y S. Sudarshan, "AniPQO: Almost Non-Intrusive Parametric Query Optimization for Non-Linear Cost Functions", *Proc. of the International Conf. on Very Large Databases* (2003).

[**IBM 1987**] IBM, "Systems Application Architecture: Common Programming Interface, Database Reference", informe técnico, IBM Corporation, IBM Form Number SC26–4348–0 (1987).

[**IDEF1X 1993**] IDEF1X, "Integration Definition for Information Modeling (IDEF1X)", informe técnico Federal Information Processing Standards Publication 184, National Institute of Standards and Technology (NIST), disponible en www.idef.com/Downloads/pdf/Idef1x.pdf (1993).

[**Imielinski y Badrinath 1994**] T. Imielinski y B. R. Badrinath, "Mobile Computing — Solutions and Challenges", *Communications of the ACM*, volumen 37, número 10 (1994).

[**Imielinski y Korth 1996**] T. Imielinski y H. F. Korth, editores, *Mobile Computing*, Kluwer Academic Publishers (1996).

[**Ioannidis et al. 1992**] Y. E. Ioannidis, R. T. Ng, K. Shim y T. K. Sellis, "Parametric Query Optimization", *Proc. of the International Conf. on Very Large Databases* (1992), páginas 103–114.

[**Ioannidis y Christodoulakis 1993**] Y. Ioannidis y S. Christodoulakis, "Optimal Histograms for Limiting Worst-Case Error Propagation in the Size of Join Results", *ACM Transactions on Database Systems*, volumen 18, número 4 (1993), páginas 709–748.

[**Ioannidis y Poosala 1995**] Y. E. Ioannidis y V. Poosala, "Balancing Histogram Optimality and Practicality for Query Result Size Estimation", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), páginas 233–244.

[**Jackson y Moulinier 2002**] P. Jackson y I. Moulinier, *Natural Language Processing for Online Applications: Text Retrieval, Extraction, and Categorization*, John Benjamin (2002).

[**Jagadish et al. 1993**] H. V. Jagadish, A. Silberschatz y S. Sudarshan, "Recovering from Main-Memory Lapses", *Proc. of the International Conf. on Very Large Databases* (1993).

[**Jagadish et al. 1994**] H. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz y S. Sudarshan, *Dali: A High Performance Main Memory Storage Manager* (1994).

[**Jain y Dubes 1988**] A. K. Jain y R. C. Dubes, *Algorithms for Clustering Data*, Prentice Hall (1988).

[**Jensen et al. 1994**] C. S. Jensen et al., "A Consensus Glossary of Temporal Database Concepts", *ACM SIGMOD Record*, volumen 23, número 1 (1994), páginas 52–64.

[**Jensen et al. 1996**] C. S. Jensen, R. T. Snodgrass y M. Soo, "Extending Existing Dependency Theory to Temporal Databases", *IEEE Transactions on Knowledge and Data Engineering*, volumen 8, número 4 (1996), páginas 563–582.

[**Jhingran et al. 1997**] A. Jhingran, T. Malkemus y S. Padmanabhan, "Query Optimization in DB2 Parallel Edition", *Data Engineering Bulletin*, volumen 20, número 2 (1997), páginas 27–34.

[**Johnson 1999**] T. Johnson, "Performance Measurements of Compressed Bitmap Indices", *Proc. of the International Conf. on Very Large Databases* (1999).

[**Johnson y Shasha 1993**] T. Johnson y D. Shasha, "The Performance of Concurrent B-Tree Algorithms", *ACM Transactions on Database Systems*, volumen 18, número 1 (1993).

[**Jones y Willet 1997**] K. S. Jones y P. Willet, editores, *Readings in Information Retrieval*, Morgan Kaufmann (1997).

[**Jordan y Russell 2003**] D. Jordan y C. Russell, *Java Data Objects*, O'Reilly (2003).

[**Joshi 1991**] A. Joshi, "Adaptive Locking Strategies in a Multi-Node Shared Data Model Environment", *Proc. of the International Conf. on Very Large Databases* (1991).

[**Joshi et al. 1998**] A. Joshi, W. Bridge, J. Loaiza y T. Lahiri, "Checkpointing in Oracle", *Proc. of the International Conf. on Very Large Databases* (1998), páginas 665–668.

[**Kanne y Moerkotte 2000**] C.-C. Kanne y G. Moerkotte, "Efficient Storage of XML Data", *Proc. of the International Conf. on Data Engineering* (2000), página 198.

[**Kapitskaia et al. 2000**] O. Kapitskaia, R. T. Ng y D. Srivastava, "Evolution and Revolutions in LDAP Directory Caches", *Proc. of the International Conf. on Extending Database Technology* (2000), páginas 202–216.

[**Katz et al. 2004**] H. Katz, D. Chamberlin, D. Draper, M. Fernandez, M. Kay, J. Robie, M. Rys, J. Simeon, J. Tivy y P. Wadler, *XQuery from the Experts: A Guide to the W3C XML Query Language*, Addison Wesley (2004).

[**Kaushik et al. 2004**] R. Kaushik, R. Krishnamurthy, J. F. Naughton y R. Ramakrishnan, "On the Integration of Structure Indexes and Inverted Lists", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004).

[**Kedem y Silberschatz 1979**] Z. M. Kedem y A. Silberschatz, "Controlling Concurrency Using Locking Protocols", *Proc. of the Annual IEEE Symposium on Foundations of Computer Science* (1979), páginas 275–285.

[**Kedem y Silberschatz 1983**] Z. M. Kedem y A. Silberschatz, "Locking Protocols: From Exclusive to Shared Locks", *ACM Press*, volumen 30, número 4 (1983), páginas 787–804.

[**Kim 1982**] W. Kim, "On Optimizing an SQL-like Nested Query", *ACM Transactions on Database Systems*, volumen 3, número 3 (1982), páginas 443–469.

[**Kim 1995**] W. Kim, editor, *Modern Database Systems*, ACM Press/ Addison Wesley (1995).

[**King 1981**] J. J. King, "QUIST: A System for Semantic Query Optimization in Relational Data Bases", *Proc. of the International Conf. on Very Large Databases* (1981), páginas 510–517.

[**King et al. 1991**] R. P. King, N. Halim, H. Garcia-Molina y C. Polyzois, "Management of a Remote Backup Copy for Disaster Recovery", *ACM Transactions on Database Systems*, volumen 16, número 2 (1991), páginas 338–368.

[**Kitsuregawa et al. 1983**] M. Kitsuregawa, H. Tanaka y T. MotoOka, "Application of Hash to a Database Machine and its Architecture", *New Generation Computing*, volumen 1, número 1 (1983), páginas 62–74.

[**Kitsuregawa y Ogawa 1990**] M. Kitsuregawa y Y. Ogawa, "Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Skew in the Super Database Computer", *Proc. of the International Conf. on Very Large Databases* (1990), páginas 210–221.

[**Kleinberg 1999**] J. M. Kleinberg, "Authoritative Sources in a Hyperlinked Environment", *Journal of the ACM*, volumen 46, número 5 (1999), páginas 604–632.

[**Kleinrock 1975**] L. Kleinrock, *Queuing Systems, Volume 1: Theory*, John Wiley and Sons (1975).

[**Kleinrock 1976**] L. Kleinrock, *Queuing Systems, Volume 2: Computer Applications*, John Wiley and Sons (1976).

[**Klug 1982**] A. Klug, "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions", *ACM Press*, volumen 29, número 3 (1982), páginas 699–717.

[**Knapp 1987**] E. Knapp, "Deadlock Detection in Distributed Databases", *ACM Computing Survey*, volumen 19, número 4 (1987).

[**Knuth 1973**] D. E. Knuth, *The Art of Computer Programming, volumen 3*, Addison Wesley, Sorting and Searching (1973).

[**Kohavi y Provost 2001**] R. Kohavi y F. Provost, editores, *Applications of Data Mining to Electronic Commerce*, Kluwer Academic Publishers (2001).

[**Konstan et al. 1997**] J. A. Konstan, B. N. Miller, D. Maltz, J. L. Herlocker, L. R. Gordon y J. Riedl, "GroupLens: Applying Collaborative Filtering to Usenet News", *Communications of the ACM*, volumen 40, número 3 (1997), páginas 77–87.

[**Korth 1982**] H. F. Korth, "Deadlock Freedom Using Edge Locks", *ACM Transactions on Database Systems*, volumen 7, número 4 (1982), páginas 632–652.

[**Korth 1983**] H. F. Korth, "Locking Primitives in a Database System", *Journal of the ACM*, volumen 30, número 1 (1983), páginas 55–79.

[**Korth et al. 1990**] H. F. Korth, E. Levy y A. Silberschatz, "A Formal Approach to Recovery by Compensating Transactions", *Proc. of the International Conf. on Very Large Databases* (1990).

[**Korth y Speegle 1994**] H. F. Korth y G. Speegle, "Formal Aspects of Concurrency Control in Long Duration Transaction Systems Using the NT/PV Model", *ACM Transactions on Database Systems*, volumen 19, número 3 (1994), páginas 492–535.

[**Krishnaprasad et al. 2004**] M. Krishnaprasad, Z. Liu, A. Manikutty, J. W. Warner, V. Arora y S. Kotsovolos, "Query Rewrite for XML in Oracle XML DB", *Proc. of the International Conf. on Very Large Databases* (2004), páginas 1122–1133.

[**Kung y Lehman 1980**] H. T. Kung y P. L. Lehman, "Concurrent Manipulation of Binary Search Trees", *ACM Transactions on Database Systems*, volumen 5, número 3 (1980), páginas 339–353.

[**Kung y Robinson 1981**] H. T. Kung y J. T. Robinson, "Optimistic Concurrency Control", *ACM Transactions on Database Systems*, volumen 6, número 2 (1981), páginas 312–326.

[**Labio et al. 1997**] W. Labio, D. Quass y B. Adelberg, "Physical Database Design for Data Warehouses", *Proc. of the International Conf. on Data Engineering* (1997).

[**Lahiri et al. 2001a**] T. Lahiri, A. Ganesh, R. Weiss y A. Joshi, "Fast-Start: Quick Fault Recovery in Oracle", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2001).

[**Lahiri et al. 2001b**] T. Lahiri, V. Srihari, W. Chan, N. MacNaughton y S. Chandrasekaran, "Cache Fusion: Extending Shared-Disk Clusters with Shared Caches", *Proc. of the International Conf. on Very Large Databases* (2001), páginas 683–686.

[**Lai y Wilkinson 1984**] M. Y. Lai y W. K. Wilkinson, "Distributed Transaction Management in JASMIN", *Proc. of the International Conf. on Very Large Databases* (1984), páginas 466–472.

[**Lam y Kuo 2001**] K.-Y. Lam y T.-W. Kuo, editores, *Real-Time Database Systems*, Kluwer (2001).

[**Lamb et al. 1991**] C. Lamb, G. Landis, J. Orenstein y D. Weinreb, "The ObjectStore Database System", *Communications of the ACM*, volumen 34, número 10 (1991), páginas 51–63.

[**Lamport 1978**] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, volumen 21, número 7 (1978), páginas 558–565.

[**Lampson y Sturgis 1976**] B. Lampson y H. Sturgis, "Crash Recovery in a Distributed Data Storage System", informe técnico, Computer Science Laboratory, Xerox Palo Alto Research Center,Palo Alto (1976).

[**Lanzelotte et al. 1993**] R. Lanzelotte, P. Valduriez y M. Zar, "On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces", *Proc. of the International Conf. on Very Large Databases* (1993).

[**Larson y Yang 1985**] P. Larson y H. Z. Yang, "Computing Queries from Derived Relations", *Proc. of the International Conf. on Very Large Databases* (1985), páginas 259–269.

[**Lecluse et al. 1988**] C. Lecluse, P. Richard y F. Velez, "O2: An Object-Oriented Data Model", *Proc. of the International Conf. on Very Large Databases* (1988), páginas 424–433.

[**Lehman y Yao 1981**] P. L. Lehman y S. B. Yao, "Efficient Locking for Concurrent Operations on B-trees", *ACM Transactions on Database Systems*, volumen 6, número 4 (1981), páginas 650–670.

[**Lehner et al. 2000**] W. Lehner, R. Sidle, H. Pirahesh y R. Cochrane, "Maintenance of Automatic Summary Tables", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2000), páginas 512–513.

[**Li y Mozes 2004**] W. Li y A. Mozes, "Computing Frequent Itemsets Inside Oracle 10G", *Proc. of the International Conf. on Very Large Databases* (2004), páginas 1253–1256.

[**Lin et al. 1994**] E. T. Lin, E. R. Omiecinski y S. Yalamanchili, "Large Join Optimization on a Hypercube Multiprocessor", *IEEE Transactions on Knowledge and Data Engineering*, volumen 6, número 2 (1994), páginas 304–315.

[**Lindsay et al. 1980**] B. G. Lindsay, P. G. Selinger, C. Galtieri, J. N. Gray, R. A. Lorie, T. G. Price, G. R. Putzolu, I. L. Traiger y B. W. Wade. "Notes on Distributed Databases", In Draffen y Poole, editores, *Distributed Data Bases*, páginas 247–284. Cambridge University Press, Cambridge, England (1980).

[**Litwin 1978**] W. Litwin, "Virtual Hashing: A Dynamically Changing Hashing", *Proc. of the International Conf. on Very Large Databases* (1978), páginas 517–523.

[**Litwin 1980**] W. Litwin, "Linear Hashing: A New Tool for File and Table Addressing", *Proc. of the International Conf. on Very Large Databases* (1980), páginas 212–223.

[**Litwin 1981**] W. Litwin, "Trie Hashing", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1981), páginas 19–29.

[**Lo y Ravishankar 1996**] M.-L. Lo y C. V. Ravishankar, "Spatial Hash-Joins", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[**Loeb 1998**] L. Loeb, *Secure Electronic Transactions : Introduction and Technical Reference*, Artech House (1998).

[**Lomet 1981**] D. G. Lomet, "Digital B-trees", *Proc. of the International Conf. on Very Large Databases* (1981), páginas 333–344.

[**Lynch 1983**] N. A. Lynch, "Multilevel Atomicity–A New Correctness Criterion for Database Concurrency Control", *ACM Transactions on Database Systems*, volumen 8, número 4 (1983), páginas 484–502.

[**Lynch et al. 1988**] N. A. Lynch, M. Merritt, W. Weihl y A. Fekete, "A Theory of Atomic Transactions", *Proc. of the International Conf. on Database Theory* (1988), páginas 41–71.

[**Lynch y Merritt 1986**] N. A. Lynch y M. Merritt, "Introduction to the Theory of Nested Transactions", *Proc. of the International Conf. on Database Theory* (1986).

[**Mackert y Lohman 1986**] L. F. Mackert y G. M. Lohman, "R* Optimizer Validation and Performance Evaluation for Distributed Queries", *Proc. of the International Conf. on Very Large Databases* (1986).

[**Maier 1983**] D. Maier, *The Theory of Relational Databases*, Computer Science Press, Rockville (1983).

[**Martin et al. 1989**] J. Martin, K. K. Chapman y J. Leben, *DB2, Concepts, Design, and Programming*, Prentice Hall (1989).

[**Mattison 1996**] R. Mattison, *Data Warehousing: Strategies, Technologies, and Techniques*, McGraw-Hill (1996).

[**McHugh y Widom 1999**] J. McHugh y J. Widom, "Query Optimization for XML", *Proc. of the International Conf. on Very Large Databases* (1999), páginas 315–326.

[**Mehrotra et al. 1991**] S. Mehrotra, R. Rastogi, H. F. Korth y A. Silberschatz, "Non-Serializable Executions in Heterogeneous Distributed Database Systems", *Proc. of the International Conf. on Parallel and Distributed Information Systems* (1991).

[**Mehrotra et al. 2001**] S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. . Korth y A. Silberschatz, "Overcoming Heterogeneity and Autonomy in Multidatabase Systems.", *Inf. Comput.*, volumen 167, número 2 (2001), páginas 137–172.

[**Melton 2002**] J. Melton, *Advanced SQL:1999 – Understanding Object-Relational and Other Advanced Features*, Morgan Kaufmann (2002).

[**Melton y Eisenberg 2000**] J. Melton y A. Eisenberg, *Understanding SQL and Java Together : A Guide to SQLJ, JDBC, and Related Technologies*, Morgan Kaufmann (2000).

[**Melton y Simon 1993**] J. Melton y A. R. Simon, *Understanding The New SQL: A Complete Guide*, Morgan Kaufmann (1993).

[**Melton y Simon 2001**] J. Melton y A. R. Simon, *SQL:1999 Understanding Relational Language Components*, Morgan Kaufmann (2001).

[**Menasce et al. 1980**] D. A. Menasce, G. Popek y R. Muntz, "A Locking Protocol for Resource Coordination in Distributed Databases", *ACM Transactions on Database Systems*, volumen 5, número 2 (1980), páginas 103–138.

[**Microsoft 1997**] Microsoft, *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference*, Microsoft Press (1997).

[**Mistry et al. 2001**] H. Mistry, P. Roy, S. Sudarshan y K. Ramamritham, "Materialized View Selection and Maintenan-

ce Using Multi-Query Optimization", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2001).

[**Mitchell 1997**] T. M. Mitchell, *Machine Learning*, McGraw-Hill (1997).

[**Mohan 1990a**] C. Mohan, "ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operations on B-Tree indexes", *Proc. of the International Conf. on Very Large Databases* (1990), páginas 392–405.

[**Mohan 1990b**] C. Mohan, "Commit-LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems", *Proc. of the International Conf. on Very Large Databases* (1990), páginas 406–418.

[**Mohan 1993**] C. Mohan, "IBM's Relational Database Products:Features and Technologies", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993).

[**Mohan et al. 1986**] C. Mohan, B. Lindsay y R. Obermarck, "Transaction Management in the R* Distributed Database Management System", *ACM Transactions on Database Systems*, volumen 11, número 4 (1986), páginas 378–396.

[**Mohan et al. 1992**] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh y P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", *ACM Transactions on Database Systems*, volumen 17, número 1 (1992).

[**Mohan y Levine 1992**] C. Mohan y F. Levine, "ARIES/IM:An Efficient and High-Concurrency Index Management Method Using Write-Ahead Logging", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1992).

[**Mohan y Lindsay 1983**] C. Mohan y B. Lindsay, "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions", *Proc. of the ACM Symposium on Principles of Distributed Computing* (1983).

[**Mohan y Narang 1991**] C. Mohan y I. Narang, "Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment", *Proc. of the International Conf. on Very Large Databases* (1991).

[**Mohan y Narang 1992**] C. Mohan y I. Narang, "Efficient Locking and Caching of Data in the Multisystem Shared Disks Transaction Environment", *Proc. of the International Conf. on Extending Database Technology* (1992).

[**Mohan y Narang 1994**] C. Mohan y I. Narang, "ARIES/CSA: A Method for Database Recovery in Client-Server Architectures", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1994), páginas 55–66.

[**Moss 1985**] J. E. B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, Cambridge (1985).

[**Moss 1987**] J. E. B. Moss, "Log-Based Recovery for Nested Transactions", *Proc. of the International Conf. on Very Large Databases* (1987), páginas 427–432.

[**MSDN: XML Developer Center**] MSDN: XML Developer Center. "XML and the Database". http://msdn.microsoft.-com/XML/BuildingXML/XMLandDatabase/default.aspx.

[**Murthy y Banerjee 2003**] R. Murthy y S. Banerjee, "XML Schemas in Oracle XML DB", *Proc. of the International Conf. on Very Large Databases* (2003), páginas 1009–1018.

[**Nakayama et al. 1984**] T. Nakayama, M. Hirakawa y T. Ichikawa, "Architecture and Algorithm for Parallel Execution of a Join Operation", *Proc. of the International Conf. on Data Engineering* (1984).

[**Ng y Han 1994**] R. T. Ng y J. Han, "Efficient and Effective Clustering Methods for Spatial Data Mining", *Proc. of the International Conf. on Very Large Databases* (1994).

[**Nyberg et al. 1995**] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray y D. B. Lomet, "AlphaSort: A Cache-Sensitive Parallel External Sort", *VLDB Journal*, volumen 4, número 4 (1995), páginas 603–627.

[**O'Neil et al. 2004**] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller y N. Westbury, "ORDPATHs: Insert-Friendly XML Node Labels", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004), páginas 903–908.

[**O'Neil y O'Neil 2000**] P. O'Neil y E. O'Neil, *Database: Principles, Programming, Performance*, 2^a edición, Morgan Kaufmann (2000).

[**O'Neil y Quass 1997**] P. O'Neil y D. Quass, "Improved Query Performance with Variant Indexes", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1997).

[**Orenstein 1982**] J. A. Orenstein, "Multidimensional Tries Used for Associative Searching", *Information Processing Letters*, volumen 14, número 4 (1982), páginas 150–157.

[**Ozcan et al. 1997**] F. Ozcan, S. Nural, P. Koksal, C. Evrendilek y A. Dogac, "Dynamic Query Optimization in Multidatabases", *Data Engineering Bulletin*, volumen 20, número 3 (1997), páginas 38–45.

[**Ozden et al. 1994**] B. Ozden, A. Biliris, R. Rastogi y A. Silberschatz, "A Low-cost Storage Server for a Movie on Demand Database", *Proc. of the International Conf. on Very Large Databases* (1994).

[**Ozden et al. 1996a**] B. Ozden, R. Rastogi, P. Shenoy y A. Silberschatz, "Fault-Tolerant Architectures for Continuous Media Servers", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[**Ozden et al. 1996b**] B. Ozden, R. Rastogi y A. Silberschatz, "On the Design of a Low-Cost Video-on-Demand Storage

System", *Multimedia Systems Journal*, volumen 4, número 1 (1996), páginas 40–54.

[**Ozsoyoglu y Snodgrass 1995**] G. Ozsoyoglu y R. Snodgrass, "Temporal and Real-Time Databases: A Survey", *IEEE Transactions on Knowledge and Data Engineering*, volumen 7, número 4 (1995), páginas 513–532.

[**Ozsu y Valduriez 1999**] T. Ozsu y P. Valduriez, *Principles of Distributed Database Systems*, 2ª edición, Prentice Hall (1999).

[**Padmanabhan et al. 2003**] S. Padmanabhan, B. Bhattacharjee, T. Malkemus, L. Cranston y M. Huras, "Multi-Dimensional Clustering: A New Data Layout Scheme in DB2", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2003), páginas 637–641.

[**Pal et al. 2004**] S. Pal, I. Cseri, G. Schaller, O. Seeliger, L. Giakoumakis y V. Zolotov, "Indexing XML Data Stored in a Relational Database", *Proc. of the International Conf. on Very Large Databases* (2004), páginas 1134–1145.

[**Pang et al. 1995**] H.-H. Pang, M. J. Carey y M. Livny, "Multiclass Scheduling in Real-Time Database Systems", *IEEE Transactions on Knowledge and Data Engineering*, volumen 2, número 4 (1995), páginas 533–551.

[**Papadimitriou 1979**] C. H. Papadimitriou, "The Serializability of Concurrent Database Updates", *Journal of the ACM*, volumen 26, número 4 (1979), páginas 631–653.

[**Papadimitriou 1986**] C. H. Papadimitriou, *The Theory of Database Concurrency Control*, Computer Science Press, Rockville (1986).

[**Papadimitriou et al. 1977**] C. H. Papadimitriou, P. A. Bernstein y J. B. Rothnie, "Some Computational Problems Related to Database Concurrency Control", *Proc. of the Conf. on Theoretical Computer Science* (1977), páginas 275–282.

[**Papakonstantinou et al. 1996**] Y. Papakonstantinou, A. Gupta y L. Haas, "Capabilities-Based Query Rewriting in Mediator Systems", *Proc. of the International Conf. on Parallel and Distributed Information Systems* (1996).

[**Parker et al. 1983**] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser y C. Kline, "Detection of Mutual Inconsistency in Distributed Systems", *IEEE Transactions on Software Engineering*, volumen 9, número 3 (1983), páginas 240–246.

[**Patel y DeWitt 1996**] J. Patel y D. J. DeWitt, "Partition Based Spatial-Merge Join", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[**Patterson 2004**] D. P. Patterson, "Latency Lags Bandwidth", *Communications of the ACM*, volumen 47, número 10 (2004), páginas 71–75.

[**Patterson et al. 1988**] D. A. Patterson, G. Gibson y R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1988), páginas 109–116.

[**Pellenkoft et al. 1997**] A. Pellenkoft, C. A. Galindo-Legaria y M. Kersten, "The Complexity of Transformation-Based Join Enumeration", *Proc. of the International Conf. on Very Large Databases*, Athens, Greece (1997), páginas 306–315.

[**Pless 1998**] V. Pless, *Introduction to the Theory of Error-Correcting Codes*, 3ª edición, John Wiley and Sons (1998).

[**Poe 1995**] V. Poe, *Building a Data Warehouse for Decision Support*, Prentice Hall (1995).

[**Poess y Floyd 2000**] M. Poess y C. Floyd, "New TPC Benchmarks for Decision Support and Web Commerce", *ACM SIGMOD Record*, volumen 29, número 4 (2000).

[**Poess y Potapov 2003**] M. Poess y D. Potapov, "Data Compression in Oracle", *Proc. of the International Conf. on Very Large Databases* (2003), páginas 937–947.

[**Polyzois y Garcia-Molina 1994**] C. Polyzois y H. Garcia-Molina, "Evaluation of Remote Backup Algorithms for Transaction-Processing Systems", *ACM Transactions on Database Systems*, volumen 19, número 3 (1994), páginas 423–449.

[**Poosala et al. 1996**] V. Poosala, Y. E. Ioannidis, P. J. Haas y E. J. Shekita, "Improved Histograms for Selectivity Estimation of Range Predicates", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996), páginas 294–305.

[**Popek et al. 1981**] G. J. Popek, B. J. Walker, J. M. Chow, D. Edwards, C. Kline, G. Rudisin y G. Thiel, "LOCUS: A Network Transparent, High Reliability Distributed System", *Proc. of the Eighth Symposium on Operating System Principles* (1981), páginas 169–177.

[**Pu et al. 1988**] C. Pu, G. Kaiser y N. Hutchinson, "Split-Transactions for Open-Ended Activities", *Proc. of the International Conf. on Very Large Databases* (1988), páginas 26–37.

[**Rahm 1993**] E. Rahm, "Empirical Performance Evaluation of Concurrency and Coherency Control Protocols for Database Sharing Systems", *ACM Transactions on Database Systems*, volumen 8, número 2 (1993).

[**Ramakrishna y Larson 1989**] M. V. Ramakrishna y P. Larson, "File Organization Using Composite Perfect Hashing", *ACM Transactions on Database Systems*, volumen 14, número 2 (1989), páginas 231–263.

[**Ramakrishnan et al. 1992**] R. Ramakrishnan, D. Srivastava y S. Sudarshan, *Controlling the Search in Bottom-up Evaluation* (1992).

[**Ramakrishnan y Gehrke 2002**] R. Ramakrishnan y J. Gehrke, *Database Management Systems*, 3ª edición, McGraw-Hill (2002).

[**Ramakrishnan y Ullman 1995**] R. Ramakrishnan y J. D. Ullman, "A Survey of Deductive Database Systems", *Journal of Logic Programming*, volumen 23, número 2 (1995), páginas 125–149.

[**Ramesh et al. 1989**] R. Ramesh, A. J. G. Babu y J. P. Kincaid, "Index Optimization: Theory and Experimental Results", *ACM Transactions on Database Systems*, volumen 14, número 1 (1989), páginas 41–74.

[**Rangan et al. 1992**] P. V. Rangan, H. M. Vin y S. Ramanathan, "Designing an On-Demand Multimedia Service", *Communications Magazine*, volumen 1, número 1 (1992), páginas 56–64.

[**Rao y Ross 2000**] J. Rao y K. A. Ross, "Making B+-Trees Cache Conscious in Main Memory", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2000), páginas 475–486.

[**Rathi et al. 1990**] A. Rathi, H. Lu y G. E. Hedrick, "Performance Comparison of Extendable Hashing and Linear Hashing Techniques", *Proc. ACM SIGSmall/PC Symposium on Small Systems* (1990), páginas 178–185.

[**Reed 1978**] D. Reed, *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, Department of Electrical Engineering, MIT, Cambridge (1978).

[**Reed 1983**] D. Reed, "Implementing Atomic Actions on Decentralized Data", *Transactions on Computer Systems*, volumen 1, número 1 (1983), páginas 3–23.

[**Revesz 2002**] P. Revesz, *Introduction to Constraint Databases*, Springer Verlag (2002).

[**Richardson et al. 1987**] J. Richardson, H. Lu y K. Mikkilineni, "Design and Evaluation of Parallel Pipelined Join Algorithms", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1987).

[**Rivest 1976**] R. L. Rivest, "Partial Match Retrieval Via the Method of Superimposed Codes", *SIAM Journal of Computing*, volumen 5, número 1 (1976), páginas 19–50.

[**Rizvi et al. 2004**] S. Rizvi, A. Mendelzon, S. Sudarshan y P. Roy, "Extending Query Rewriting Techniques for Fine-Grained Access Control", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004).

[**Robinson 1981**] J. Robinson, "The k-d-B Tree: A Search Structure for Large Multidimensional Indexes", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1981), páginas 10–18.

[**Roos 2002**] R. M. Roos, *Java Data Objects*, Pearson Education (2002).

[**Rosch 2003**] W. L. Rosch, *The Winn L. Rosch Hardware Bible*, 6^a edición, Que (2003).

[**Rosenblum y Ousterhout 1991**] M. Rosenblum y J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System", *Proc. of the International Conf. on Architectural Support for Programming Languages and Operating Systems* (1991), páginas 1–15.

[**Rosenthal y Reiner 1984**] A. Rosenthal y D. Reiner, "Extending the Algebraic Framework of Query Processing to Handle Outerjoins", *Proc. of the International Conf. on Very Large Databases* (1984), páginas 334–343.

[**Ross 1990**] K. A. Ross, "Modular Stratification and Magic Sets for DATALOG Programs with Negation", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990).

[**Ross 1999**] S. M. Ross, *Introduction to Probability and Statistics for Engineers and Scientists*, Harcourt/Academic Press (1999).

[**Ross et al. 1996**] K. Ross, D. Srivastava y S. Sudarshan, "Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[**Ross y Srivastava 1997**] K. A. Ross y D. Srivastava, "Fast Computation of Sparse Datacubes", *Proc. of the International Conf. on Very Large Databases* (1997), páginas 116–125.

[**Rothermel y Mohan 1989**] K. Rothermel y C. Mohan, "ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions", *Proc. of the International Conf. on Very Large Databases* (1989), páginas 337–346.

[**Roy et al. 2000**] P. Roy, S. Seshadri, S. Sudarshan y S. Bhobhe, "Efficient and Extensible Algorithms for Multi-Query Optimization", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2000).

[**Rusinkiewicz y Sheth 1995**] M. Rusinkiewicz y A. Sheth. "Specification and Execution of Transactional Workflows", In *Kim [1995]*, páginas 592–620 (1995).

[**Rustin 1972**] R. Rustin, *Data Base Systems*, Prentice Hall (1972).

[**Rys 2001**] M. Rys, "Bringing the Internet to Your Database: Using SQL Server 2000 and XML to Build Loosely-Coupled Systems", *Proc. of the International Conf. on Data Engineering* (2001), páginas 465–472.

[**Rys 2003**] M. Rys. "XQuery and Relational Database Systems", In H. Katz, editor, *XQuery From the Experts*, páginas 353–391. Addison Wesley (2003).

[**Rys 2004**] M. Rys. "What's New in FOR XML in Microsoft SQL Server 2005". http://msdn.microsoft.com/library/en-us/dnsql90/html/forxml2k5.asp (2004).

[**Sagiv y Yannakakis 1981**] Y. Sagiv y M. Yannakakis, "Equivalence among Relational Expressions with the Union and Difference Operators", *Proc. of the ACM SIGMOD Conf. on Management of Data*, volumen 27, número 4 (1981).

[**Salem et al. 1994**] K. Salem, H. Garcia-Molina y J. Sands, "Altruistic Locking", *ACM Transactions on Database Systems*, volumen 19, número 1 (1994), páginas 117–165.

[**Salem y Garcia-Molina 1986**] K. Salem y H. Garcia-Molina, "Disk Striping", *Proc. of the International Conf. on Data Engineering* (1986), páginas 336–342.

[**Salton 1989**] G. Salton, *Automatic Text Processing*, Addison Wesley (1989).

[**Samet 1990**] H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison Wesley (1990).

[**Samet 1995a**] H. Samet, "General Research Issues in Multimedia Database Systems", *ACM Computing Survey*, volumen 27, número 4 (1995), páginas 630–632.

[**Samet 1995b**] H. Samet. "Spatial Data Structures", In *Kim [1995]*, páginas 361–385 (1995).

[**Samet y Aref 1995**] H. Samet y W. Aref. "Spatial Data Models and Query Processing", In *Kim [1995]*, páginas 338–360 (1995).

[**Sanders 1998**] R. E. Sanders, *ODBC 3.5 Developer's Guide*, McGraw-Hill (1998).

[**Sanders 2000**] R. E. Sanders, *DB2 Universal Database SQL Developer's Guide*, McGraw-Hill (2000).

[**Sarawagi 2000**] S. Sarawagi, "User-Adaptive Exploration of Multidimensional Data", *Proc. of the International Conf. on Very Large Databases* (2000), páginas 307–316.

[**Sarawagi et al. 2002**] S. Sarawagi, A. Bhamidipaty, A. Kirpal y C. Mouli, "ALIAS: An Active Learning Led Interactive Deduplication System", *Proc. of the International Conf. on Very Large Databases* (2002), páginas 1103–1106.

[**Schlageter 1981**] G. Schlageter, "Optimistic Methods for Concurrency Control in Distributed Database Systems", *Proc. of the International Conf. on Very Large Databases* (1981), páginas 125–130.

[**Schneider 1982**] H. J. Schneider, *Distributed Data Bases* (1982).

[**Schneider y DeWitt 1989**] D. Schneider y D. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1989).

[**Schning 2001**] H. Schning, "Tamino - A DBMS Designed for XML", *Proc. of the International Conf. on Data Engineering* (2001), páginas 149–154.

[**Selinger et al. 1979**] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie y T. G. Price, "Access Path Selection in a Relational Database System", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1979), páginas 23–34.

[**Selinger y Adiba 1980**] P. G. Selinger y M. E. Adiba, "Access Path Selection in Distributed Database Management Systems", informe técnico RJ2338, IBM Research Laboratory, San Jose (1980).

[**Sellis 1988**] T. K. Sellis, "Multiple Query Optimization", *ACM Transactions on Database Systems*, volumen 13, número 1 (1988), páginas 23–52.

[**Sellis et al. 1987**] T. K. Sellis, N. Roussopoulos y C. Faloutsos, "The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects", *Proc. of the International Conf. on Very Large Databases* (1987), páginas 507–518.

[**Seshadri et al. 1996**] P. Seshadri, H. Pirahesh y T. Y. C. Leung, "Complex Query Decorrelation", *Proc. of the International Conf. on Data Engineering* (1996), páginas 450–458.

[**Shafer et al. 1996**] J. C. Shafer, R. Agrawal y M. Mehta, "SPRINT: A Scalable Parallel Classifier for Data Mining", *Proc. of the International Conf. on Very Large Databases* (1996), páginas 544–555.

[**Shanmugasundaram et al. 1999**] J. Shanmugasundaram, G. He, K. Tufte, C. Zhang, D. DeWitt y J. Naughton, "Relational Databases for Querying XML Documents: Limitations and Opportunities", *Proc. of the International Conf. on Very Large Databases* (1999).

[**Shapiro 1986**] L. D. Shapiro, "Join Processing in Database Systems with Large Main Memories", *ACM Transactions on Database Systems*, volumen 11, número 3 (1986), páginas 239–264.

[**Shasha et al. 1995**] D. Shasha, F. Llirabat, E. Simon y P. Valduriez, "Transaction Chopping: Algorithms and Performance Studies", *ACM Transactions on Database Systems*, volumen 20, número 3 (1995), páginas 325–363.

[**Shasha y Bonnet 2002**] D. Shasha y P. Bonnet, *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*, Morgan Kaufmann (2002).

[**Shatdal y Naughton 1993**] A. Shatdal y J. Naughton, "Using Shared Virtual Memory for Parallel Join Processing", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993).

[**Silberschatz 1982**] A. Silberschatz, "A Multi-Version Concurrency Control Scheme With No Rollbacks", *Proc. of the ACM Symposium on Principles of Distributed Computing* (1982), páginas 216–223.

[**Silberschatz et al. 1990**] A. Silberschatz, M. R. Stonebraker y J. D. Ullman, "Database Systems: Achievements and Opportunities", *ACM SIGMOD Record*, volumen 19, número 4 (1990).

[**Silberschatz et al. 1996**] A. Silberschatz, M. Stonebraker y J. Ullman, "Database Research: Achievements and Opportunities into the 21st Century", informe técnico CS-TR-96-1563, Department of Computer Science, Stanford University, Stanford (1996).

[**Silberschatz et al. 2001**] A. Silberschatz, P. B. Galvin y G. Gagne, *Operating System Concepts*, 6^a edición, John Wiley and Sons (2001).

[**Silberschatz y Kedem 1980**] A. Silberschatz y Z. Kedem, "Consistency in Hierarchical Database Systems", *Journal of the ACM*, volumen 27, número 1 (1980), páginas 72–80.

[**Simmen et al. 1996**] D. Simmen, E. Shekita y T. Malkemus, "Fundamental Techniques for Order Optimization", *Proc. of the ACM SIGMOD Conf. on Management of Data*, Montreal, Canada (1996), páginas 57–67.

[**Skeen 1981**] D. Skeen, "Non-blocking Commit Protocols", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1981), páginas 133–142.

[**Soderland 1999**] S. Soderland, "Learning Information Extraction Rules for Semi-structured and Free Text", *Machine Learning*, volumen 34, número 1–3 (1999), páginas 233–272.

[**Soo 1991**] M. Soo, "Bibliography on Temporal Databases", *ACM SIGMOD Record*, volumen 20, número 1 (1991), páginas 14–23.

[**Spector y Schwarz 1983**] A. Z. Spector y P. M. Schwarz, "Transactions: A Construct for Reliable Distributed Computing", *Operating Systems Review*, volumen 17, número 2 (1983), páginas 18–35.

[**SQL/XML 2004**] SQL/XML. "ISO/IEC 9075-14:2003, Information Technology: Database languages: SQL.Part 14: XML-Related Specifications (SQL/XML)" (2004).

[**Srikant y Agrawal 1996a**] R. Srikant y R. Agrawal, "Mining Quantitative Association Rules in Large Relational Tables", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996).

[**Srikant y Agrawal 1996b**] R. Srikant y R. Agrawal, "Mining Sequential Patterns: Generalizations and Performance Improvements", *Proc. of the International Conf. on Extending Database Technology* (1996), páginas 3–17.

[**Srinivasan et al. 2000a**] J. Srinivasan, S. Das, C. Freiwald, E. I. Chong, M. Jagannath, A. Yalamanchi, R. Krishnan, A.-T. Tran, S. DeFazio y J. Banerjee, "Oracle 8i Index-Organized Table and Its Application to New Domains", *Proc. of the International Conf. on Very Large Databases* (2000), páginas 285–296.

[**Srinivasan et al. 2000b**] J. Srinivasan, R. Murthy, S. Sundara, N. Agarwal y S. DeFazio, "Extensible Indexing: A Framework for Integrating Domain-Specific Indexing Schemes into Oracle 8i", *Proc. of the International Conf. on Data Engineering* (2000), páginas 91–100.

[**Stam y Snodgrass 1988**] R. Stam y R. Snodgrass, "A Bibliography on Temporal Databases", *IEEE Transactions on Knowledge and Data Engineering*, volumen 7, número 4 (1988), páginas 231–239.

[**Stinson 2002**] B. Stinson, *PostgreSQL Essential Reference*, New Riders (2002).

[**Stonebraker 1986**] M. Stonebraker, "Inclusion of New Types in Relational Database Systems", *Proc. of the International Conf. on Data Engineering* (1986), páginas 262–269.

[**Stonebraker et al. 1989**] M. Stonebraker, P. Aoki y M. Seltzer, "Parallelism in XPRS", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1989).

[**Stonebraker et al. 1990**] M. Stonebraker, A. Jhingran, J. Goh y S. Potamianos, "On Rules, Procedure, Caching and Views in Database Systems", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), páginas 281–290.

[**Stonebraker y Rowe 1986**] M. Stonebraker y L. Rowe, "The Design of POSTGRES", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1986).

[**Stuart et al. 1984**] D. G. Stuart, G. Buckley y A. Silberschatz, "A Centralized Deadlock Detection Algorithm", informe técnico, Department of Computer Sciences, University of Texas, Austin (1984).

[**Sudarshan y Ramakrishnan 1991**] S. Sudarshan y R. Ramakrishnan, "Aggregation and Relevance in Deductive Databases", *Proc. of the International Conf. on Very Large Databases* (1991).

[**Tanenbaum 2002**] A. S. Tanenbaum, *Computer Networks*, 4^a edición, Prentice Hall (2002).

[**Tansel et al. 1993**] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev y R. Snodgrass, *Temporal Databases: Theory, Design and Implementation*, Benjamin Cummings, Redwood City (1993).

[**Teorey et al. 1986**] T. J. Teorey, D. Yang y J. P. Fry, "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model", *ACM Computing Survey*, volumen 18, número 2 (1986), páginas 197–222.

[**Thalheim 2000**] B. Thalheim, *Entity-Relationship Modeling : Foundations of Database Technology*, Springer Verlag (2000).

[**Thomas 1979**] R. H. Thomas, "A Majority Consensus Approach to Concurrency Control", *ACM Transactions on Database Systems*, volumen 4, número 2 (1979), páginas 180–219.

[**Thomas 1996**] S. A. Thomas, *IPng and the TCP/IP Protocols: Implementing the Next Generation Internet*, John Wiley and Sons (1996).

[**Traiger et al. 1982**] I. L. Traiger, J. N. Gray, C. A. Galtieri y B. G. Lindsay, "Transactions and Consistency in Distributed Database Management Systems", *ACM Transactions on Database Systems*, volumen 7, número 3 (1982), páginas 323–342.

[**Tremblay y Sorenson 1985**] J. P. Tremblay y P. G. Sorenson, *The Theory and Practice of Compiler Writing*, McGraw-Hill (1985).

[**Tsukuda et al. 1992**] S. Tsukuda, M. Nakano, M. Kitsuregawa y M. Takagi, "Parallel Hash Join on Shared-Everything Multiprocessor", *Proc. of the International Conf. on Data Engineering* (1992).

[**Tyagi et al. 2003**] S. Tyagi, M. Vorburger, K. McCammon y H. Bobzin, *Core Java Data Objects*, Prentice Hall (2003).

[**Ullman 1988**] J. D. Ullman, *Principles of Database and Knowledge-base Systems, volumen 1*, Computer Science Press, Rockville (1988).

[**Ullman 1989**] J. D. Ullman, *Principles of Database and Knowledge-base Systems, volumen 2*, Computer Science Press, Rockville (1989).

[**Umar 1997**] A. Umar, *Application (Re)Engineering : Building Web-Based Applications and Dealing With Legacies*, Prentice Hall (1997).

[**UniSQL 1991**] *UniSQL/X Database Management System User's Manual: Release 1.2*. UniSQL, Inc. (1991).

[**Verhofstad 1978**] J. S. M. Verhofstad, "Recovery Techniques for Database Systems", *ACM Computing Survey*, volumen 10, número 2 (1978), páginas 167–195.

[**Vista 1998**] D. Vista, "Integration of Incremental View Maintenance into Query Optimizers", *Proc. of the International Conf. on Extending Database Technology* (1998).

[**Vitter 2001**] J. S. Vitter, "External Memory Algorithms and Data Structures: Dealing with Massive Data", *ACM Computing Surveys*, volumen 33, (2001), páginas 209–271.

[**Wachter y Reuter 1992**] H. Wachter y A. Reuter. "The Con-Tract Model", In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann (1992).

[**Walsh et al. 2004**] N. Walsh et al. "XQuery 1.0 and XPath 2.0 Data Model". http://www.w3.org/TR/xpath-datamodel/. currently a W3C Working Draft (2004).

[**Walton et al. 1991**] C. Walton, A. Dale y R. Jenevein, "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins", *Proc. of the International Conf. on Very Large Databases* (1991).

[**Weihl y Liskov 1990**] W. Weihl y B. Liskov. "Implementation of Resilient, Atomic Data Types", In *Zdonik y Maier [1990]*, páginas 332–344 (1990).

[**Weikum 1991**] G. Weikum, "Principles and Realization Strategies of Multilevel Transaction Management", *ACM Transactions on Database Systems*, volumen 16, número 1 (1991).

[**Weikum et al. 1990**] G. Weikum, C. Hasse, P. Broessler y P. Muth, "Multi-Level Recovery", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), páginas 109–123.

[**Weikum y Schek 1984**] G. Weikum y H. J. Schek, "Architectural Issues of Transaction Management in Multi-Level Systems", *Proc. of the International Conf. on Very Large Databases* (1984), páginas 454–465.

[**Weltman y Dahbura 2000**] R. Weltman y T. Dahbura, *LDAP Programming with Java*, Addison Wesley (2000).

[**Wilschut et al. 1995**] A. N. Wilschut, J. Flokstra y P. M. Apers, "Parallel Evaluation of Multi-Join Queues", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), páginas 115–126.

[**Wipfler 1987**] A. J. Wipfler, *CICS: Application Development and Programming*, Macmillan Publishing, New York (1987).

[**Witkowski et al. 2003a**] A. Witkowski, S. Bellamkonda, T. Bozkaya, G. Dorman, N. Folkert, A. Gupta, L. Sheng y S. Subramanian, "Spreadsheets in RDBMS for OLAP", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2003), páginas 52–63.

[**Witkowski et al. 2003b**] A. Witkowski, S. Bellamkonda, T. Bozkaya, N. Folkert, A. Gupta, L. Sheng y S. Subramanian, "Business Modelling Using SQL Spreadsheets", *Proc. of the International Conf. on Very Large Databases* (2003), páginas 1117–1120.

[**Witten et al. 1999**] I. H. Witten, A. Moffat y T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*, Morgan Kaufmann (1999).

[**Witten y Frank 1999**] I. H. Witten y E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann (1999).

[**Wolf 1991**] J. Wolf, "An Effective Algorithm for Parallelizing Hash Joins in the Presence of Data Skew", *Proc. of the International Conf. on Data Engineering* (1991).

[**Wong 1977**] E. Wong, "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases", *Proc. of the Berkeley Workshop on Distributed Data Management and Computer Networks* (1977), páginas 217–235.

[**Wu et al. 2003**] Y. Wu, J. M. Patel y H. V. Jagadish, "Structural Join Order Selection for XML Query Optimization", *Proc. of the International Conf. on Data Engineering* (2003).

[**Wu y Buchmann 1998**] M. Wu y A. Buchmann, "Encoded Bitmap Indexing for Data Warehouses", *Proc. of the International Conf. on Data Engineering* (1998).

[**X/Open 1991**] *X/Open Snapshot: X/Open DTP: XA Interface*. X/Open Company, Ltd. (1991).

[**Yan y Larson 1995**] W. P. Yan y P. A. Larson, "Eager Aggregation and Lazy Aggregation", *Proc. of the International Conf. on Very Large Databases*, Zurich (1995).

[**Yannakakis et al. 1979**] M. Yannakakis, C. H. Papadimitriou y H. T. Kung, "Locking Protocols: Safety and Freedom from Deadlock", *Proc. of the IEEE Symposium on the Foundations of Computer Science* (1979), páginas 286–297.

[**Zaharioudakis et al. 2000**] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh y M. Urata, "Answering Complex SQL Queries using Automatic Summary Tables", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2000), páginas 105–116.

[**Zaniolo et al. 1997**] C. Zaniolo, S. Ceri, C. Faloutsos, R. Snodgrass, R. Zicari y V. S. Subrahmanian, *Advanced Database Systems*, Morgan Kaufmann (1997).

[**Zdonik y Maier 1990**] S. Zdonik y D. Maier, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann (1990).

[**Zeller y Gray 1990**] H. Zeller y J. Gray, "An Adaptive Hash Join Algorithm for Multiuser Environments", *Proc. of the International Conf. on Very Large Databases* (1990), páginas 186–197.

[**Zhang et al. 1996**] T. Zhang, R. Ramakrishnan y M. Livny, "BIRCH: An Efficient Data Clustering Method for Very Large Databases", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1996), páginas 103–114.

[**Zhou y Ross 2004**] J. Zhou y K. A. Ross, "Buffering Database Operations for Enhanced Instruction Cache Performance", *Proc. of the ACM SIGMOD Conf. on Management of Data* (2004), páginas 191–202.

[**Zhuge et al. 1995**] Y. Zhuge, H. Garcia-Molina, J. Hammer y J. Widom, "View Maintenance in a Warehousing Environment", *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), páginas 316–327.

[**Zikopoulos et al. 2000**] P. Zikopoulos, R. Melnyk y L. Logomirski, *DB2 for Dummies*, Hungry Minds, Inc. (2000).

[**Zikopoulos et al. 2004**] P. Zikopoulos, G. Baklarz, D. deRoos y R. B. Melnyk, *DB2 Version 8: The Official Guide*, IBM Press (2004).

[**Zilio et al. 2004**] D. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano y S. Fadden, "DB2 Design Advisor: Integrated Automatic Physical Database Design", *Proc. of the International Conf. on Very Large Databases* (2004).

[**Zloof 1977**] M. M. Zloof, "Query-by-Example: A Data Base Language", *IBM Systems Journal*, volumen 16, número 4 (1977), páginas 324–343.

Índice

1FN, *véase* forma normal, primera
2FN, *véase* forma normal, segunda
3FN, *véase* forma normal, tercera
4FN, *véase* forma normal, cuarta

a priori, 622
ABD, *véase* administrador de bases de datos
abstracción, 605
abstracción de datos, 4
  nivel de vistas, 5
  nivel físico, 5
  nivel lógico, 5
Access de Microsoft, 59
ACID, 508, 777, 780
Active Directory, 290
Active Server Pages, 270
Active Server Pages.NET, 272
actualizaciones, 55, 87
ad hoc, 743
administrador de bases de datos, 22
ADO (ActiveX Data Objects), 117, 747, 905
ADO.NET, 117
AES (Advanced Encryption Standard), 286
Agarwal, Sameet, 885
agregación, 196
  funciones, 49, 73, 75
    binarias, 608
  operación, 50
  paralela, 684
agrupación de conexiones, 273
agrupaciones, 615, 660
agrupamiento, 623
  aglomerativo, 623
  divisivo, 623
  jerárquico, 623
Ailamaki, Anastassia, 807
aislamiento, 508, 509
AIX, 859
ajuste de la curva, 620
ajuste del rendimiento, 733
álgebra relacional, 36
  expresión, 38, 715
algoritmo
  de acoso, 713
  de elección, 713, 724
  de grafos, 644
  de programación dinámica, 489
  de recuperación, 701
  del ascensor, 373
  híbrido de reunión por mezcla, 457
  impaciente, 616
  Rijndael, 286
alias, 696, 721
  de tipos, 102, 103
all privileges, 111
almacenamiento, 367
  óptico, 368
  conectado en red, 372
  distribuido de datos, 694
  en cinta, 368
  en conexión, 369
  en discos magnéticos, 368
  estable, 568
  nativo, 353
  no volátil, 369, 568, 584
  primario, 369
  secundario, 369
  seguro, 287
  sin conexión, 369
  terciario, 369, 382
  volátil, 369, 568
almacenes de datos, 602, 612
  arquitectura dirigida por
    el destino, 613
    los orígenes, 613
alta disponibilidad, 567
Amazon, 357
ámbito, 313
Ampex, exploración helicoidal, 383
ampliabilidad, 658
  de transacciones, 658
  lineal, 658
  por lotes, 658
  sublineal, 658
análisis estadístico, 602
anfitriones móviles, 768
anidamiento, 312
anidar, 332
  subconsultas, 76
ANSI (American National Standards Institute), 61, 745, 809, 891
Apache
  DB, 328
  proyecto Jakarta, 270
  sistema Tomcat, 297
aplicaciones
  multisistema, 781
árboles, 340
  B+, *véase* índices, árbol B+
  B, *véase* índices, árbol B
  binario, 410
    completo, 500
  cuadráticos, 762
    PR, 762
    regionales, 762
  de operadores, 466, 678
  en memoria, 410
  equilibrado, 408
  k-d, 761
  k-d B, 761
  R, *véase* índices, árbol R
archivos, 386
  cabecera, 388
  estructura, 390
  organización, 373
    asociativa (hash), 390
    en montículos, 390
    secuencial, 390
  reorganizar, 391
  secuenciales, 390
    fragmentados, 374
    indexados, 402
  secuenciales indexados, 436
ARIES, 566, 588, 597, 803, 824, 859, 879, 902
Armstrong, axiomas, 232
  completos, 232
  correctos, 232
Arpanet, 667
arquitectura
  de dos capas, 20
  de memoria no uniforme, 663
  de tres capas, 20
  disco compartido, 660
  jerárquica, 660, 662
  memoria compartida, 660
  paralela de bases de datos, 660
  sin compartimiento, 660
Arquitectura común de agente para solicitudes de objetos, 747
ASCII, 717
aserto, 110
asignación, operación, 48
asistentes para el ajuste, 738
asociación, 421
  abierta, 424
  cerrada, 424
  dinámica, 426
  esquemas, 431
  extensible, 426, 430
  función, 401
  lineal, 431
  reglas, 621
asociaciones, 615, 621

asociativa, propiedad, 46, 478
ASP (Active Server Pages), 260, 270, 272
ASP.NET, 272
asunción de un rol único, 249
atómico, 223
ATA (AT Attachment), 371
ataques
  de diccionario, 287
  de personas intermedias, 788
atasco, 423
atomicidad, 18, 507, 508, 512, 567, 571
  ante fallos, 783
atributos, 13, 29, 172, 334
  autorreferencial, 313
  compuesto, 175, 304
  de dimensión, 603
  de medida, 603
  de partición, 617
  derivado, 176
  descriptivo, 174
  determinado funcionalmente, 233
  determinantes, 618
  monovalorado, 175
  multivalorado, 175
  raros, 235
  simple, 175
  valor, 172
aumento, 522
autentificar, 288
autodocumentado, 331
autonomía, 717
  local, 664
autoridad, 637
autorización, 9, 62, 111
  de actualización, 9
  de eliminación, 9
  de inserción, 9
  de lectura, 9
  en el nivel de las filas, 285
  read, 284
  update, 284
axiomas, 232
ayuda a la toma de decisiones, 602, 673, 742, 743

B+, árbol, *véase* índices, árbol B+
B, árbol, *véase* índices, árbol B
bases de datos, 1
  CAD, 323
  con memoria intermedia, 582
  de diseño, 757
  distribuidas
    heterogéneas, 693
    homogéneas, 693
  en memoria principal, 788
  móviles, 754
  multimedia, 765
  orientadas a objetos, 301, 322
    basadas en lenguajes de programación persistentes, 323
  relacionales
    diseño, 219
    orientadas a objetos, 322
  relacionales basadas en objetos, 301
  temporales, 754
  virtual, 718
    privada, 284
batidores Web, 641
Bayes
  lógica ingenua, 920
  teorema, 620
BEA, 270, 777
biblioteca de etiquetas, 271
big-endian, 718
billete, 798
bits de sal, 287
Blakeley, José A., 885
blob, 104
bloqueos, 522, 529, 700
  bajada, 535
  compartido, 544
  compatibilidad, 530
  comunicación, 657
  con límite de tiempo, 550
  concesión, 529, 533
  conversiones, 535
  de dos fases, 792
  de la siguiente clave, 559
  de valores clave, 559
  del índice, 555
  en modo compartido, 529
  en modo exclusivo, 529
  exclusivo, 544
  explícito, 544
  expropiación, 549
  función de compatibilidad, 529
  gestor, 536
    único, 704
    distribuido, 704
  implícito, 544
  liberación, 657
  modos
    intencional, 545
    intencional-compartido, 545
    intencional-exclusivo, 545
    intencional-exclusivo y compartido, 545
  protocolos, 560
    con árboles B enlazados, 558
    de árbol, 538
    de dos fases, 533, 544
    de dos fases multiversión, 548
    de granularidad múltiple, 546
    definición, 532
    del índice, 555
    del cangrejo, 557
    estricto de dos fases, 534
    riguroso de dos fases, 534
  solicitud, 529
  subida, 535
  tabla, 536
bloques, 373, 383, 570
  clavados, 384
  de memoria intermedia, 570
  físicos, 570
Bluetooth, 769
bolsas, 787
borrado, 54, 84
borrar, instrucción, 553, 553
brazo del disco, 370
buffer, *véase* memoria intermedia
bus, 660
búsqueda, 558
  binaria, 446
  difusa, 613
  lineal, 446

C, 11, 62, 112
  DB2, 863
  monitores de teleprocesamiento, 779
  PostgreSQL, 809
C++, 11, 62, 316, 319, 507
  monitores de teleprocesamiento, 779
C2F, *véase* protocolos, de compromiso, de dos fases
C3F, *véase* protocolos, de compromiso, de tres fases
cabeza, 153
  de lectura y escritura, 370
caché, 367
  alojar, 657
  coherencia, 657, 677
CAD (Computer Aided Design), 756, 757
cadena de desbordamiento, 424
cadenas de caracteres
  operaciones, 69
caída del sistema, 567
caja límite, 762
cajones, 421
  de desbordamiento, 424
  desbordamiento, 423
cálculo relacional
  de dominios, 141
  de tuplas, 137
Call Level Interface, *véase* CLI (Call Level Interface)
cambiadores automáticos, 368
  de cintas, 383
  de discos, 382
cambio de contexto, 778
caminos de acceso, 447
carga, 614
carga de trabajo, 498, 738
case, 124
casos, 621
catálogos, 104, 117, 120, 356
  del sistema, 393
cauce, 466
CD (Compact Disk), 368
celda, 768
celular, *véase* teléfonos móviles
centroide, 623
certificados digitales, 289, 788
CGI (Common Gateway Interface), *véase* interfaz, de pasarela común
ChemML, 356
ciclos falsos, 710
CICS, 777
cierre, 226, 232, 234, 246
  transitivo, 127, 160, 160
cifrado
  clave pública, 286
  clave privada, 286
CIFS, 372
cilindro, 370

cintas magnéticas, 382
círculos, 756
clases de objetos, 720
clasificación, 616
  densa, 98
  por popularidad, 635
  por prestigio, 635
clasificadores
  bayesianos, 620, 620
    ingenuos, 620
  de árboles de decisión, 616
  de redes neuronales, 620
clave de búsqueda, 390
claves, 178
  candidata, 34, 178
  de búsqueda, 402
  externa, 35, 107, 109, 279
  primaria, 34, 63, 178
claves de búsqueda
  accesos bajo varias, 418
  compuesta, 419
CLI (Call Level Interface), 117, 746
  normas, 745
clientes, 20
clob, 104
CLR (Common Language Runtime), 125, 908, 909
CLR .NET, 908
Cobol, 11, 62, 112
CODASYL de DBTG, 745
Codd, E. F., 23, 59, 257
códigos
  de corrección de errores tipo memoria, 377
  de Reed-Solomon, 380
coerción, 102
coincidencia, 347
cola duradera, 780
ColdFusion, *véase* lenguajes, de marcas, de ColdFusion
COM (Component Object Model), 908
Common Object Request Broker Architecture, *véase* CORBA
componentes
  control de concurrencia, 510, 516
  gestión de recuperaciones, 509
  gestión de transacciones, 18, 509
compresión
  de la carga de trabajo, 739
  del prefijo, 417
comprobación
  de la secuencialidad, 522
  de validación, 543
compromiso, *véase* protocolos, de compromiso
concepto, 639
concreción, 605
condiciones
  de excepción, 124
  de partición, 617
conexión
  continua, 667
  discontinua, 667
confianza, 621
conflicto, 517
conjunto de entidades, 13, 171
  débiles, 189, 189, 203
  de nivel inferior, 193
  de nivel superior, 193
  fuertes, 189, 203
  identificadoras, 189
  propietarias, 189
conjunto de relaciones, 13, 173, 179, 200
  binario, 175
  grado, 175
  recursivo, 173
conjuntos
  comparación, 77
  de formación, 616
  de resultados actualizables, 120
  de valores, 175
  grandes de elementos, 622
  operaciones, 71
conmutativa, propiedad, 477
connect by, 834
Consejo para el rendimiento del procesamiento de las transacciones, 742
consenso de quórum, 706
conserva las dependencias, 229
consistencia, 18, 507, 508, 770
  de grado dos, 555
  de los datos, 105, 513, 581
consistente en cuanto a operaciones, 588
consulta gráfica mediante ejemplos, 150
consultas, 410
  básica, 128
  basada en conceptos, 639
  concreta, 674
  de los primeros K, 503
  de proximidad, 760
  de rango, 674
  de vecino más próximo, 760
  definición, 8
  dependiente de la ubicación, 768
  espacial, 760
  monótona, 129
  motor de ejecución, 444
  optimización, 475, 686
    basada en costes, 476
    con tableau, 503
    de actualizaciones, 503
    de agregación, 502
    de los primeros K, 503
    de multiconsultas, 503
    paramétrica, 502
    semántica, 503
  plan de ejecución, 444
  plan de evaluación, 444
  procesador, 22
  procesamiento, 443
    distribuido, 714
    técnicas, 789
  procesamiento distribuido, 769
  recursiva, 128
  regional, 760
  sobre una relación, 144
  sobre varias relaciones, 146
contador lógico, 540
contaminación de los motores de búsqueda, 638
contenido de información, 618
contexto, 333
control de admisión, 766
control de concurrencia, 701
  esquema, 543, 579
  esquemas, 514, 522, 529
  multiversión, 547, 793
  optimista, 544
  subsistema, 386
control de transacciones, 62
controlador de disco, 371
cookie, 266
coordinador
  de transacciones, 697
  suplente, 713
copia en la sombra, 512
copia principal, 694, 705
copo de nieve, esquema, 614
CORBA, 747, 749
corrección fuerte, 797
correlaciones, 622
corresponden, 308
correspondencia de cardinalidades, 14, 177, 200
corte, 605
  de cubos, 605
costes, 714
  de inicio, 659
crawlers, *véase* batidores Web
creación de imágenes, 375, 377
creador de mercado, 787
cross join, 94
Crystal Reports, 261
cuadrícula de diseño, 150
cuadro de condiciones, 147, 147
cubo, 125
cubos de datos, 604
cuellos de botella, 733
cuerpo, 153
cume_dist, 610
current_date, 102
current_time, 102

D'Hers, Thierry, 885
DAT (Digital Audio Tape), 383
Data Encryption Standard, 286
data mining, *véase* minería de datos
Data Universal Numbering System, 747
data warehouses, *véase* almacenes de datos
Database Task Group, 745
Datalog, 27, 137, 151, 162
  programa, 153
  recursividad, 158
  reglas, 153
DataSet, 747
datos
  archivados, 368
  CAD, 756
  de difusión, 770
  de medios continuos, 754, 765
  espaciales, 753
  fragmentados, 353

datos (*cont.*)
geográficos, 753, 756, 758, 759
aplicaciones, 759
globales, 797
isócronos, 765
locales, 797
medios continuos, 766
multidimensionales, 603
multimedia, 754
formatos, 766
por líneas, 758
temporales, 251, 753
válidos, 251
vectoriales, 759
DB2 de IBM, 23, 59
DB2 Universal Database de IBM, 859
DBA, *véase* administrador de bases de datos
DBC de Teradata, 680, 687
DBTG, *véase* Database Task Group
DEC (Digital Equipment Corporation), 671
definición de tipos de documentos, 335
delete, 64, 887
depende
de, 154
directamente de, 154
indirectamente de, 154
dependencias
de compromiso, 539
de reunión, 248
de subconjuntos, 107
del valor, 798
existencial, 189
parcial, 255
que generan igualdades, 245
que generan tuplas, 245
transitivas, 230
dependencias funcionales, 14, 222, 225, 231
cumple, 225
implicada lógicamente, 232
temporales, 251, 755
triviales, 226
dependencias multivaloradas, 244-246
trivial, 245
DES, 286
desanidamiento, 311
desbordamiento de una tabla de asociación, 460
descomposiciones, 224
con pérdidas, 223, 237
que conservan las dependencias, 238
sin pérdidas, 223, 237
desconexiones, 770
descorrelación, 494
descripción, descubrimiento e integración universales, 357
desduplicación, 613
desnormalización, 250
desviación, 622
detección de fallos, 592
diagrama E-R, 180, 200
diagramas de esquema, 35
diccionarios de datos, 9, 393, 394, 837, 842
diferencia de conjuntos, operación, 39, 464
diferencial, 495
DigiCash, 788
Digital Subscriber Line, 667
dimension, 843
directorio, 643, 644
Directorio Activo, 290
discos
ópticos, 382
compartido, 662
con imagen, 569
de registro histórico, 374
de vídeo digital, 368
digital versátil, 368
fallo, 567
magnéticos, 370
planificación, 373
salida forzada de bloques, 384
discriminante, 189
diseño conceptual, 12, 170
disparador, 273
disponibilidad, 710
elevada, 591, 710
dispositivo cabeza-disco, 370
distinct types, 102
distribución de datos, 376
en el nivel de bits, 376
en el nivel de bloques, 376
distribuidor, 907
dividir, 411
división, 558
cuadrática, 764
de la red, 701
estrategias, 674
horizontal, 674
operación, 46
por asociación, 681
por rangos, 681
recursiva, 460
sesgada, 460
sesgo, 676
DLT (Digital Linear Tape), 383
documentos sin estructurar, 632
DOM (Document Object Model), 265, 349
domain type, 103
domiciliación, 613
dominios, 29, 63, 140, 175
atómicos, 30, 302
drop trigger, 276
DSL, 667
DTD (Document Type Definition), 335
DUNS, 747
duplicación
instantánea, 907
por mezcla, 907
transaccional, 907
durabilidad, 18, 508, 509, 512, 567
grados
dos muy seguro, 593
dos seguro, 593
uno seguro, 593
DVD (Digital Video Disk), 368

EBCDIC, 717
ejecuciones no secuenciables, 792
ejemplares, 6, 154
básico de una regla, 154
de la base de datos, 31
de la relación, 31, 173
ejemplos de formación, 616
elección, 711
elemento, 332, 656
eliminación de duplicados, 145
paralela, 684
elipses, 756
encauzamiento
bajo demanda, 468
por los productores, 468
Encina, 777, 781
enfoque incremental, 749
entidad, 171
de procesamiento, 781
entornos distribuidos, 704
entradas, 720
de construcción, 459
de prueba, 459
envolturas, 718, 748
equipos asociados, 474
equirreunión, 452
equivalentes, 476
ERP, 785
error
del sistema, 567
lógico, 567
es/son, 639
escribir, operación, 793
escrituras
a ciegas, 520
externas observables, 511
espacio de intercambio, 583
espacio de nombres, 334
predeterminado, 334
espacios de tablas, 837
especialización, 190, 191, 308
parcial, 194
total, 194
esperar, 530
esperar--morir, 549
esqueletos de tablas, 144, 149
resultado, 149
esquemas
único de relación, 149
de la base de datos, 31
de la relación, 31
definición, 6, 63
físico, 6
lógico, 6
estabilidad del cursor, 556
estaciones para el soporte de movilidad, 768
estadísticas y estimación del coste, 502
estado de ejecución, 468, 783
estados
de terminación
aceptables, 783
no aceptables, 783
inconsistente, 509
preparado, 699
estrategia
de extracción inmediata, 385

estrategia (*cont.*)
de gestión de la memoria intermedia, 385
de semirreunión, 716
de sustitución, 384
más recientemente utilizado, 385
estrella, esquema, 614
estructura
de índice ordenado, 420
de páginas con ranuras, 389, 389
etiquetas, 329
evaluación
correlacionada, 493
encauzada, 467
materializada, 467
evento-condición-acción, 274
evitación del desbordamiento, 461
except, 71
excepto, operación, 72
exceso de ajuste, 619
exclusión mutua, 655
expansión de vistas, 83, 156
exploración
de la relación, 490, 674
del índice, 447, 490
sólo del índice, 490
explorador de archivo, 446
exposición de datos no comprometidos, 791
expresión de ruta, 341
expresiones
de camino, 315
extensiones de clases, 319
extensiones persistentes, 323
extracción, 468, 614
extracción de información, sistemas, 642

factor de escape, 461
fallo
durante una transferencia de datos, 569
en el sistema, 512
falso negativo, 640
falso positivo, 640
fantasma, fenómeno, 554, 559
fase de diseño
físico, 12, 170
lógico, 12, 170
fases
crecimiento, 533
decrecimiento, 533
deshacer, 587
escritura, 543
lectura, 543
rehacer, 587
validación, 543
fiabilidad, 375
Fibre Channel, 373
fila, 30
filas de ejemplo, 144
FireWire, 371
firmas digitales, 288
Flashback, 850
flujos de trabajo, 703, 783
definición, 781
ejecución, 782
especificación, 782, 783
insegura, 785
estado, 783
recuperación, 785
sistema gestor, 784
arquitectura centralizada, 784
arquitectura completamente distribuida, 784
arquitectura parcialmente distribuida, 784
FLWOR, 343
FNBC, *véase* forma normal, de Boyce-Codd
FNDC, *véase* forma normal, de dominios y claves
foreign key, 107
forma normal, 219
cuarta, 244, 246
de Boyce-Codd, 226
de dominios y claves, 248
de reunión por proyección, 248
primera, 224, 302
quinta, 248
segunda, 229, 248, 255
tercera, 229
algoritmo de síntesis, 242
Forms de Oracle, 260
Fortran, 62, 112
FoxPro, 59
fragmentación
de los datos, 695
horizontal, 695, 695
vertical, 695, 695
fragmentos y réplicas, 681, 683
frecuencia del término, 633
frecuencia inversa de los documentos, 633
from, 67, 311
función
de asociación, 421
funciones
constructoras, 306
de agregación no-descomponibles, 607
de tabla, 121
fusión, 558
fusionar, 411

ganancia de información, 618
ganancia de velocidad, 658
lineal, 658
sublineal, 658
generación impaciente de tuplas, 468
generación interactiva de índices, 688
generalización, 191, 308
de solapamiento, 194
parcial, 194
sobre la condición de disjunción, 194
total, 194
gestor
de bloqueos, *véase* bloqueos, gestor
de colas, 780
de control de concurrencia, 18
de la memoria intermedia, 384
de recursos, 780
de transacciones, 697
getConnection, 117
Global Trade Item Number, 747
GNU, 808
Google, 357
google.com, 266
GPS (Global Positioning System), 759
GQBE, 150
grafo
de autorización, 280
de espera, 551
de la base de datos, 538
de precedencia, 522, 525
dirigido acíclico, 538
global de espera, 709
local de espera, 708
gran explosión, enfoque, 749
grant, 111, 280
granularidad, 544
múltiple, 544
Graphical Query-By-Example (GQBE), 150
Grupo de administración de objetos, 747
Grupo de gestión de bases de datos de objetos, 747
grupos, 50
GTIN, 747
guiones en el lado del servidor, 270

hebra, 654
hecho, 153
herencia
única, 193
de los atributos, 193
múltiple, 193, 307
herir-esperar, 549
heurísticas, 502
Himalaya de Compaq, 687
Hinson, Gerald, 885
hipercubo, 660
hipervínculos, 262
histograma, 483, 676
de equianchura, 483
de equiprofundidad, 483
hojas de estilo, 264, 347
en cascada, 264
homónimos, 638
hora universal coordinada, 755
HP-UX, 859
HTML (Hyper-Text Markup Language), 329
HTML-DB, 260
HTTP (Hyper-Text Transfer Protocol), 263, 748
HTTPS (Hyper-Text Transfer Protocol Secure), 289

IDE (Integrated Drive Electronics), 371
idempotente, 574
identificador del elemento de datos, 572
identificadores lógicos de fila, 839
IDF (Inverse Document Frequency), 633
IDL (Interface Description Language), 747
id_tupla, 695

IEEE (Institute of Electrical and Electronics Engineers), 745
inanición, 533, 550, 552
inconsistencia de los datos, 3
incrementar, 563
independencia
  de recuperación, 591
  física respecto de los datos, 6
indexación ordenada
  esquemas, 431
índices, 401
  árbol B, 417
    algoritmo de control de concurrencia, 585
    enlazados, 558
  árbol B+, 408, 415, 431, 561
    borrado, 411
    inserción, 411
    redistribuir, 412
  árbol R, 420
  asociativo, 401, 421, 425
    construcción, 459
    prueba, 459
  compuesto, 449
  con agrupación, 402
  construcción ascendente, 439
  datos espaciales, 761
  de cobertura, 420
  de función, 352
  de ganancia de información, 618
  de proyección, 441
  denso, 403
  disperso, 403
  documentos, 639
  entrada, 403
  espacio adicional requerido, 402
  GiST, 816, 824, 826
  invertido, 639
  mapas de bits, 433
    de existencia, 434
    intersección, 433
  multinivel, 405
  ordenado, 401
  por capas de bits, 441
  primario, 402
  secuencial, 408, 414, 431
  secundario, 402
  selección, 498
  sin agrupación, 402
  tiempo
    de acceso, 402
    de borrado, 402
    de inserción, 402
  tipos de acceso, 402
inferir, 155
informática móvil, 768
información geométrica, 756
informar, 743
informes de invalidación, 771
Informix, 59
ingeniería inversa, 748
Ingres, 23, 59, 807
inicio de sesión único, 289
inserción, 55, 85, 468
inserción y borrado, 558
insert, 64, 887
insertar, 553
instantáneas, 251, 854
  actualizable, 854
  sólo de lectura, 854
instrucciones atómicas, 655
int, 102
integridad, 61
  referencial, 107
interactivamente, 688
interbloqueos, 532, 546, 548, 565
  detección, 549, 551, 708
    centralizada, 709
  prevención, 549, 708
  recuperación, 549, 552
intercambio de operadores, 687
intercambio en caliente, 381
interfaz
  de nivel de llamada, 746
  de pasarela común, 265
  del gestor de recursos, 780
  para programas de aplicación, 115
interferencia, 659, 742
Internet, 667
intersección, operación, 44, 72, 464
intersect, 71
interval, 755
Inverse Document Frequency (IDF), 633
IPv4, 812
IPv6, 812
ISO (International Organization for Standardization), 61, 720, 745
iterador, 468, 829

J2EE (Java 2 Enterprise Edition), 270, 834
Jakobsson, Hakan, 833
Java, 11, 62, 112, 265, 507
Java Database Objects (JDO), 321
Java Server Pages, 270
Javascript, 265
JDBC (Java Database Connectivity), 11, 62, 104, 115, 117, 265, 316, 347, 653, 835
JDeveloper, 834
JDO, 321
jerarquías, 127, 193, 605
  de clasificación, 643
JPEG (Joint Picture Experts Group), 766
JScript, 270, 747
JSP (Java Server Pages), 260
jukebox, 368, 382

Kerberos, 289
Krishnamurthy, Sailesh, 807

límite de tiempo, 550
línea de suscriptor digital, 667
LAN, 666
LDAP (Lightweight Directory Access Protocol), 719, 720
LDAP Data Interchange Format, 721
LDD, *véase* lenguajes, de definición de datos
LDIF, 721
Least Recently Used, 384, 385
leer, 508
leer uno, escribir todos, 712
leer uno, escribir todos los disponibles, 712
leer, operación, 793
lenguajes
  de almacenamiento y definición de datos, 8
  de consultas, 8, 35
    incorporado, 316
    no procedimental, 137
    temporales, 755
  de definición de datos, 7, 8, 61, 62
  de descripción de interfaces, 747
  de guiones, 265
    del lado del cliente, 265
  de manipulación de datos, 7, 61
  de marcas, 329
    de ColdFusion, 270
    de hipertexto, 329
    extensible, 7, 329
    inalámbrico, 769
  de marcas de hipertexto, 262
  de modelado unificado, 210
  de programación persistentes, 301, 316, 322
  estándar generalizado de marcas, 329
  no procedimentales, 36
  procedimentales, 35
limpieza de los datos, 613
línea
  poligonal, 756
  quebrada, 756
Linux, 807, 859
lista libre, 388
lista-deshacer, 580
lista-rehacer, 580
literal
  negativo, 153
  positivo, 153
little-endian, 718
llamada a procedimientos remotos, 781
LMD, *véase* lenguajes, de manipulación de datos
Loader, 855
Local Area Network, 666
localtime, 102
lógica de negocio, 21
Lotus Notes, 668
LRU, 385

MAC (Media Access Control), 812
Macromedia Flash, 265
Macromedia Shockwave, 265
malla, 660
manejadores, 124
máquina paralela
  grano fino, 653, 657
  grano grueso, 657
  masivamente paralela, 657
marca, 329
marca-temporal-E, 540, 564
marca-temporal-L, 540
marcas temporales
  esquema multiversión, 547
materializa, 466

media armónica, 742
mediadora, aplicación, 358
mediadores, sistemas, 718
medidas
  de Gini, 617
  de la entropía, 617
mejor partición, 618
memoria
  compartida, 661, 661
  flash, 367
  intermedia, 384
    de disco, 570
    de escritura no volátil, 374
    doble, 467
    forzar la salida, 571
  no volátil de acceso aleatorio, 374
  principal, 367
  virtual, 582
    distribuida, 662
menos recientemente utilizado, 384
mensajería persistente, 665
  implementación, 703
mensajes persistentes, 702
merge, 131
metadatos, 9
métodos, 305
métrica compuesta
  consultas por hora, 744
  precio/rendimiento, 744
mezcla de N vías, 450
mezcla-purga, operación, 613
microcomputadora, 767
minería de datos, 18, 602, 615
minería de texto, 624
modelo
  de árbol, 340
  de alambres, 758
  de datos
    de red, 7
    definición, 6
    entidad-relación, 13, 171
    jerárquico, 7
    orientado a objetos, 15
    relacional, 29, 673
    relacional orientado a objetos, 16
  de espacio vectorial, 634
  de objetos componentes, 908
  de objetos documento, 265, 349
  de proceso por cliente, 777
  de recorrido aleatorio, 636
  de servidor único, 778
  de simulación del rendimiento, 741
  de varios servidores y un solo encaminador, 779
  de varios servidores y varios encaminadores, 779
modificaciones no comprometidas, 575
módulo de almacenamiento persistente, 122
momento de la transacción, 251
monótona, 161
monitores de procesamiento de transacciones, 777
Most Recently Used, 385
MP3, 766
MPEG (Moving Picture Experts Group), 766
multiconjuntos, 49, 309
multienhebramiento, 778
multiprogramación, 514
multitarea, 778
MVS, 859
Myers, Dirk, 885

número de secuencia del registro histórico (NSR), 588, 589
NAS (Network Attached Storage), 372
Netscape, 270
NetWare, 778
NFS (Network File System), 372
niveles de consistencia
  compromiso de lectura, 557
  lectura repetible, 557
  secuenciable, 556
  sin compromiso de lectura, 557
nodos, 340, 637, 663
nombre distinguido, 720
  relativo, 720
normas, 744
  801.11, 769
  802.16, 769
  anticipativas, 744
  de cifrado avanzado, 286
  de cifrado de datos, 286
  de facto, 744
  formales, 744
  JDBC, 117, 745
  ODBC, 745, 746
  reaccionarias, 744
  SQL, 435
    opcionales, 890
  X/Open Distributed Transaction Processing, 780
notificación, 273
Novell, 778
NSRPágina, 589
nulos, 15, 31, 176
NUMA (Nonuniform Memory Architecture), 663
numeración de versiones, 771
NVRAM (Nonvolatile Random-Access Memory), *véase* memoria, no volátil de acceso aleatorio

ObjectStore, 319
objeto hueco, 322
Objetos de bases de datos de Java (JDO), 321
ODBC (Open Database Connectivity), 11, 62, 115, 265, 316, 653, 904
ODMG (Object Database Management Group), 319, 320, 747
OLE-DB, 746, 899
OMA (Object Management Architecture), 747
OMG (Object Management Group), 747
Online Analytical Processing, 742
Online Transaction Processing, 742
ontologías, 639
OO1, 744
OO7, 744
Open Directory Project, 645
operaciones
  coste de la evaluación paralela, 684
  deshacer, 575, 576
    física, 585
    lógica, 585
  entrada, 570
  escribir, 571
  lógicas, 585
  leer, 571
  rehacer, 574, 576
    fisiológicas, 588
  relacionales, 157
  salida, 570
operadores relacionales, 70
optimización
  de consultas, 445, 475
    global, 718
  heurística, 491
optimizadores basados en el coste, 488
Oracle, 2, 23, 59, 121, 285, 327, 352, 399, 492, 597, 603, 678, 740, 789, 833
ORB (Object Request Broker), 747
orden
  de secuencialidad, 523
  interesante, 490
  lexicográfico, 419
  más significativo, 718
  menos significativo, 718
ordenación
  con división por rangos, 679
  externa, 450
  paralela, 679
  por marcas temporales, 540
  topológica, 523
ordenación y mezcla externas paralelas, 679
ordenación-mezcla externa, 450
order by, 70
organización de archivos en agrupaciones de varias tablas, 390, 392
OS X, 807
OS/400, 859

páginas amarillas, 719
páginas blancas, 719
páginas Web activas, 264
Padmanabhan, Sriram, 859
PageRank, 636
paginación en la sombra, 796
palabras clave, 631
palabras de parada, 633
Papadimitriou, Spiros, 807
parámetros ajustables, 735
paralelismo
  de datos, 679
  de encauzamiento, 685
  de grano grueso, 652
  en consultas, 678
  en operaciones, 678, 679
  entre consultas, 677
  entre operaciones, 678, 685
  independiente, 686

paridad distribuida con bloques entrelazados, 379
parte-de, 639
particiones, 698
  binarias, 618
  múltiples, 618
participación, 173
  parcial y total, 179
Pascal, 62, 112
pasos, 781
PATA (Parallel ATA), 371, 373
patrones descriptivos, 615
perezosa, generación, 468
periodo de validez, 251
Perl, 270, 807, 809, 817
persistencia de los objetos, 317
pertenencia
  a conjuntos, 76
  definida por el atributo, 193
  definida por el usuario, 193
  definida por la condición, 193
pestillos, 583
petabyte, 369
PHP, 270, 809
Pirzada, Vaqar, 885
pistas, 370
pivotaje, 605
PL/I, 62, 112
planes sólo de índices, 502
planificaciones, 514
  consultas paralelas, 686
  equivalente
    en cuanto a conflictos, 518
    en cuanto a vistas, 520
  legal, 533
  recuperable, 521, 521
  secuenciable
    en cuanto a conflictos, 519
    en cuanto a vistas, 520
  secuencial, 514
  sin cascada, 521, 521
plantillas, 347
plato, 370
población, 621
polígonos, 756
posee, 189
PostgreSQL, 807
postmaster, 830
PowerBuilder de Sybase, 260
precede, 533
precisión, 640
predicción, 615
preextracción, 656
prefs, 266
preparada, 665
primitivas de evaluación, 444
privilegios, 111
  concesión, 279
  execute, 279
  references, 279
    de SQL, 281
  usage, 279
problema de Halloween, 504
procesadores
  local, 677
  virtual, 676
procesamiento
  en conexión
    analítico, 742
    de transacciones, 742
proceso
  escritor de bases de datos, 654
  escritor del registro, 655
  gestor de bloqueos, 654
  monitor de procesos, 655
  punto de revisión, 655
  servidor, 654
proceso de entrega de mensajes, 703
productividad, 513, 657, 742
producto cartesiano, 30, 36, 39
programas de aplicación, 11
Prolog, 137, 151, 162
propagación perezosa, 708
protocolos
  basados en grafos, 792
  basados en marcas temporales, 792
  de acceso a directorios, 719
  de acceso a directorios X.500, 720
  de aplicaciones inalámbrico, 769
  de bloqueo, 704
  de compromiso, 698, 699
    de dos fases, 665, 698, 699
    de tres fases, 698, 701
    en grupo, 789, 790
  de control de concurrencia, 677
  de mayoría, 705
  de ordenación por marcas temporales, 540
  de transferencia de hipertexto, 265
  de validación, 792
  globales-lectura, 798
  globales-lectura-escritura/locales-lectura, 798
  ligero de acceso a directorios, 719, 720
  locales-lectura, 798
  sesgado, 706
  simple de acceso a objetos, 357, 748
proximidad, 634
proyección, 37
  generalizada, 48
  paralela, 684
  temporal, 755
proyecto de directorio abierto, 645
pruebas de rendimiento, 741
  familias de tareas, 741
  OLAP, 742
  OLTP, 742
  OODB, 744
  TPC, 742
PSM (Persistent Storage Module), 122
public, 111, 281
publicador, 906
puntero persistente, 318
punto de bloqueo, 534
punto fijo, 129, 159
PuntoFijo-Datalog, 159
puntos de almacenamiento, 591
puntos de revisión, 578, 580, 586
  difuso, 580, 588
Python, 270
QBE, 27
QBE (Query-By-Example), 137, 144, 162
QMF (Query Management Facility), 165
quórum
  de escritura, 706
  de lectura, 706

R, árbol, *véase* índices, árbol R
réplica, 885
  completa, 694
  de actualización distribuida, 708
  de datos, 694
  maestro-esclavo, 707
  multimaestro, 708
raíz, 332
RAID, 569, 767
RAID (Redundant Array of Independent Disks
  paridad con bits entrelazados, 378
  paridad con bloques enrelazados, 379
RAID (Redundant Array of Independent Disks), 375
Ramos, Bill, 885
Rathakrishnan, Balaji, 885
Rdb de DEC, 23
Rdb de Oracle, 678
REA, 582
RealAudio, 766
realimentación de la relevancia, 634
reasignación de los sectores dañados, 371
recall, 640
rechazo falso, 640
recubrimiento canónico, 235, 235
recuperabilidad, 520, 770, 791
  de elementos de datos de gran tamaño, 795
recuperación, 571, 579, 701
  a un instante, 596
  al reiniciar, 580, 587
  basada en el registro histórico, 572
  basada en la semejanza, 767
  esquema, 567, 581
  optimizaciones, 591
  subsistema, 386
recuperación (recall), 640
recuperación de fallos, 18
  algoritmo, 789
recuperación de información, 631
  basada en la semejanza, 634
recursividad, 160
  estructural, 348
recursos para presentaciones, 780
redes, 767
  de área amplia, 666, 667
  de área de almacenamiento, 371, 667
  de área local, 666
  de interconexión, 659
redundancia, 3, 375
  P+Q, 380
referencia, 723
referencias cruzadas, 251
registro de escritura anticipada (REA), 582
registro histórico, 572, 796
  con memoria intermedia, 581

registro histórico (*cont.*)
de operaciones, 795
físico, 585
registros, 585
forzar, 582
lógico, 585, 795
registro de actualización, 572
registro punto de revisión, 589
registros de compensación, 586, 589
registros, 572
índice, 403
reglas, 19, 151, 153, 615
aumentatividad, 232
de equivalencia, 477
conjunto mínimo, 480
de escritura de Thomas, 542
de los cinco minutos, 736
del minuto, 736
descomposición, 233
pseudotransitividad, 233
reflexividad, 232
transitividad, 232
unión, 233
regresión, 620
lineal, 620
reingeniería, 749
relaciones, 13, 30, 173
bitemporales, 754
derivadas, 80
desnormalizada, 737
externa, 452
identificadora, 189
instantáneas, 755
interna, 452
referenciada, 35
referenciante, 35
temporales, 754
uno a uno, 177
uno a varios, 177
varios a uno, 177
varios a varios, 177
relevancia, 633
relevo, 768
en caliente, 592
reloj
del sistema, 540
lógico, 707
rendimiento, 791
de la reconstrucción, 380
renombramiento, operación, 41, 68
repetición de la historia, 587
replicados, 664
representación
relacional
agregación, 206
atributos compuestos, 204
combinación de esquemas, 204
conjuntos de entidades débiles, 202
conjuntos de entidades fuertes, 201
conjuntos de relaciones, 202
generalización, 205
redundancia, 203
repudio, 288
requisitos funcionales
especificación, 12, 170
resolución del desbordamiento, 461
respuesta a las preguntas, 643
respuesta por desafío, 288
restricciones, 238, 247
de completitud, 194
de consistencia, 3, 8
de definición por condición, 195
de integridad, 105
referencial, 107
legales, 225
resultados, 783
retículo, 193
retroceso, 552
en cascada, 534
parcial, 552
total, 552
reunión, 45
con bucles anidados en paralelo, 683
con fragmentos y réplicas, 681, 682
asimétricos, 681
cruzada, 94
de banda, 689
de unión, 94
en bucle anidado, 452
indexada, 455
por bloques, 454
encauzada, 469
espacial, 760
externa, 51
completa, 53, 93
por la derecha, 52, 93
por la izquierda, 52, 93
minimización, 503
natural, 45, 92
operaciones, 92
orden en profundidad por la izquierda, 491
paralela
algoritmos, 680
por asociación, 459
híbrida, 462
por asociación dividida en paralelo, 683
por división, 680, 680
por mezcla, 455
por ordenación-mezcla, 455
temporal, 755
zeta, 46
revoke, 111, 282
REXX, 863
robustez, 710
rol
entidad, 173
RosettaNet, 356
RPC (Remote Procedure Call), 781
transaccionales, 781
Rys, Michael, 885

saga, 794
SAN (Storage Area Network), 667
SATA (Serial ATA), 371, 373
satisface, 155, 225
SAX (Simple API for XML), 350
Schroeder, Bianca, 807
SCSI (Small Computer System Interconnect), 371
sectores, 370
secuencialidad, 516
de dos niveles, 797
en cuanto a conflictos, 517
asegurar, 533
en cuanto a vistas, 517
propiedad, 529
protocolos
basados en el bloqueo, 529
basados en grafos, 537
secuencias, 450
secuencias de ejecución, 514
segmento rectilíneo, 756
seguridad, 157
selección, 36, 347
conjuntiva, 448
de reemplazamiento, 473
disyuntiva, 448
paralela, 684
temporal, 755
select, 65, 66, 313, 887, 891
selectividad, 484
semántica
programa, 155, 156
reglas, 154
semejanza del coseno, 634
Sequel, 61
Server-Side Javascript, 270
servicio Web, 357
servidores, 20
de aplicaciones, 21
de nombres, 696
Web, 265, 779
servlet, 267
sesgo, 659, 681, 688
de ejecución, 675, 680
de los valores de los atributos, 676
en una relación, 675
SET (Secure Electronic Transaction), 788
SGBD, 1
SGML (Standard Generalized Markup Language), 329
showplan, 887
SIGMOD, conferencia de la ACM, 807
sin compartimiento, 662
sin conexión, 266
sinónimos, 638
sintaxis
bidimensional, 144
sistema
dividido, 698
sistema de archivos, 386
sistema de bases de datos
múltiples, 717
sistema de posicionamiento global, 759
sistema de procesamiento de archivos, 3
sistema gestor de bases de datos, 1
sistemas
clientes, 653
de archivos
de diario, 374
de colas, 734

sistemas (*cont.*)
de discos compartidos, 667
de diseño asistido por computadora, 757
de gestión de flujos de trabajo, 665
de información geográfica, 756
de planificación de los recursos de las empresas, 785
de tiempo real, 790
heredados, 748
monousuario, 652
multiusuario, 652
paralelos, 657
persistentes de C++, 319
relacionales, 323
remotos de copia de seguridad, 595
servidores, 653
de consultas, 654
de datos, 654, 656
de transacciones, 654
sistemas de bases de datos
cliente-servidor, 651
distribuidos, 651, 663, 693
distribuidos y heterogéneos, 664
múltiples, 664, 796
paralelos, 651, 673
sitio, 663
principal, 591, 705
remoto de copia de seguridad, 591
secundario, 591
smallint, 102
SOAP, 357
SOAP (Simple Object Access Protocol), 748
sobrecarga, 319
software envolvente, 358
Solaris, 859
soporte, 621
span, 755
SQL, 61, 323, 507
dinámico, 62
dominios
char, 63
double precision, 63
float, 63
int, 63
numeric, 63
real, 63
smallint, 63
varchar, 63
incorporado, 62, 112
SQL Server de Microsoft, 59
SQL-92, 62
Standard Generalized Markup Language, 329
Storage Area Network, 371
subárboles disjuntos, 538
subastas, 787
inversas, 787
subclase, 192
subconsultas, 78
escalares, 130
subesquemas, 6
subexpresiones comunes, 503
subtabla, 308
subtareas, 791
sufijo, 722
sumas de comprobación, 371
Sun Microsystems, 321, 745
superclase, 192
superclase-subclase, 191
superclave, 34, 178, 225
superusuario, 279
supuesto de fallo-parada, 567
suscriptores, 907
sustitución de bloques utilizados menos recientemente, 385
Sybase, 59
synset, 639
System R, 23, 59, 61, 474, 491, 527, 597, 828, 859

término, 632
tabla de páginas desfasadas, 588
TablaPáginasDesfasadas, 589
tablas, 29, 30
de dimensiones, 614
de hechos, 614
dinámica, 603
tabulación cruzada, 603
tag library, 271
tarea, 781
tarjetas inteligentes, 288
Tcl, 808
teléfonos móviles, 184, 768
teoría de colas, 735
terabyte, 369
Teradata, 687
Term Frequency (TF), 633
terminales, 767
test
de potencia, 744
de productividad, 744
texto completo, 632
TF (Term Frequency), 633
Thomas, *véase* reglas, de escritura de Thomas
tiempo
de acceso, 372
de búsqueda, 372
medio, 372
de compromiso, 592
de latencia medio, 372
de latencia rotacional, 372
de recuperación, 592
de respuesta, 657
de servicio, 741
de transacción, 754
en SQL, 754
límite
estricto, 790
firme, 790
flexible, 790
medio
de reparación, 376
de respuesta, 514
entre fallos, 372
entre pérdidas de datos, 376
para concluir, 742
válido, 754
tiempo límite, 790
tipos de datos
coerción, 102
compatibles, 102
de conjunto, 318
de fila, 305
definidos por el usuario, 304
estructurados, 102
más concreto, 307
tipos de dominios, 63, 103
Tk, 808
tolerancia ante fallos, 662
Top End, 777
TP, 777
TPS (transacciones por segundo), 743
transacciones, 90, 777
abortada, 510
activa, 510
anidada, 794
cancelar, 511
compensadora, 510, 794
comprometida, 510
concurrentes, 579
de actualización, 548
de corta duración, 792
de sólo lectura, 548
definición, 18, 507, 507
distribuida, 697
dudosas, 701
ejecución
concurrente, 513
secuencial, 513
electrónica segura, 788
fallida, 510
fallo, 567
flujo de tareas, 781
flujo de trabajo, 781
global, 663, 697, 796
instantánea consistente, 707
interactiva
compleja, 795
de larga duración, 799
larga duración, 791
llamada a procedimientos remotos, 653
local, 663, 697, 796
múltiples, 513
modificación
diferida, 573
inmediata, 575
multinivel, 794
parcialmente comprometida, 510
procesadas por minilotes, 741
reiniciar, 511
retrocedida, 510
retroceso, 579, 585
en cascada, 521
terminada, 510
Transaction Processing Performance Council, 742
transferencia
de prestigio, 636
del control, 592
transformación, 614
transformar, 613

transparencia
de la fragmentación, 696
de la réplica, 696
de la ubicación, 696
de los datos, 696
traza de auditoría, 285
triangulación, 756
tries, 440
triviales, 226
tupla, 30
Turing
máquina universal, 11
premio, 23
Tuxedo, 777

UDDI, 357
Ultrium, 383
UML (Unified Modeling Language), 210
union, 71
unión, 38, 71, 464
union join, 94
unique, 106
Universel Temps Coordoné, 755
Unix, 807, 859
update, 111, 887
upsert, 834
URL (Uniform Resource Locator), 263
USB (Universal Serial Bus), 368, 371
UTC, 755
utilización, 513

valores
anterior, 572
continuos, 618
nuevo, 572
variables
de correlación, 493
externas, 783
tupla, 30, 68
varrays, 835
VBScript, 270, 747
vector de división, 674
por rangos equilibrado, 676
vector de versiones, 771
velocidad de transferencia de datos, 372
ventana, 611
versiones, 547
multiconjunto, 70
vídeo
servidor, 767
Virtual Private Database, 284
vistas, 81, 82, 155, 283
actualizables, 89
definición, 61, 82
materializadas, 83, 494, 503
mantenimiento, 83, 495, 613
mantenimiento diferido, 495, 738
mantenimiento incremental, 495
mantenimiento inmediato, 495, 738
selección, 498
no recursiva, 154
paramétricas, 122
recursiva, 83, 154
Visual Basic, 115, 779, 905
Visual C++, 260
visualización de datos, 624
VM, 859
volcado de archivo, 584
difuso, 585
volcar, 584
VPD, 284
VRML (Virtual Reality Markup Language), 265

Waas, Florian, 885
WAN, 667
WAP (Wireless Application Protocol), 769
Web, 262
WebSphere, 270
where, 66, 84
Wide Area Networks, 667
Windows, 859
2000, 859
XP, 859
with, 80, 94, 134
WML (Wireless Markup Language), 769
World Wide Web, 262
WSDL, Web Services Descripción Language, 357

XML (Extensible Markup Language), 329
datos relacionales publicados, 353
XPath, 340, 341
XPS de Informix, 687
XQJ, 347
XQuery, 340, 343
XSLT (XSL Transformations), 340
XSLT (XSLT Transformations), 347, 347
keys, 348

Yahoo, 623, 645

z/OS, 859
Zope, 270
Zwilling, Michael, 885

ACERCA DEL LIBRO

La quinta edición del libro *Fundamentos de bases de datos* de los autores Silberschatz, Korth y Sudarshan ofrece todo lo necesario para una profunda comprensión de los sistemas de bases de datos. Los autores explican conceptos fundamentales de la gestión de bases de datos con todo el detalle y atención que los lectores de este clásico libro de texto esperan. Se examinan en profundidad los lenguajes de consultas, el diseño de esquemas, el desarrollo de aplicaciones, la implementación de sistemas, el análisis de datos y las arquitecturas de las bases de datos. El libro es apropiado tanto para cursos introductorios como avanzados, así como referencia para profesionales.

La quinta edición de *Fundamentos de bases de datos* conserva el estilo general de las ediciones anteriores a la vez que evoluciona su contenido y organización para reflejar los cambios que se han producido al diseñar, gestionar y utilizar las bases de datos. Algunos de los contenidos de esta nueva edición son:

- Nueva organización de los capítulos, adelantando el estudio de SQL.
- Tratamiento ampliado de SQL, incluyendo SQL:2003.
- Una nueva parte dedicada al diseño de bases de datos con un tratamiento ampliado de la normalización y los datos temporales.
- Tratamiento ampliado y actualizado de XML.
- Tratamiento detallado de la implementación y las arquitecturas de los sistemas de bases de datos.
- Tratamiento ampliado del diseño y desarrollo de aplicaciones.
- Capítulos sobre minería y análisis de datos, y de recuperación de información.
- Estudios de casos que incluyen las últimas versiones de IBM DB2, Oracle, Microsoft SQL Server y PostgreSQL (nuevo en esta edición).
- Énfasis en los aspectos prácticos, aplicaciones e implementación, acompañados de un tratamiento intuitivo de los conceptos teóricos clave.

ACERCA DE LOS AUTORES

Abraham Silberschatz (Dr. por la Universidad estatal de Nueva York en Stony Brook) es catedrático de Informática en la Universidad de Yale. Antes de trabajar para Yale, fue vicepresidente del Information Sciences Research Center en Bell Laboratories, Murray Hill, Nueva Jersey. Anteriormente fue catedrático del Departamento de Informática en la Universidad de Texas en Austin. Sus intereses en investigación incluyen los sistemas operativos, los sistemas de bases de datos, los sistemas en tiempo real, los sistemas de almacenamiento, la administración de redes y los sistemas distribuidos. El profesor Silberschatz es miembro de las sociedades ACM e IEEE. Ha recibido el premio Taylor L. Booth Education Award de la IEEE en el 2000, el premio Karl V. Karlstrom Outstanding Educator Award de la ACM en 1998, el premio SIGMOD Contribution Award de la ACM en 1997 y el premio Computer Society Outstanding Paper Award de la IEEE. También es autor del libro *Fundamentos de sistemas operativos*, publicado en castellano por McGraw-Hill.

Henry F. Korth (Dr. por la Universidad de Princeton) es catedrático Weiseman y director del departamento de Ciencias e Ingeniería Informática en la Universidad de Lehigh. Antes de trabajar para Lehigh, fue director de investigación en principios de bases de datos de Bell Laboratories, Murray Hill, Nueva Jersey. Entre sus líneas de investigación se encuentran la gestión de datos XML, datos en Web, sistemas de bases de datos en memoria principal, sistemas en tiempo real, sistemas paralelos y otros temas. Antes de trabajar en Bell Laboratories, el profesor Korth fue vicepresidente de Panasonic Technologies y director del Matsushita Information Technology Laboratory. Anteriormente ha sido catedrático asociado en el departamento de Informática en la Universidad de Texas en Austin e investigador en el T. J. Watson Research Center de IBM. El profesor Korth es miembro de las sociedades ACM e IEEE.

S. Sudarshan (Dr. por la Universidad de Wisconsin, Madison) es catedrático en el Departamento de Ciencias e Ingeniería Informática del Indian Institute of Technology (IIT) en Bombay. Anteriormente ha sido miembro del equipo técnico del grupo de investigación de bases de datos de Bell Laboratories, Murray Hill, Nueva Jersey. El profesor Sudarshan es autor de alrededor de 60 artículos en diferentes áreas de los sistemas de bases de datos y tiene 13 patentes. Entre las líneas de investigación del profesor Sudarshan se encuentran el procesamiento y optimización de consultas, la autorización y la consulta bajo palabras clave en bases de datos. Además de ser arquitecto de varios sistemas software relacionados con la implementación de los sistemas de bases de datos, también ha sido responsable de construir y mantener varias aplicaciones de bases de datos utilizadas en el IIT de Bombay.

9 788448 146443

www.mcgraw-hill.es

The McGraw-Hill Companies

ISBN: 84-481-4644-1