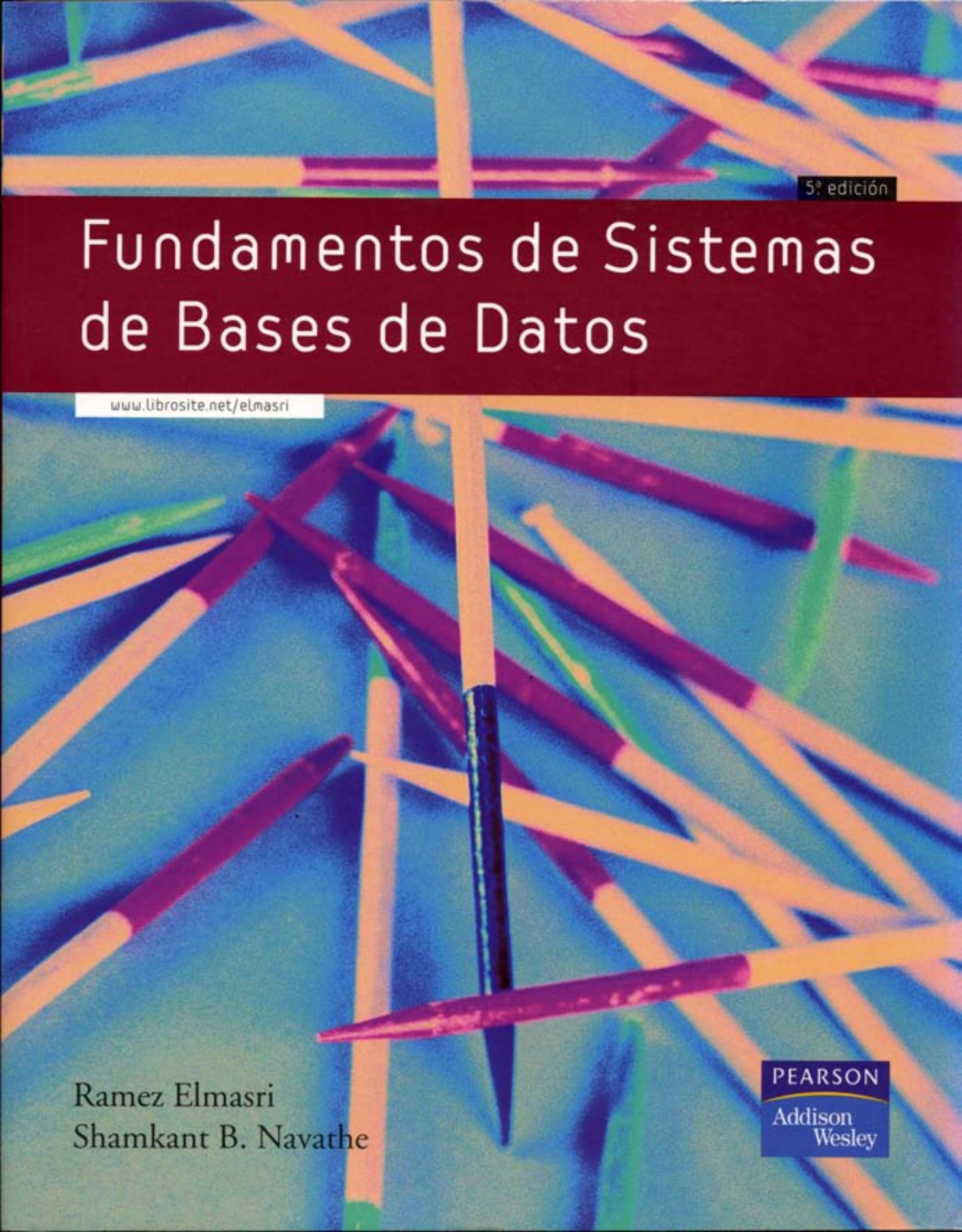

5ª edición
Fundamentos de Sistemas de Bases de Datos
www.librosite.net/elmasri
Ramez Elmasri
Shamkant B. Navathe
PEARSON
Addison Wesley

# Fundamentos de Sistemas de Bases de Datos

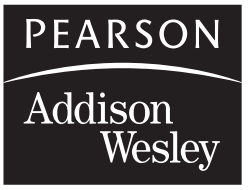
# Fundamentos de Sistemas de Bases de Datos

Quinta Edición

**RAMEZ ELMASRI**
*Department of Computer Science and Engineering*
*The University of Texas at Arlington*

**SHAMKANT B. NAVATHE**
*College of Computing*
*Georgia Institute of Technology*

**Traducción**
José Manuel Díaz




Boston ● San Francisco ● Nueva York ● Londres
Toronto ● Sydney ● Tokio ● Singapur ● Madrid ● Ciudad de México
Munich ● París ● Ciudad del Cabo ● Hong Kong ● Montreal

**Datos de catalogación bibliográfica**

**Fundamentos de Sistemas de Bases de Datos**
**Ramez Elmasri y Shamkant B. Navathe**
PEARSON EDUCACIÓN S.A., Madrid, 2007

ISBN: 978-84-7829-085-7

Materia: Informática, 004.4

Formato: 195 x 250 mm. Páginas: 1012

Todos los derechos reservados.
Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución, comunicación pública y transformación de esta obra sin contar con autorización de los titulares de propiedad intelectual. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual *(arts. 270 y sgts. Código Penal)*.

DERECHOS RESERVADOS
© 2007 por PEARSON EDUCACIÓN S.A.
Ribera del Loira, 28
28042 Madrid

**Fundamentos de Sistemas de Bases de Datos**
**Ramez Elmasri y Shamkant B. Navathe**

**ISBN: 978-84-7829-085-7**

Deposito Legal:
ADDISON WESLEY es un sello editorial autorizado de PEARSON EDUCACIÓN S.A.

Authorized translation from the English language edition, entitled FUNDAMENTALS OF DATABASE SYSTEMS, 5th Edition by ELMASRI, RAMEZ; NAVATHE, SHAMKANT B., published by Pearson Education, Inc, publishing as Addison Wesley, Copyright © 2007

**EQUIPO EDITORIAL**
Editor: Miguel Martín-Romo
Técnico editorial: Marta Caicoya

**EQUIPO DE PRODUCCIÓN:**
Director: José A. Clares
Técnico: Diego Marín

**Diseño de Cubierta: Equipo de diseño de Pearson Educación S.A.**

**Impreso por:**

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Este libro ha sido impreso con papel y tintas ecológicos

*A la artista, la abogada y*
*a la música de mi vida.*
*R. E.*

*A Aruna y a Amol, Manisha y*
*por su amor y apoyo.*
*S. B. N.*

# Prefacio

Este libro introduce los conceptos fundamentales necesarios para diseñar, utilizar e implementar sistemas y aplicaciones de bases de datos. Nuestra presentación acentúa los principios básicos del modelado y el diseño de una base de datos, así como los lenguajes y servicios proporcionados por los sistemas gestores de bases de datos, sin olvidar las técnicas de implementación del sistema. El libro está pensado para ser utilizado como libro de texto para un curso (de nivel principiante, medio o avanzado) sobre sistemas de bases de datos de uno o dos semestres, o como libro de referencia. Asumimos que el lector está familiarizado con los conceptos elementales sobre programación y estructura de los datos.

Empezamos en la Parte 1 con una introducción y una presentación de los conceptos y la terminología básicos, y los principios del modelado conceptual de una base de datos. Concluimos en las Partes 7 y 8 con una introducción a las tecnologías emergentes, como la minería de datos, XML, la seguridad y las bases de datos web. Por el camino (en las Partes 2 a 6) proporcionamos un tratamiento en profundidad de los aspectos más importantes de los fundamentos de las bases de datos.

En la quinta edición hemos incluido las siguientes características:

- Una organización flexible e independiente que puede ajustarse a las necesidades individuales.
- Un capítulo nuevo de introducción a las técnicas de programación en SQL para aplicaciones web utilizando PHP, el popular lenguaje de *scripting*.
- Un conjunto actualizado y ampliado de ejercicios al final de cada capítulo.
- Una explicación actualizada sobre seguridad, bases de datos móviles, GIS y la manipulación de datos en bioinformática.
- Un sitio web complementario (www.librosite.net/elmasri) que incluye datos que pueden cargarse en distintos tipos de bases de datos relacionales al objeto de conseguir unos ejercicios más realistas.
- Un sencillo intérprete de cálculo y álgebra relacionales.
- Los ejercicios propuestos al final de los capítulos (del 3 al 12) versan sobre los temas del capítulo y funcionan en combinación con las bases de datos del sitio web complementario; estos ejercicios se amplían posteriormente a medida que se explica material nuevo.
- Una revisión significativa de los suplementos, incluyendo un robusto conjunto de materiales para los profesores y los estudiantes, como diapositivas de PowerPoint, las figuras del texto y la guía del profesor con las soluciones.

# Principales diferencias con la cuarta edición

Los cambios organizativos en la quinta edición son mínimos. Las mejoras de esta edición se han centrado en los capítulos individuales. Los principales cambios son los siguientes:

- Inclusión de nuevos ejercicios de práctica y la mejora de los ejercicios propuestos al final de los capítulos (Partes 1 a 3).
- Un nuevo Capítulo 26, que es una introducción a la programación de bases de datos web utilizando el lenguaje de scripting PHP.
- Ejemplos nuevos que ilustran los algoritmos de normalización y diseño de una base de datos (Capítulos 10 y 11).
- Un Capítulo 23 actualizado sobre seguridad.
- Un Capítulo 30 revisado dedicado a las tecnologías y aplicaciones de bases de datos emergentes para reflejar lo más actual sobre bases de datos móviles, GIS y la gestión de los datos del genoma.
- Un diseño nuevo que mejora la apariencia visual de las figuras, y el uso de fuentes especiales para los atributos y los tipos de entidades que mejoran la lectura y la comprensión.

# Contenidos de la quinta edición

La Parte 1 describe los conceptos básicos necesarios para un buen entendimiento del diseño y la implementación de bases de datos, así como las técnicas de modelado conceptual utilizadas en los sistemas de bases de datos. Los Capítulos 1 y 2 son una introducción a las bases de datos, los usuarios típicos y los conceptos de DBMS, su terminología y su estructura. En el Capítulo 3 se presentan y utilizan los conceptos sobre el modelo ER (entidad-relación) y los diagramas ER para ilustrar el diseño conceptual de una base de datos. El Capítulo 4 se centra en la abstracción de los datos y los conceptos de modelado semántico de los mismos, y amplía la explicación del modelo ER para incorporar estas ideas, lo que conduce al modelo de datos EER (modelo ER mejorado) y los diagramas EER. Los conceptos presentados incluyen los tipos de subclases, la especialización, la generalización y la unión (categorías). En los Capítulos 3 y 4 también explicamos la notación UML para los diagramas de clase.

La Parte 2 describe el modelo de datos relacional y los DBMSs relacionales. El Capítulo 5 describe el modelo relacional básico, sus restricciones de integridad y las operaciones de actualización. El Capítulo 6 describe las operaciones del álgebra relacional e introduce el cálculo relacional. El Capítulo 7 explica el diseño de bases de datos relacionales utilizando el mapeado ER- y EER-a-relacional. El Capítulo 8 ofrece una panorámica detallada del lenguaje SQL, incluyendo el estándar SQL que se implementa en la mayoría de los sistemas relacionales. El Capítulo 9 abarca temas de programación en SQL, como SQLJ, JDBC y SQL/CLI.

La Parte 3 abarca varios temas relacionados con el diseño de bases de datos. Los Capítulos 10 y 11 están dedicados a los formalismos, las teorías y los algoritmos desarrollados para el diseño de bases de datos relacionales. Este material incluye los tipos de dependencias funcionales, entre otros, y las formas normales de las relaciones. En el Capítulo 10 se presentan una normalización intuitiva por pasos, mientras que en el Capítulo 11 se incluyen los algoritmos de diseño relacional con ejemplos. En este capítulo también se definen otros tipos de dependencias, como las multivalor y las de concatenación. El Capítulo 12 presenta una visión general de las diferentes fases del proceso de diseño de una base de datos para aplicaciones de tamaño medio y grande, utilizando UML.

La Parte 4 empieza con una descripción de las estructuras físicas de los ficheros y de los métodos de acceso que se utilizan en los sistemas de bases de datos. El Capítulo 13 describe los principales métodos para organizar los ficheros de registros en el disco, incluyendo la dispersión (*hashing*) estática y dinámica. El Capítulo

14 describe las técnicas de indexación para ficheros, como las estructuras de datos árbol B y árbol B+ y los ficheros rejilla. El Capítulo 15 ofrece una introducción de los fundamentos básicos del procesamiento y la optimización de consultas, mientras que el Capítulo 16 explica el diseño y la refinación de una base de datos física.

La Parte 5 explica el procesamiento de transacciones, el control de la concurrencia y las técnicas de recuperación, además de descripciones de cómo se materializan estos conceptos en SQL.

La Parte 6 ofrece una introducción global a los sistemas de bases de datos de objetos y de objetos relacionales. El Capítulo 20 introduce los conceptos de orientación a objetos. El Capítulo 21 ofrece una panorámica detallada del modelo de objeto ODMG y sus lenguajes ODL y OQL asociados. El Capítulo 22 describe cómo las bases de datos relacionales se están ampliando con el fin de incluir conceptos de orientación a objetos, y presenta las características de los sistemas de objetos relacionales, así como una visión general de algunas características del estándar SQL3 y del modelo de datos relacional anidado.

Las Partes 7 y 8 están dedicadas a temas más avanzados. El Capítulo 23 ofrece una visión general de la seguridad en las bases de datos, incluyendo el modelo de control de acceso discrecional con comandos SQL para otorgar y revocar privilegios, sin olvidar el modelo de control de acceso obligatorio con categorías de usuario y la instanciación múltiple. Se explican más en detalle las medidas de control de la seguridad, incluyendo el control del acceso, el control de la inferencia, el control del flujo y el cifrado de los datos, así como los problemas relacionados con la privacidad. El Capítulo 24 introduce varios modelos de bases de datos mejorados para aplicaciones avanzadas, como las bases de datos activas y los *triggers*, así como las bases de datos de tiempo, espaciales, multimedia y deductivas. El Capítulo 25 ofrece una introducción a las bases de datos distribuidas y la arquitectura de tres niveles cliente/servidor. El Capítulo 26 es un capítulo nuevo que introduce la programación de bases de datos web mediante PHP. El Capítulo 27 es una introducción a XML; presenta sus conceptos y compara el modelo XML con los modelos de bases de datos tradicionales. El Capítulo 28 sobre la minería de datos ofrece una visión general del proceso de minería y el descubrimiento del conocimiento, además de ofrecer una explicación breve sobre distintos métodos y herramientas comerciales. El Capítulo 29 introduce los conceptos de almacenamiento de datos. Por último, el Capítulo 30 es una introducción a las bases de datos móviles, las bases de datos multimedia, los sistemas GIS y la administración de datos del genoma en bioinformática.

El Apéndice A ofrece algunas notaciones alternativas para visualizar un esquema ER o EER conceptual, que pueden sustituirse por la notación que utilizamos nosotros, si así lo prefiere el profesor. El apéndice B ofrece algunos parámetros importantes de los discos. El Apéndice C ofrece una visión general del lenguaje de consulta gráfico QBE.

Los apéndices D y E (disponibles en el sitio web complementario del libro, www.librosite.net/elmasri) están dedicados a los sistemas de bases de datos heredados, basados en los modelos de bases de datos jerárquicos y de red. Se han utilizado durante más de treinta años como base de muchas de las aplicaciones de bases de datos comerciales y sistemas de procesamiento de transacciones, y pasarán décadas hasta que se reemplacen completamente. Consideramos que es importante que los estudiantes de bases de datos conozcan estos métodos tan longevos.

# Directrices para utilizar este libro

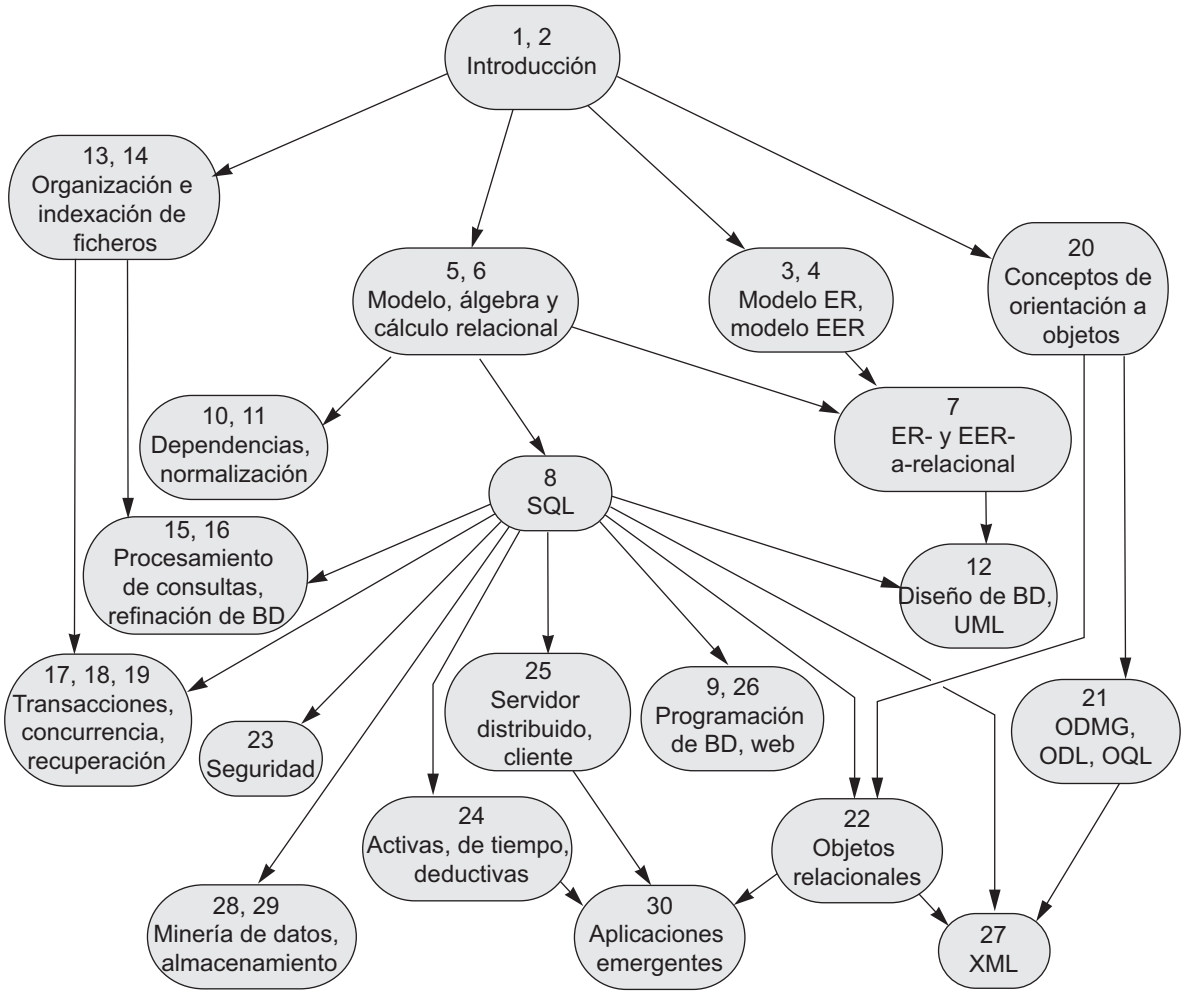
Hay muchas formas diferentes de impartir un curso de bases de datos. Los capítulos de las Partes 1 a 5 se pueden utilizar, en el orden en que aparecen o en el orden deseado, como introducción a los sistemas de bases de datos. Los capítulos y las secciones seleccionados se pueden omitir, y el profesor puede añadir otros capítulos del resto del libro, en función de los objetivos del curso. Al final de la sección inicial de cada capítulo, se enumeran las secciones candidatas a ser omitidas en caso de que se precise una explicación menos detallada del tema en cuestión. Sugerimos llegar hasta el Capítulo 14 en un curso de introducción a las bases de datos, e incluir las partes seleccionadas de otros capítulos, en función de los conocimientos de los estudiantes y de

los objetivos perseguidos. En el caso de que el curso abarque también las técnicas de implementación de sistemas, habría que incluir los capítulos de las Partes 4 y 5.

Los Capítulos 3 y 4, que abarcan el modelado conceptual mediante los modelos ER y EER, son importantes para un buen conocimiento de las bases de datos. No obstante, estos capítulos se pueden ver parcialmente, verse más tarde en el curso, u omitirse completamente si el objetivo de este último es la implementación de un DBMS. Los Capítulos 13 y 14, dedicados a la organización e indexación de ficheros, también se pueden ver más tarde o temprano en el curso, u omitirse completamente si el objetivo son los modelos de datos y los lenguajes. Los estudiantes que han completado un curso sobre organización de ficheros, ciertas partes de estos capítulos pueden considerarse como material de lectura, o pueden asignarse algunos ejercicios como un repaso de los conceptos.

Un proyecto de diseño e implementación de bases de datos completo abarca el diseño conceptual (Capítulos 3 y 4), el mapeado del modelo de datos (Capítulo 7), la normalización (Capítulo 10) y la implementación en SQL (Capítulo 9). También es preciso considerar el Capítulo 26 si el objetivo del curso abarca las aplicaciones de bases de datos web. Se precisa documentación adicional sobre los lenguajes de programación y los RDBMS utilizados.

El libro está escrito para que sea posible abarcar temas en diferentes secuencias. El gráfico de la siguiente figura muestra las principales dependencias entre los capítulos. Como el diagrama ilustra, es posible empezar



con varios temas diferentes a continuación de los dos primeros capítulos de introducción. Aunque el gráfico puede parecer complejo, es importante saber que si los capítulos se cubren en orden, las dependencias no se pierden. El gráfico lo pueden consultar los profesores que desean seguir un orden alternativo de presentación.

En un curso de un semestre basado en este libro, los capítulos seleccionados pueden asignarse como material de lectura. Las Partes 4, 7 y 8 se pueden considerar para este cometido. El libro también se puede utilizar para una secuencia de dos semestres. El primer curso, *Introducción al diseño/sistemas de bases de datos*, a un nivel de estudiante de segundo año, medio o de último año, puede cubrir la mayoría de los Capítulos 1 a 14. El segundo curso, *Técnicas de diseño e implementación de bases de datos*, a un nivel de estudiante de último año o graduado de primer año, puede abarcar los Capítulos 15 a 30. Los Capítulos de las Partes 7 y 8 se pueden utilizar selectivamente en cualquier semestre, y el material que describe el DBMS y que está disponible para los estudiantes en la institución local, se puede utilizar como complemento del material de este libro.

# Materiales suplementarios

Existe material de apoyo para todos los usuarios de este libro, así como material adicional para los profesores cualificados.

Las anotaciones de lectura y las figuras están disponibles como diapositivas de PowerPoint en el sitio web de Computer Science: *http://www.aw.com/cssupport*.

Un manual de prácticas, novedad en la quinta edición, está disponible en el sitio web complementario del libro (www.librosite.net/elmasri). Este manual abarca las herramientas de modelado de datos más populares, un intérprete de álgebra y cálculo relacional, y ejemplos del libro implementados utilizando dos sistemas de gestión de bases de datos muy difundidos. Las prácticas de la parte final de los capítulos de este libro están correlacionadas con el manual.

Los profesores cualificados tienen a su disposición un manual de soluciones. Visite el centro de recursos para profesores de Addison-Wesley (http://www.aw.com/irc), o envíe un mensaje de correo electrónico a *computing@aw.com* si desea información sobre cómo acceder a estas soluciones.

# Material de apoyo adicional

Database Place, de Addison-Wesley, contiene materiales interactivos de asistencia a los estudiantes durante sus estudios sobre modelado, normalización y SQL. Mediante un código de acceso, que se incluye con cada copia de este texto, se ofrece una suscripción complementaria a Database Place. Las suscripciones también pueden adquirirse *online*. Si desea más información, visite http://www.aw.com/databaseplace.

# Agradecimientos

Es un gran placer reconocer la ayuda y contribución de tantas personas a este proyecto. En primer lugar, queremos dar las gracias a nuestros editores, Matt Goldstein y Katherine Harutunian. En particular, queremos reconocer el esfuerzo y la ayuda de Matt Goldstein, nuestro editor principal para la quinta edición. También queremos dar las gracias a aquellas personas de Addison-Wesley que han contribuido con su esfuerzo a esta quinta edición: Michelle Brown, Gillian Hall, Patty Mahtani, Maite Suarez-Rivas, Bethany Tidd y Joyce Cosentino Wells. Estamos agradecidos a Gillian Hall por el diseño interior de esta edición y por su detallada atención sobre los estilos, las fuentes y los elementos artísticos que tan cuidadosamente ha preparado para este libro.

También queremos agradecer la contribución de los siguientes revisores: Hani Abu-Salem, *DePaul University*; Jamal R. Alsabbagh, *Grand Valley State University*; Ramzi Bualuan, *University of Notre Dame*;

Soon Chung, *Wright State University*; Sumali Conlon, *University of Mississippi*; Hasan Davulcu, *Arizona State University*; James Geller, *New Jersey Institute of Technology*; Le Gruenwald, *The University of Oklahoma*; Latifur Khan, *University of Texas at Dallas*; Herman Lam, *University of Florida*; Byung S. Lee, *University of Vermont*; Donald Sanderson, *East Tennessee State University*; Jamil Saquer, *Southwest Missouri State University*; Costas Tsatsoulis, *University of Kansas*; y Jack C. Wileden, *University of Massachusetts, Amherst*. Queremos dar las gracias a Raj Sunderraman por trabajar con nosotros en las prácticas de este libro y en el diseño de los ejercicios. Salman Azar, de la *Universidad de San Francisco*, también contribuyó con algunos ejercicios.

A Sham Navathe le gustaría dar las gracias a sus estudiantes de la Georgia Tech: Saurav Sahay, Liora Sahar, Fariborz Farahmand, Nalini Polavarapu, Wanxia Xie, Ying Liu y Gaurav Bhatia. Ed Omiecinski también ayudó con valiosas sugerencias y correcciones.

Nos gustaría repetir nuestro agradecimiento a todas esas personas que revisaron y contribuyeron con su trabajo en las ediciones anteriores de este libro:

- **Primera edición**. Alan Apt (editor), Don Batory, Seott Downing, Dennis Heimbinger, Julia Hodges, Yannis Ioannidis, Jim Larson, Dennis McLeod, Per-Ake Larson, Rahul Patel, Nicholas Roussopoulos, David Stemple, Michael Stonebraker, Frank Tompa y Kyu-YoungWhang.
- **Segunda edición**. Dan Joraanstad (editor), Rafi Ahmed, Antonio Albano, David Beeeh, Jose Blakeley, Panos Chrysanthis, Suzanne Dietrich, Vic Ghorpadey, Goets Graefe, Eric Hanson, Junguk L. Kim, Roger King, Vram Kouramajian, Vijay Kumar, John Lowther, Sanjay Manchanda, Toshimi Minoura, Inderpal Mumick, Ed Omiecinski, Girish Pathak, Raghu Ramakrishnan, Ed Robertson, Eugene Sheng, David Stotts, Marianne Winslett y Stan Zdonick.
- **Tercera edición**. Maite Suarez-Rivas y Katherine Harutunian (editoras); Suzanne Dietrieh, Ed Omiecinski, Rafi Ahmed, Francois Bancilhon, Jose Blakeley, Rick Cattell, Ann Chervenak, David W. Embley, Henry A. Etlinger, Leonidas Fegaras, Dan Forsyth, Farshad Fotouhi, Michael Franklin, Sreejith Gopinath, Goetz Craefe, Richard Hull, Sushil Jajodia, Ramesh K. Karne, Harish Kotbagi, Vijay Kumar, Tarcisio Lima, Ramon A. Mata-Toledo, Jaek MeCaw, Dennis McLeod, Rokia Missaoui, Magdi Morsi, M. Narayanaswamy, Carlos Ordonez, Joan Peckham, Betty Salzberg, Ming-Chien Shan, Junping Sun, Rajshekhar Sunderraman, Aravindan Veerasamy y Emilia E. Villareal.
- **Cuarta edición**. Maite Suarez-Rivas, Katherine Harutunian, Daniel Rausch y Juliet Silveri (editores); Phil Bernhard, Zhengxin Chen, Jan Chomicki, Hakan Ferhatosmanoglu, Len Fisk, William Hankley, Ali R. Hurson, Vijay Kumar, Peretz Shoval, Jason T. L. Wang (revisores); Ed Omiecinski (que contribuyó en el Capítulo 27); Las personas de la Universidad de Texas en Arlington que contribuyeron en esta edición fueron Hyoil Han, Babak Hojabri, Jack Fu, Charley Li, Ande Swathi y Steven Wu; Las personas de la Georgia Tech que contribuyeron en esta obra fueron Dan Forsythe, Weimin Feng, Angshuman Guin, Abrar Ul-Haque, Bin Liu, Ying Liu, Wanxia Xie y Waigen Yee.

Por último, pero no menos importante, queremos agradecer el apoyo, el ánimo y la paciencia de nuestras familias.

*R.E.*

*S.B.N.*

# Contenido


■ **Parte 1 Introducción y modelado conceptual 1**

**Capítulo 1 Bases de datos y usuarios de bases de datos 3**

1.1 Introducción 4
1.2 Un ejemplo 6
1.3 Características de la metodología de bases de datos 8
1.4 Actores de la escena 13
1.5 Trabajadores entre bambalinas 15
1.6 Ventajas de utilizar una metodología DBMS 15
1.7 Breve historia de las aplicaciones de bases de datos 20
1.8 Cuándo no usar un DBMS 23
1.9 Resumen 24
Preguntas de repaso 24
Ejercicios 25
Bibliografía seleccionada 25

**Capítulo 2 Conceptos y arquitectura de los sistemas de bases de datos 27**

2.1 Modelos de datos, esquemas e instancias 28
2.2 Arquitectura de tres esquemas e independencia de los datos 31
2.3 Lenguajes e interfaces de bases de datos 33
2.4 Entorno de un sistema de bases de datos 36
2.5 Arquitecturas cliente/servidor centralizadas para los DBMSs 40
2.6 Clasificación de los sistemas de administración de bases de datos 44
2.7 Resumen 47
Preguntas de repaso 48
Ejercicios 48
Bibliografía seleccionada 48

**Capítulo 3 Modelado de datos con el modelo Entidad-Relación (ER) 51**

3.1 Uso de modelos de datos conceptuales de alto nivel para el diseño de bases de datos 52
3.2 Un ejemplo de aplicación de base de datos 54

3.3 Tipos de entidad, conjuntos de entidades, atributos y claves 55
3.4 Tipos de relaciones, conjuntos de relaciones, roles y restricciones estructurales 61
3.5 Tipos de entidades débiles 67
3.6 Perfeccionamiento del diseño ER para la base de datos EMPRESA 68
3.7 Diagramas ER, convenciones de denominación y problemas de diseño 69
3.8 Ejemplo de otra notación: diagramas de clase UML 72
3.9 Tipos de relación con grado mayor que dos 75
3.10 Resumen 78
Preguntas de repaso 79
Ejercicios 80
Ejercicios de práctica 86
Bibliografía seleccionada 87

**Capítulo 4 El modelo Entidad-Relación mejorado (EER) 89**
4.1 Subclases, superclases y herencia 90
4.2 Especialización y generalización 91
4.3 Restricciones y características de las jerarquías de especialización y generalización 94
4.4 Modelado de tipos UNION usando categorías 100
4.5 Ejemplo EER de un esquema UNIVERSIDAD, diseños y definiciones formales 102
4.6 Ejemplo de otra notación: representación de la especialización y la generalización en diagramas de clase UML 105
4.7 Abstracción de datos, representación del conocimiento y conceptos de ontología 107
4.8 Resumen 112
Preguntas de repaso 112
Ejercicios 113
Ejercicios de práctica 119
Bibliografía seleccionada 119

**■ Parte 2 Modelo relacional: conceptos, restricciones, lenguajes, diseño y programación**

**Capítulo 5 El modelo de datos relacional y las restricciones de una base de datos relacional 123**
5.1 Conceptos del modelo relacional 124
5.2 Restricciones del modelo relacional y esquemas de bases de datos relacionales 129
5.3 Actualizaciones, transacciones y negociado de la violación de una restricción 137
5.4 Resumen 140

Preguntas de repaso 140
Ejercicios 141
Bibliografía seleccionada 144

**Capítulo 6 El álgebra relacional y los cálculos relacionales145**

6.1 Operaciones relacionales unarias: SELECCIÓN (SELECT) y PROYECCIÓN (PROJECT) 146
6.2 Operaciones de álgebra relacional de la teoría de conjuntos 151
6.3 Operaciones relacionales binarias: CONCATENACIÓN (JOIN) y DIVISIÓN (DIVISION) 155
6.4 Operaciones relacionales adicionales 162
6.6 Cálculos relacionales de tupla 169
6.7 Los cálculos relacionales de dominio 177
6.8 Resumen 179
Preguntas de repaso 180
Ejercicios 180
Ejercicios de práctica 185
Bibliografía seleccionada 186

**Capítulo 7 Diseño de bases de datos relacionales por mapeado ER- y EER-a-relacional 189**

7.1 Diseño de una base de datos relacionalutilizando el mapeado ER-a-relacional 189
7.2 Mapeado de construcciones del modelo EER a las relaciones 196
7.3 Resumen 200
Preguntas de repaso 201
Ejercicios 201
Ejercicios de práctica 202
Bibliografía seleccionada 202

**Capítulo 8 SQL-99: definición del esquema, restricciones, consultas y vistas 203**

8.1 Definición de datos y tipos de datos de SQL 205
8.2 Especificación de restricciones en SQL 209
8.3 Sentencias de SQL para cambiar el esquema 212
8.4 Consultas básicas en SQL 213
8.5 Consultas SQL más complejas 222
8.6 Sentencias INSERT, DELETE y UPDATE de SQL 235
8.7 Restricciones como aserciones y triggers 238
8.8 Vistas (tablas virtuales) en SQL 239
8.9 Características adicionales de SQL 243
8.10 Resumen 244

Preguntas de repaso 244
Ejercicios 244
Ejercicios de práctica 248
Bibliografía seleccionada 249

**Capítulo 9 Introducción a las técnicas de programación SQL 251**

9.1 Programación de bases de datos: problemas y técnicas 252
9.2 SQL incrustado, SQL dinámico y SQLJ 254
9.3 Programación de bases de datos con llamadas a funciones: SQL/CLI y JDBC 264
9.4 Procedimientos almacenados de bases de datos y SQL/PSM 272
9.5 Resumen 274
Preguntas de repaso 275
Ejercicios 275
Ejercicios de práctica 276
Bibliografía seleccionada 277

**■ Parte 3 Teoría y metodología del diseño de bases de datos 279**

**Capítulo 10 Dependencias funcionales y normalización en bases de datos relacionales 281**

10.1 Directrices de diseño informales para los esquemas de relación 282
10.2 Dependencias funcionales 291
10.3 Formas normales basadas en claves principales 298
10.4 Definiciones generales de la segunda y tercera formas normales 305
10.5 Forma normal de Boyce-Codd 308
10.6 Resumen 311
Preguntas de repaso 311
Ejercicios 312
Ejercicios de práctica 316
Bibliografía seleccionada 316

**Capítulo 11 Algoritmos de diseño de bases de datos relacionales y dependencias adicionales 317**

11.1 Propiedades de las descomposiciones relacionales 318
11.2 Algoritmos para el diseño de un esquema de base de datos relacional 323
11.3 Dependencias multivalor y cuarta forma normal 332
11.4 Dependencias de concatenación y quinta forma normal 337
11.5 Dependencias de inclusión 338
11.6 Otras dependencias y formas normales 339
11.7 Resumen 341
Preguntas de repaso 341

Ejercicios 342
Ejercicios de práctica 344
Bibliografía seleccionada 344

**Capítulo 12 Metodología práctica de diseño de bases de datos y uso de los diagramas UML 345**

12.1 El papel de los sistemas de información en las empresas 346
12.2 El diseño de la base de datos y el proceso de implementación 349
12.3 Uso de diagramas UML como ayuda a la especificación del diseño de la base de datos 366
12.4 Rational Rose, una herramienta de diseño basada en UML 373
12.5 Herramientas automáticas de diseño de bases de datos 379
12.6 Resumen 381
Preguntas de repaso 382
Bibliografía seleccionada 383

**■ Parte 4 Almacenamiento de datos, indexación, procesamiento de consultas y diseño físico 387**

**Capítulo 13 Almacenamiento en disco, estructuras básicas de ficheros y dispersión 389**

13.1 Introducción 390
13.2 Dispositivos de almacenamiento secundario 393
13.3 Almacenamiento de bloques en el búfer 398
13.4 Ubicación de los registros de fichero en disco 399
13.5 Operaciones sobre ficheros 403
13.6 Ficheros de registros desordenados (ficheros heap) 405
13.7 Ficheros de registros ordenados (ficheros ordenados) 406
13.8 Técnicas de dispersión 409
13.9 Otras organizaciones principales de ficheros 417
13.10 Paralelismo del acceso al disco mediante la tecnología RAID 418
13.11 Nuevos sistemas de almacenamiento 423
13.12 Resumen 424
Preguntas de repaso 425
Ejercicios 426
Bibliografía seleccionada 428

**Capítulo 14 Estructuras de indexación para los ficheros 429**

14.1 Tipos de índices ordenados de un nivel 430
14.2 Índices multinivel 438
14.3 Índices multinivel dinámicos utilizando árboles B y B1 442
14.4 Índices en claves múltiples 453

14.5 Otros tipos de índices 456
14.6 Resumen 457
Preguntas de repaso 458
Ejercicios 458
Bibliografía seleccionada 461

**Capítulo 15 Algoritmos para procesamiento y optimización de consultas 463**
15.1 Traducción de consultas SQL al álgebra relacional 465
15.2 Algoritmos para ordenación externa 466
15.3 Algoritmos para las operaciones SELECT y JOIN 468
15.4 Algoritmos para las operaciones de proyección y de conjunto 477
15.5 Implementación de las operaciones de agregación y de OUTER JOIN 478
15.6 Combinación de operaciones mediante flujos 480
15.7 Utilización de la heurística en la optimización de consultas 480
15.8 Utilización de la selectividad y la estimación de costes en la optimización de consultas 489
15.9 Revisión de la optimización de consultas en Oracle 498
15.10 Optimización semántica de consultas 499
15.11 Resumen 499
Preguntas de repaso 500
Ejercicios 500
Bibliografía seleccionada 501

**Capítulo 16 Diseño físico y refinación de la base de datos 503**
16.1 Diseño físico de las bases de datos relacionales 503
16.2 Visión general de la refinación de una base de datos en los sistemas relacionales 507
16.3 Resumen 512
Preguntas de repaso 513
Bibliografía seleccionada 513

**■ Parte 5 Conceptos del procesamiento de transacciones 515**

**Capítulo 17 Introducción a los conceptos y la teoría sobre el procesamiento de transacciones 517**
17.1 Introducción al procesamiento de transacciones 517
17.2 Conceptos de transacción y sistema 523
17.3 Propiedades deseables de las transacciones 526
17.4 Clasificación de las planificaciones en base a la recuperabilidad 527
17.5 Clasificación de las planificaciones basándose en la serialización 530
17.6 Soporte de transacciones en SQL 538
17.7 Resumen 540

Preguntas de repaso 541
Ejercicios 541
Bibliografía seleccionada 542

**Capítulo 18 Técnicas de control de la concurrencia 545**

18.1 Técnicas de bloqueo en dos fases para controlar la concurrencia 545
18.2 Control de la concurrencia basado en la ordenación de marcas de tiempo 555
18.3 Técnicas multiversión para controlar la concurrencia 557
18.4 Técnicas de control de la concurrencia optimistas (validación) 559
18.5 Granularidad de los elementos de datos y bloqueo de la granularidad múltiple 560
18.6 Uso de bloqueos para controlar la concurrencia en los índices 563
18.7 Otros problemas del control de la concurrencia 565
18.8 Resumen 566
Preguntas de repaso 567
Ejercicios 568
Bibliografía seleccionada 568

**Capítulo 19 Técnicas de recuperación de bases de datos 571**

19.1 Conceptos de recuperación 571
19.2 Técnicas de recuperación basadas en la actualización diferida 577
19.3 Técnicas de recuperación basadas en la actualización inmediata 581
19.4 Paginación en la sombra (*shadowing*) 583
19.5 Algoritmo de recuperación ARIES 584
19.6 Recuperación en sistemas multibase de datos 587
19.7 Copia de seguridad de la base de datos y recuperación ante fallos catastróficos 588
19.8 Resumen 589
Preguntas de repaso 590
Ejercicios 591
Bibliografía seleccionada 593

**■ Parte 6 Bases de datos de objetos y relacionales de objetos 595**

**Capítulo 20 Conceptos de las bases de datos de objetos 597**

20.1 Panorámica de los conceptos de orientación a objetos 598
20.2 Identidad del objeto, estructura del objeto y constructores de tipos 601
20.3 Encapsulamiento de operaciones, métodos y persistencia 604
20.4 Herencia y jerarquías de tipos y clases 610
20.5 Objetos complejos 613
20.6 Otros conceptos de orientación a objetos 615

20.7 Resumen 617
Preguntas de repaso 618
Ejercicios 618
Bibliografía seleccionada 619

**Capítulo 21 Estándares, lenguajes y diseño de bases de datos de objetos 621**

21.1 Visión general del modelo de objeto del ODMG 622
21.2 El lenguaje de definición de objetos ODL 633
21.3 El lenguaje de consulta de objetos OQL 638
21.4 Visión general de la vinculación del lenguaje C++ 645
21.5 Diseño conceptual de bases de datos de objetos 647
21.6 Resumen 649
Preguntas de repaso 650
Ejercicios 651
Bibliografía seleccionada 651

**Capítulo 22 Sistemas de objetos relacionales y relacionales extendidos 653**

22.1 Visión general de SQL y sus características objeto-relacional 654
22.2 Evolución de los modelos de datos y tendencias actuales de la tecnología de bases de datos 660
22.3 Informix Universal Server 5 661
22.4 Características objeto-relacional de Oracle 8 671
22.5 Implementación y problemas relacionados con los sistemas de tipos extendidos 673
22.6 El modelo relacional anidado 674
22.7 Resumen 676
Bibliografía seleccionada 677

**■ Parte 7 Temas avanzados: seguridad, modelación avanzada y distribución 679**

**Capítulo 23 Seguridad en las bases de datos 681**

23.1 Introducción a los temas de seguridad en las bases de datos 681
23.2 Control de acceso discrecional basado en la concesión y revocación de privilegios 685
23.3 Control de acceso obligatorio y control de acceso basado en roles para la seguridad multinivel 689
23.5 Introducción al control de flujo 696
23.6 Cifrado e infraestructuras de clave pública 697
23.7 Mantenimiento de la privacidad 699
23.8 Retos en la seguridad en las bases de datos 700

23.9 Resumen 701
Preguntas de repaso 702
Ejercicios 702
Bibliografía seleccionada 703

**Capítulo 24 Modelos de datos mejorados para aplicaciones avanzadas 705**
24.1 Conceptos de bases de datos activas y triggers 706
24.2 Conceptos de bases de datos de tiempo (temporales) 715
24.3 Bases de datos multimedia y espaciales 727
24.4 Introducción a las bases de datos deductivas 730
24.5 Resumen 742
Preguntas de repaso 743
Ejercicios 743
Bibliografía seleccionada 746

**Capítulo 25 Bases de datos distribuidas y arquitecturas cliente-servidor 749**
25.1 Conceptos de bases de datos distribuidas 750
25.2 Técnicas de fragmentación, replicación y asignación de datos para el diseño de bases de datos distribuidas 754
25.3 Tipos de sistemas de bases de datos distribuidas 759
25.4 Procesamiento de consultas en bases de datos distribuidas 762
25.5 El control de la concurrencia y la recuperación en bases de datos distribuidas 768
25.6 Una aproximación a la arquitectura cliente-servidor de tres niveles 770
25.7 Bases de datos distribuidas en Oracle 772
25.8 Resumen 775
Preguntas de repaso 775
Ejercicios 776
Bibliografía seleccionada 778

■ **Parte 8 Tecnologías emergentes 781**

**Capítulo 26 Programación de una base de datos web usando PHP 783**
26.1 Datos estructurados, semiestructurados y no estructurados 784
26.2 Un sencillo ejemplo PHP 788
26.3 Visión general de las características básicas de PHP 790
26.4 Visión general de la programación de bases de datos PHP 795
26.5 Resumen 799
Preguntas de repaso 799
Ejercicios 800
Ejercicios de práctica 800
Bibliografía seleccionada 801

**Capítulo 27 XML: Lenguaje de marcado extensible 803**
27.1 Modelo de datos jerárquico (árbol) de XML 803
27.2 Documentos XML, DTD y XML Schema 805
27.3 Documentos XML y bases de datos 813
27.4 Consulta XML 819
27.5 Resumen 821
Preguntas de repaso 821
Ejercicios 821
Bibliografía seleccionada 822

**Capítulo 28 Conceptos de minería de datos 823**
28.1 Repaso a la tecnología de minería de datos 823
28.2 Reglas de asociación 827
28.3 Clasificación 836
28.4 Agrupamiento 839
28.5 Planteamiento de otras cuestiones en minería de datos 841
28.6 Aplicaciones de la minería de datos 844
28.7 Herramientas comerciales de minería de datos 844
28.8 Resumen 847
Preguntas de repaso 847
Ejercicios 847
Bibliografía seleccionada 849

**Capítulo 29 Visión general del almacenamiento de datos y OLAP 851**
29.1 Introducción, definiciones y terminología 851
29.2 Características de los almacenes de datos 852
29.3 Modelado de datos para los almacenes 854
29.4 Construcción de una almacén de datos 858
29.5 Funcionalidad típica de un almacén de datos 861
29.6 Almacenes de datos frente a vistas 861
29.7 Problemas y problemas abiertos en los almacenes de datos 862
29.8 Resumen 864
Preguntas de repaso 864
Bibliografía seleccionada 864

**Capítulo 30 Tecnologías y aplicaciones emergentes de bases de datos 865**
30.1 Bases de datos móviles 866
30.2 Bases de datos multimedia 872
30.3 GIS (Sistemas de información geográfica, Geographic Information Systems) 878
30.4 Control de los datos del genoma 889

Bibliografía seleccionada 897
Créditos 899

**Apéndice A Notaciones diagramáticas alternativas para los modelos 901**

**Apéndice B Parámetros de disco 905**

**Apéndice C Introducción al lenguaje QBE 909**

**Bibliografía seleccionada 917**

**Índice 955**

PARTE 1

# Introducción y modelado conceptual

CAPÍTULO 1

# Bases de datos y usuarios de bases de datos

Las bases de datos y los sistemas de bases de datos son un componente esencial de la vida cotidiana en la sociedad moderna. Actualmente, la mayoría de nosotros nos enfrentamos a diversas actividades que implican cierta interacción con una base de datos. Por ejemplo, ir al banco a depositar o retirar fondos, realizar una reserva en un hotel o una compañía aérea, acceder al catálogo computerizado de una biblioteca para buscar un libro, o comprar algo *online* (un juguete o un computador, por ejemplo), son actividades que implican que alguien o algún programa de computador acceda a una base de datos. Incluso la compra de productos en un supermercado, en muchos casos, provoca la actualización automática de la base de datos que mantiene el stock de la tienda.

Estas interacciones son ejemplos de lo que podemos llamar **aplicaciones de bases de datos tradicionales**, en las que la mayor parte de la información que hay almacenada y a la que se accede es textual o numérica. En los últimos años, los avances en la tecnología han conducido a excitantes aplicaciones y sistemas de bases de datos nuevos. La tecnología de los medios de comunicación nuevos hace posible almacenar digitalmente imágenes, clips de audio y flujos (*streams*) de vídeo. Estos tipos de archivos se están convirtiendo en un componente importante de las **bases de datos multimedia**. Los **sistemas de información geográfica (GIS, *Geographic information systems*)** pueden almacenar y analizar mapas, datos meteorológicos e imágenes de satélite. Los **almacenes de datos** y los sistemas de **procesamiento analítico en línea** (**OLAP**, *online analytical processing*) se utilizan en muchas compañías para extraer y analizar información útil de bases de datos mucho más grandes para permitir la toma de decisiones. Las tecnologías de **tiempo real** y **bases de datos activas** se utilizan para controlar procesos industriales y de fabricación. Y las técnicas de búsqueda en las bases de datos se están aplicando a la WWW para mejorar la búsqueda de la información que los usuarios necesitan para navegar por Internet.

No obstante, para entender los fundamentos de la tecnología de bases de datos debemos empezar por los principios básicos de las aplicaciones de bases de datos tradicionales. En la Sección 1.1 definiremos una base de datos y, a continuación, explicaremos otros términos básicos. En la Sección 1.2 ofrecemos un ejemplo de bases de datos sencillo, UNIVERSIDAD, a fin de ilustrar nuestra explicación. La Sección 1.3 describe algunas de las características principales de los sistemas de bases de datos, y las Secciones 1.4 y 1.5 clasifican los tipos de personal cuyos trabajos implican el uso e interacción con sistemas de bases de datos. Las Secciones 1.6 a 1.8 ofrecen una explicación más completa de las diferentes capacidades que los sistemas de bases de datos ofrecen y explican algunas aplicaciones de bases de datos típicas. La Sección 1.9 es un resumen del capítulo.

El lector que desee una introducción rápida a los sistemas de bases de datos sólo tiene que estudiar las Secciones 1.1 a 1.5, después omitir u ojear rápidamente las Secciones 1.6 a 1.8, y pasar al Capítulo 2.

# 1.1 Introducción

Las bases de datos y la tecnología de bases de datos tienen mucha culpa del uso creciente de los computadores. Es justo decir que las bases de datos juegan un papel fundamental en la mayoría de las áreas en las que se utilizan computadores, como en el ámbito empresarial, en el comercio electrónico, ingeniería, medicina, justicia, educación y bibliotecas. La expresión *base de datos* se utiliza tan a menudo que empezaremos por definir su significado. Nuestra primera definición es muy general.

Una **base de datos** es una colección de datos relacionados. Con la palabra **datos** nos referimos a los hechos (datos) conocidos que se pueden grabar y que tienen un significado implícito. Por ejemplo, piense en los nombres, números de teléfono y direcciones de las personas que conoce. Puede tener todos estos datos grabados en un libro de direcciones indexado o los puede tener almacenados en el disco duro de un computador mediante una aplicación como Microsoft Access o Excel. Esta colección de datos relacionados con un significado implícito es una base de datos.

La definición anterior de base de datos es muy genérica; por ejemplo, podemos pensar que la colección de palabras que compone esta página de texto es una colección de datos relacionados y que, por tanto, constituye una base de datos. No obstante, el uso común del término *base de datos* es normalmente más restringido. Una base de datos tiene las siguientes propiedades implícitas:

- Una base de datos representa algún aspecto del mundo real, lo que en ocasiones se denomina **minimundo** o **universo de discurso (UoD, *Universe of discourse*)**. Los cambios introducidos en el minimundo se reflejan en la base de datos.
- Una base de datos es una colección de datos lógicamente coherente con algún tipo de significado inherente. No es correcto denominar base de datos a un surtido aleatorio de datos.
- Una base de datos se diseña, construye y rellena con datos para un propósito específico. Dispone de un grupo pretendido de usuarios y algunas aplicaciones preconcebidas en las que esos usuarios están interesados.

En otras palabras, una base de datos tiene algún origen del que se derivan los datos, algún grado de interacción con eventos del mundo real y un público que está activamente interesado en su contenido. Los usuarios finales de una base de datos pueden efectuar transacciones comerciales (por ejemplo, un cliente que compra una cámara) o se pueden producir unos eventos (por ejemplo, un empleado tiene un hijo) que provoquen un cambio en la información almacenada en la base de datos. Al objeto de que una base de datos sea en todo momento precisa y fiable, debe ser un reflejo exacto del minimundo que representa; por consiguiente, en la base de datos deben reflejarse los cambios tan pronto como sea posible.

Una base de datos puede ser de cualquier tamaño y complejidad. Por ejemplo, la lista de nombres y direcciones a la que nos referíamos anteriormente puede constar de únicamente unos cuantos cientos de registros, cada uno de ellos con una estructura sencilla. Por el contrario, el catálogo computerizado de una gran biblioteca puede contener medio millón de entradas organizadas en diferentes categorías (por los apellidos del autor principal, por el tema, por el título del libro), y cada categoría ordenada alfabéticamente. El Departamento de tesorería de Estados Unidos (IRS, *Internal Revenue Service*) mantiene una base de datos de un tamaño y complejidad aún mayores para supervisar los formularios de impuestos presentados por los contribuyentes americanos. Si asumimos que hay 100 millones de contribuyentes y que cada uno presenta una media de cinco formularios con aproximadamente 400 caracteres de información por cada uno, tenemos una base de datos de $100 \times 10^6 \times 400$ x 5 caracteres (bytes) de información. Si el IRS conserva las tres últimas declaraciones de cada contribuyente, además de la actual, tenemos una base de datos de $8 \times 10^{11}$ bytes (800 gigabytes). Esta

inmensa cantidad de información debe organizarse y administrarse para que los usuarios puedan buscar, recuperar y actualizar los datos que necesiten. Amazon.com es un buen ejemplo de una gran base de datos comercial. Contiene datos de más de 20 millones de libros, CDs, vídeos, DVDs, juegos, ropa y otros productos. La base de datos ocupa más de 2 terabytes (un terabyte es $10^{12}$ bytes de almacenamiento) y se almacena en 200 computadores diferentes (denominados servidores). Cada día acceden a Amazon.com aproximadamente 15 millones de visitantes que utilizan la base de datos para hacer compras. La base de datos se actualiza continuamente a medida que se añaden libros y otros productos nuevos al inventario, mientras que el stock se actualiza al tiempo que se tramitan las compras. Alrededor de 100 personas son las responsables de mantener actualizada la base de datos de Amazon.

Una base de datos se puede generar y mantener manualmente o estar computerizada. Por ejemplo, el catálogo de cartas de una biblioteca es una base de datos que se puede crear y mantener de forma manual. Una base de datos computerizada se puede crear y mantener con un grupo de aplicaciones escritas específicamente para esa tarea o mediante un sistema de administración de bases de datos. En este libro sólo nos ocuparemos de las bases de datos computerizadas.

Un **sistema de administración de datos (DBMS, *database management system*)** es una colección de programas que permite a los usuarios crear y mantener una base de datos. El DBMS es un *sistema de software de propósito general* que facilita los procesos de *definición*, *construcción*, *manipulación* y *compartición* de bases de datos entre varios usuarios y aplicaciones. **Definir** una base de datos implica especificar los tipos de datos, estructuras y restricciones de los datos que se almacenarán en la base de datos. La **definición** o información descriptiva de una base de datos también se almacena en esta última en forma de catálogo o diccionario de la base de datos; es lo que se conoce como **metadatos**. La **construcción** de la base de datos es el proceso consistente en almacenar los datos en algún medio de almacenamiento controlado por el DBMS. La **manipulación** de una base de datos incluye funciones como la consulta de la base de datos para recuperar datos específicos, actualizar la base de datos para reflejar los cambios introducidos en el minimundo y generar informes a partir de los datos. **Compartir** una base de datos permite que varios usuarios y programas accedan a la base de datos de forma simultánea.

Una **aplicación** accede a la base de datos enviando consultas o solicitudes de datos al DBMS. Una **consulta**[1] normalmente provoca la recuperación de algunos datos; una **transacción** puede provocar la lectura o la escritura de algunos datos en la base de datos.

Otras funciones importantes ofrecidas por el DBMS son la *protección* de la base de datos y su *mantenimiento* durante un largo periodo de tiempo. La **protección** incluye la *protección del sistema* contra el funcionamiento defectuoso del hardware o el software (caídas) y la *protección de la seguridad* contra el acceso no autorizado o malintencionado. Una gran base de datos típica puede tener un ciclo de vida de muchos años, por lo que el DBMS debe ser capaz de **mantener** el sistema de bases de datos permitiendo que el sistema evolucione según cambian los requisitos con el tiempo.

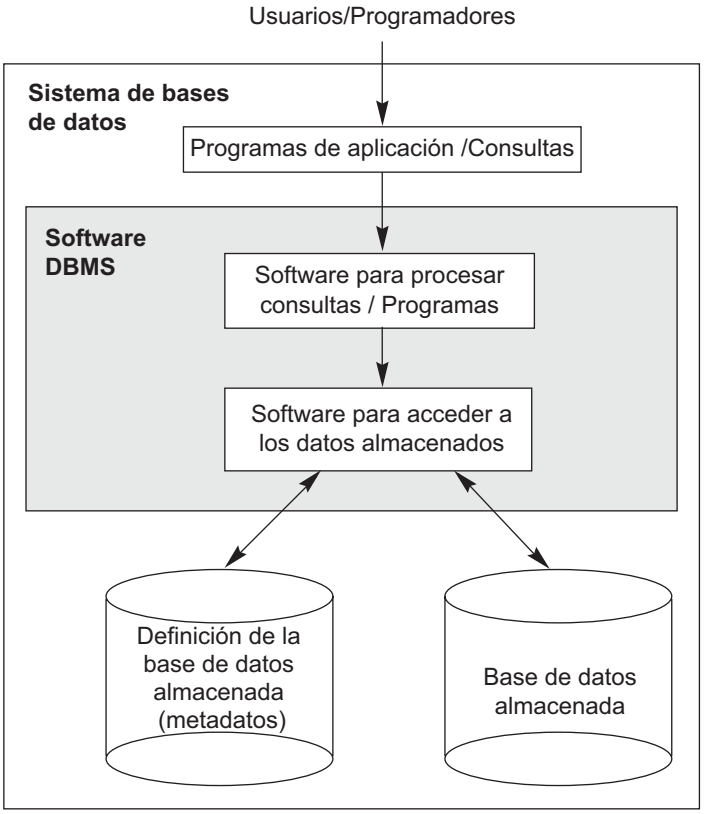
No es necesario utilizar software DBMS de propósito general para implementar una base de datos computerizada. Podríamos escribir nuestro propio conjunto de programas para crear y mantener la base de datos; en realidad, podríamos crear nuestro propio software DBMS de *propósito especial*. En cualquier caso (utilicemos o no un DBMS de propósito general), normalmente tenemos que implantar una cantidad considerable de software complejo. De hecho, la mayoría de los DBMS son sistemas de software muy complejos.

Como colofón de nuestras definiciones iniciales, denominaremos **sistema de bases de datos** a la combinación de base de datos y software DBMS. La Figura 1.1 ilustra algunos de los conceptos que hemos explicado hasta ahora.

---

[1] El término *consulta*, que inicialmente hacía referencia a una pregunta o cuestión, se utiliza ampliamente para todos los tipos de interacciones con bases de datos, incluyendo la modificación de datos.

**Figura 1.1.** Entorno de un sistema de bases de datos simplificado.




# 1.2 Un ejemplo

Vamos a ver un ejemplo con el que la mayoría de los lectores estarán familiarizados: una base de datos UNIVERSIDAD para el mantenimiento de la información relativa a los estudiantes, cursos y calificaciones en un entorno universitario. La Figura 1.2 muestra la estructura de la base de datos y algunos datos a modo de ejemplo. La base de datos está organizada en cinco archivos, cada uno de los cuales almacena registros de datos del mismo tipo.[2] El archivo ESTUDIANTE almacena los datos de todos los estudiantes, el archivo CURSO almacena los datos de todos los curso, el archivo SECCIÓN almacena los datos de las secciones de un curso, el archivo INFORME_CALIF almacena las calificaciones que los estudiantes han obtenido en las distintas secciones que han completado, y el archivo PRERREQUISITO almacena los prerrequisitos de cada curso.

Para *definir* esta base de datos debemos especificar la estructura de los registros de cada archivo detallando los diferentes tipos de **elementos de datos** que se almacenarán en cada registro. En la Figura 1.2, cada registro ESTUDIANTE incluye los datos que representan el nombre, el número, la clase (como principiante o '1', estudiante de segundo año o '2', etcétera) y la especialidad (como, por ejemplo, matemáticas o 'MAT', ciencias de la computación o 'CC'); cada registro de CURSO incluye los datos que representan el nombre, el número y las horas de crédito del curso, así como el departamento que ofrece el curso; etcétera. También hay que especificar un **tipo de datos** para cada elemento de datos de un registro. Por ejemplo, podemos especificar que el Nombre de un ESTUDIANTE es una cadena de caracteres alfabéticos, que NumEstudiante es

[2] El término *archivo* lo utilizamos aquí formalmente. A un nivel conceptual, un *archivo* es una colección de registros que pueden o no estar ordenados.

**Figura 1.2.** Base de datos que almacena la información de estudiantes y cursos.

**ESTUDIANTE**

| Nombre | NumEstudiante | Clase | Especialidad |
|---|---|---|---|
| Luis | 17 | 1 | CS |
| Carlos | 8 | 2 | CS |

**CURSO**

| NombreCurso | NumCurso | Horas | Departamento |
|---|---|---|---|
| Introducción a la computación | CC1310 | 4 | CC |
| Estructuras de datos | CC3320 | 4 | CC |
| Matemáticas discretas | MAT2410 | 3 | MAT |
| Bases de datos | CC3380 | 3 | CC |

**SECCIÓN**

| IDSeccion | NumCurso | Semestre | Año | Profesor |
|---|---|---|---|---|
| 85 | MAT2410 | Otoño | 04 | Pedro |
| 92 | CC1310 | Otoño | 04 | Ana |
| 102 | CC3320 | Primavera | 05 | Elisa |
| 112 | MAT2410 | Otoño | 05 | Antonio |
| 119 | CC1310 | Otoño | 05 | Juan |
| 135 | CC3380 | Otoño | 05 | Enrique |

**INFORME_CALIF**

| NumEstudiante | IDSeccion | Nota |
|---|---|---|
| 17 | 112 | B |
| 17 | 119 | C |
| 8 | 85 | A |
| 8 | 92 | A |
| 8 | 102 | B |
| 8 | 135 | A |

**PRERREQUISITO**

| NumCurso | NumPrerrequisito |
|---|---|
| CC3380 | CC3320 |
| CC3380 | MAT2410 |
| CC3320 | CC1310 |

un entero o que la Nota de INFORME_CALIF es un solo carácter del conjunto {'A', 'B', 'C', 'D', 'F', 'I'}. También podemos utilizar un esquema de codificación para representar los valores de un elemento de datos. Por ejemplo, en la Figura 1.2 representamos la Clase de un ESTUDIANTE como 1 para los principiantes, 2 para los estudiantes de segundo año, 3 para los junior, 4 para los sénior y 5 para los estudiantes graduados.

La *construcción* de la base de datos UNIVERSIDAD se realiza almacenando los datos que representan a todos los estudiantes, cursos, secciones, informes de calificaciones y prerrequisitos a modo de registro en el archivo adecuado. Los registros de los distintos archivos se pueden relacionar. Por ejemplo, el registro correspondiente a Luis en el archivo ESTUDIANTE está relacionado con dos registros del archivo INFORME_CALIF que especifican las calificaciones de Luis en dos secciones. De forma parecida, cada registro del archivo PRERREQUISITO relaciona dos registros de curso: uno representa el curso y el otro representa el requisito previo. La mayoría de las bases de datos de medio y gran tamaño cuentan con muchos tipos de registros y tienen *muchas relaciones* entre los registros.

La *manipulación* de bases de datos implica la consulta y la actualización. A continuación tiene algunos ejemplos de consultas:

- Recuperar el certificado de estudios (listado de todos los cursos y calificaciones) de 'Luis'.
- Listado con los nombres de los estudiantes que tomaron la sección del curso 'Bases de datos' ofrecida en otoño de 2005, así como sus calificaciones en esa sección.
- Listado de los prerrequisitos del curso 'Bases de datos'.

Y estos son algunos ejemplos de actualizaciones:

- Cambiar la clase de 'Luis' a estudiante de segundo año.
- Crear una sección nueva para el curso 'Bases de datos' para este semestre.
- Introducir una nota 'A' para 'Luis' en la sección 'Bases de datos' del último semestre.

Estas consultas y modificaciones informales deben especificarse con exactitud en el lenguaje de consulta del DBMS antes de poder ser procesadas.

A estas alturas, es útil describir la base de datos como una parte de una tarea más amplia conocida como sistema de información dentro de cualquier organización. El departamento de Tecnología de la información (TI, *Information Technology*) de una empresa diseña y mantiene un sistema de información compuesto por varios computadores, sistemas de almacenamiento, aplicaciones y bases de datos. El diseño de una aplicación nueva para una base de datos existente o el diseño de una base de datos nueva empieza con una fase denominada **definición de requisitos y análisis**. Estos requisitos son documentados en detalle y transformados en un **diseño conceptual** que se puede representar y manipular mediante algunas herramientas computerizadas, de modo que en una implementación de base de datos puedan mantenerse, modificarse y transformarse fácilmente. En el Capítulo 3 introduciremos un modelo denominado Entidad-Relación que se utiliza con este propósito. El diseño después se convierte en un **diseño lógico** que se puede expresar en un modelo de datos implementado en un DBMS comercial. Del Capítulo 5 en adelante destacaremos un modelo de datos conocido como modelo de Datos relacionales. Actualmente es la metodología más popular para diseñar e implementar bases de datos utilizando DBMSs (relacionales). La etapa final es el **diseño físico**, durante la que se proporcionan especificaciones suplementarias para el almacenamiento y acceso a la base de datos. El diseño de base de datos se implementa y rellena con datos reales y se realiza un mantenimiento continuado a fin de reflejar el estado del minimundo.

## 1.3 Características de la metodología de bases de datos

Unas cuantas características distinguen la metodología de bases de datos de la metodología tradicional de programación con archivos. En el procesamiento tradicional de archivos, cada usuario define e implementa los

archivos necesarios para una aplicación concreta como parte de la programación de esa aplicación. Por ejemplo, un usuario, la *oficina de notificación de calificaciones*, puede encargarse del mantenimiento de un archivo con los estudiantes y sus calificaciones. Los programas encargados de imprimir el certificado de estudios de un estudiante e introducir nuevas calificaciones en el archivo se implementan como parte de la aplicación. Un segundo usuario, la *oficina de contabilidad*, puede encargarse del seguimiento de las cuotas de los estudiantes y sus pagos. Aunque ambos usuarios están interesados en datos relacionados con los estudiantes, cada uno mantiene archivos separados (y programas para la manipulación de esos archivos), porque cada uno requiere algunos datos que no están disponibles en los archivos del otro. Esta redundancia en la definición y el almacenamiento de datos da como resultado un derroche de espacio de almacenamiento y unos esfuerzos redundantes por mantener al día datos comunes.

En la metodología de bases de datos se mantiene un único almacén de datos, que se define una sola vez, y al que acceden varios usuarios. En los sistemas de archivos cada aplicación tiene libertad para asignar un nombre independientemente a los elementos de datos. Por el contrario, en una base de datos, los nombres o etiquetas de los datos se definen una vez, y son utilizados por consultas, transacciones y aplicaciones. Las principales características de la metodología de bases de datos frente a la metodología de procesamiento de archivos son las siguientes:

- Naturaleza autodescriptiva de un sistema de bases de datos.
- Aislamiento entre programas y datos, y abstracción de datos.
- Soporte de varias vistas de los datos.
- Compartición de datos y procesamiento de transacciones multiusuario.

Explicaremos cada una de estas características en una sección separada. En las Secciones 1.6 a 1.8 hablaremos de otras características adicionales de los sistemas de bases de datos.

### 1.3.1 Naturaleza autodescriptiva de un sistema de bases de datos

Una característica fundamental de la metodología de bases de datos es que el sistema de bases de datos no sólo contiene la propia base de datos, sino también una completa definición o descripción de la estructura de la base de datos y sus restricciones. Esta definición se almacena en el catálogo DBMS, que contiene información como la estructura de cada archivo, el tipo y el formato de almacenamiento de cada elemento de datos, y distintas restricciones de los datos. La información almacenada en el catálogo se denomina **metadatos** y describe la estructura de la base de datos principal (véase la Figura 1.1).

El software DBMS y los usuarios de la base de datos utilizan el catálogo cuando necesitan información sobre la estructura de la base de datos. Un paquete de software DBMS de propósito general no se escribe para una aplicación de base de datos específica. Por consiguiente, debe referirse al catálogo para conocer la estructura de los archivos de una base de datos específica, como el tipo y el formato de los datos a los que accederá. El software DBMS debe funcionar igual de bien con *cualquier cantidad de aplicaciones de bases de datos* (por ejemplo, la base de datos de una universidad, la base de datos de un banco o la base de datos de una empresa), siempre y cuando la definición de la base de datos esté almacenada en el catálogo.

En el procesamiento de archivos tradicional, normalmente la definición de datos forma parte de los programas de aplicación. Así pues, esas aplicaciones están restringidas a trabajar sólo con *una base de datos específica*, cuya estructura está declarada en dichas aplicaciones. Por ejemplo, una aplicación escrita en C++ puede tener declaraciones struct o class, y un programa COBOL puede tener sentencias de "*data division*" para definir sus archivos. Mientras el software de procesamiento de archivos sólo puede acceder a bases de datos específicas, el software DBMS puede acceder a distintas bases de datos extrayendo del catálogo las definiciones de las mismas y utilizando después esas definiciones.

Para el ejemplo de la Figura 1.2, el catálogo DBMS almacenará las definiciones de todos los archivos mostrados. La Figura 1.3 muestra algunas entradas de ejemplo en un catálogo de base de datos. El diseñador de

**Figura 1.3.** Ejemplo de catálogo de base de datos para la base de datos de la Figura 1.2.

**RELACIONES**

| NombreRelacion | NumDeColumnas |
|---|---|
| ESTUDIANTE | 4 |
| CURSO | 4 |
| SECCIÓN | 5 |
| INFORME_CALIF | 3 |
| PRERREQUISITO | 2 |

**COLUMNAS**

| NombreColumna | TipoDatos | PerteneceARelacion |
|---|---|---|
| Nombre | Carácter (30) | ESTUDIANTE |
| NumEstudiante | Carácter (4) | ESTUDIANTE |
| Clase | Entero (1) | ESTUDIANTE |
| Especialidad | TipoEspecialidad | ESTUDIANTE |
| NombreCurso | Carácter (10) | CURSO |
| NumCurso | XXXXNNNN | CURSO |
| .... | .... | ..... |
| .... | .... | ..... |
| .... | .... | ..... |
| NumPrerrequisito | XXXXNNNN | PRERREQUISITO |

*Nota*: TipoEspecialidad se define como un tipo enumerado con todas las especialidades conocidas. XXXXNNNN se utiliza para definir un tipo con cuatro caracteres alfanuméricos seguidos por cuatro dígitos.

la base de datos especifica estas definiciones antes de crear la base de datos y se almacenan en el catálogo. Siempre que se crea una solicitud para acceder, por ejemplo, al Nombre de un registro de ESTUDIANTE, el software DBMS recurre al catálogo para determinar la estructura del archivo ESTUDIANTE y la posición y el tamaño del elemento de datos Nombre dentro de un registro ESTUDIANTE. Por el contrario, en una aplicación de procesamiento de archivos típica, la estructura del archivo y, en caso extremo, la ubicación exacta de Nombre dentro de un registro ESTUDIANTE también están codificadas dentro de cada programa que accede a dicho elemento de datos.

## 1.3.2 Aislamiento entre programas y datos, y abstracción de datos

En el procesamiento de archivos tradicional, la estructura de los archivos de datos está incrustada en las aplicaciones, por lo que los cambios que se introducen en la estructura de un archivo pueden obligar a realizar *cambios en todos los programas* que acceden a ese archivo. Por el contrario, los programas que acceden a un DBMS no necesitan esos cambios en la mayoría de los casos. La estructura de los archivos de datos se almacena en el catálogo DBMS, independientemente de los programas de acceso. Llamaremos a esta propiedad **independencia programa-datos**.

Por ejemplo, un programa de acceso a archivos puede escribirse de modo que sólo pueda acceder a los registros ESTUDIANTE de la estructura mostrada en la Figura 1.4. Si queremos añadir otra porción de datos a cada registro ESTUDIANTE, por ejemplo FechaNac, un programa semejante ya no funcionará y deberá modificarse. Por el contrario, en un entorno DBMS, sólo tendremos que cambiar la descripción de los registros ESTUDIANTE en el catálogo (véase la Figura 1.3) para reflejar la inclusión del nuevo elemento de datos FechaNac; ningún programa cambia. La siguiente vez que un programa DBMS haga referencia al catálogo, se podrá utilizar y acceder a la estructura nueva de los registros ESTUDIANTE.

En algunos tipos de sistemas de bases de datos, como los sistemas orientados a objetos y los de objetos relacionales (consulte los Capítulos 20 a 22), los usuarios pueden definir operaciones sobre los datos como parte de las definiciones de la base de datos. Una **operación** (también denominada *función* o *método*) se puede especificar de dos formas. La *interfaz* (o *firma*) de una operación incluye el nombre de la operación y los tipos de datos de sus argumentos (o parámetros). La *implementación* (o *método*) de la operación se especifica separadamente y puede modificarse sin que la interfaz se vea afectada. Las aplicaciones de usuario pueden operar sobre los datos invocando estas operaciones por sus nombres y argumentos, independientemente de cómo estén implementadas las operaciones. Esto puede recibir el nombre de **independencia programa-operación**.

La característica que permite la independencia programa-datos y la independencia programa-operación se denomina **abstracción de datos**. Un DBMS proporciona a los usuarios una **representación conceptual** de los datos que no incluye muchos de los detalles de cómo están almacenados los datos o de cómo están implementadas las operaciones. Informalmente, un **modelo de datos** es un tipo de abstracción de datos que se utiliza para proporcionar esa representación conceptual. El modelo de datos utiliza conceptos lógicos, como objetos, sus propiedades y sus relaciones, lo que para la mayoría de los usuarios es más fácil de entender que los conceptos de almacenamiento en el computador. Por ello, el modelo de datos *oculta* los detalles del almacenamiento y de la implementación que no resultan interesantes a la mayoría de los usuarios de bases de datos.

A modo de ejemplo, considere las Figuras 1.2 y 1.3. La implementación interna de un archivo puede definirse por la longitud de su registro (el número de caracteres o bytes de cada registro) y cada elemento de datos puede especificarse mediante su byte inicial dentro de un registro y su longitud en bytes. El registro ESTUDIANTE se representaría entonces como se muestra en la Figura 1.4. Pero un usuario típico de una base de datos no se preocupa por la ubicación de cada elemento de datos dentro de un registro, ni por su longitud; más bien, la preocupación del usuario está en que cuando haga una referencia al Nombre de un ESTUDIANTE quiere obtener el valor correcto. En la Figura 1.2 se ofrece una representación conceptual de los registros ESTUDIANTE. El DBMS puede ocultar a los usuarios de la base de datos muchos otros detalles de la organización del almacenamiento de los datos (como las rutas de acceso especificadas en un archivo); los detalles sobre el almacenamiento se explican en los Capítulos 13 y 14.

En la metodología de bases de datos, la estructura detallada y la organización de cada archivo se almacenan en el catálogo. Los usuarios de bases de datos y los programas de aplicación hacen referencia a la representación conceptual de los archivos y, cuando los módulos de acceso al archivo DBMS necesita detalles sobre el almacenamiento del archivo, el DBMS los extrae del catálogo. Se pueden utilizar muchos modelos de datos para proporcionar esta abstracción de datos a los usuarios de bases de datos. Una buena parte de este libro está

**Figura 1.4.** Formato de almacenamiento interno de un registro ESTUDIANTE, basándose en el catálogo de la base de datos de la Figura 1.3.

| Nombre del elemento de datos | Posición inicial en el registro | Longitud en caracteres (bytes) |
|---|---|---|
| Nombre | 1 | 30 |
| NumEstudiante | 31 | 4 |
| Clase | 35 | 1 |
| Especialidad | 36 | 4 |

dedicada a presentar distintos modelos de datos y los conceptos que utilizan para abstraer la representación de los datos.

En las bases de datos orientadas a objetos y de objetos relacionales, el proceso de abstracción no sólo incluye la estructura de datos, sino también las operaciones sobre los datos. Estas operaciones proporcionan una abstracción de las actividades del minimundo normalmente entendidas por los usuarios. Por ejemplo, se puede aplicar una operación CALCULAR_CM a un objeto ESTUDIANTE para calcular la calificación media. Dichas operaciones pueden ser invocadas por las consultas del usuario o por las aplicaciones, sin necesidad de conocer los detalles de cómo están implementadas esas operaciones. En este sentido, una abstracción de la actividad del minimundo queda a disposición de los usuarios como una **operación abstracta**.

### 1.3.3 Soporte de varias vistas de los datos

Normalmente una base de datos tiene muchos usuarios, cada uno de los cuales puede necesitar una perspectiva o vista diferente de la base de datos. Una **vista** puede ser un subconjunto de la base de datos o puede contener **datos virtuales** derivados de los archivos de la base de datos pero que no están explícitamente almacenados. Algunos usuarios no tienen la necesidad de preocuparse por si los datos a los que se refieren están almacenados o son derivados. Un DBMS multiusuario cuyos usuarios tienen variedad de diferentes aplicaciones debe ofrecer facilidades para definir varias vistas. Por ejemplo, un usuario de la base de datos de la Figura 1.2 puede estar interesado únicamente en acceder e imprimir el certificado de estudios de cada estudiante; la Figura 1.5(a) muestra la vista para este usuario. Un segundo usuario, que sólo está interesado en comprobar que los estudiantes cumplen con todos los prerrequisitos de cada curso para poder registrarse, puede requerir la vista representada en la Figura 1.5(b).

**Figura 1.5.** Dos vistas derivadas de la base de datos de la Figura 1.2. (a) Vista del certificado de estudios. (b) Vista de los prerrequisitos del curso.

**CERTIFICADO**

| NombreEstudiante | CertificadoEstudiante | | | | |
|---|---|---|---|---|---|
| | NumCurso | Nota | Semestre | Año | IDSeccion |
| Luis | CC1310 | C | Otoño | 05 | 119 |
| | MAT2410 | B | Otoño | 05 | 112 |
| Carlos | MAT2410 | A | Otoño | 04 | 85 |
| | CC1310 | A | Otoño | 04 | 92 |
| | CC3320 | B | Primavera | 05 | 102 |
| | CC3380 | A | Otoño | 05 | 135 |

**(a)**

**PRERREQUISITO_CURSO**

| NombreCurso | NumCurso | Prerrequisitos |
|---|---|---|
| Bases de datos | CC3380 | CC3320 |
| | | MAT2410 |
| Estructuras de datos | CC3320 | CC1310 |

**(b)**

### 1.3.4 Compartición de datos y procesamiento de transacciones multiusuario

Un DBMS multiusuario, como su nombre indica, debe permitir que varios usuarios puedan acceder a la base de datos al mismo tiempo. Esto es esencial si los datos destinados a varias aplicaciones serán integrados y mantenidos en una sola base de datos. El DBMS debe incluir software de **control de la concurrencia** para que esos varios usuarios que intentan actualizar los mismos datos, lo hagan de un modo controlado para que el resultado de la actualización sea correcto. Por ejemplo, si varios agentes de viajes intentan reservar un asiento en un vuelo, el DBMS debe garantizar que en cada momento sólo un agente tiene acceso a la asignación de ese asiento para un pasajero. Estos tipos de aplicaciones se denominan, por lo general, aplicaciones de **procesamiento de transacciones en línea (OLTP, *online transaction processing*)**. Un papel fundamental del software DBMS multiusuario es garantizar que las transacciones concurrentes operan correcta y eficazmente.

El concepto de transacción es cada vez más importante para las aplicaciones de bases de datos. Una transacción es un *programa en ejecución* o *proceso* que incluye uno o más accesos a la base de datos, como la lectura o la actualización de los registros de la misma. Se supone que una transacción ejecuta un acceso lógicamente correcto a la base de datos si lo ejecutó íntegramente sin interferencia de otras transacciones. El DBMS debe implementar varias propiedades de transacción. La propiedad **aislamiento** garantiza que parezca que cada transacción se ejecuta de forma aislada de otras transacciones, aunque puedan estar ejecutándose cientos de transacciones al mismo tiempo. La propiedad de **atomicidad** garantiza que se ejecuten o todas o ninguna de las operaciones de bases de datos de una transacción. En la Parte 5 se explican las transacciones más en profundidad.

Las características anteriores son muy importantes para distinguir un DBMS del software de procesamiento de archivos tradicional. En la Sección 1.6 explicamos las características adicionales que caracterizan un DBMS. No obstante, en primer lugar clasificaremos los diferentes tipos de personas que trabajan en el entorno de un sistema de bases de datos.

## 1.4 Actores de la escena

En el caso de una base de datos personal pequeña, como la lista de direcciones mencionada en la Sección 1.1, un usuario normalmente define, construye y manipula la base de datos, de modo que no se comparten datos. Sin embargo, en empresas grandes, muchas personas están implicadas en el diseño, uso y mantenimiento de una base de datos grande con cientos de usuarios. En esta sección identificamos las personas cuyos trabajos implican el uso diario de una base de datos grande; las denominaremos *actores de la escena*. En la Sección 1.5 hablaremos de las personas que podríamos llamar *trabajadores entre bambalinas* (los que trabajan en el mantenimiento del entorno del sistema de bases de datos pero que no están activamente interesados en la propia base de datos).

### 1.4.1 Administradores de las bases de datos

En cualquier empresa donde muchas personas utilizan los mismo recursos, se necesita un administrador jefe que supervise y administre esos recursos. En un entorno de bases de datos, el recurso principal es la base de datos en sí misma, mientras que el recurso secundario es el DBMS y el software relacionado. La administración de estos recursos es responsabilidad del **administrador de la base de datos (DBA, *database administrator*)**. El DBA es responsable del acceso autorizado a la base de datos, de la coordinación y monitorización de su uso, y de adquirir los recursos software y hardware necesarios. El DBA también es responsable de problemas como las brechas de seguridad o de unos tiempos de respuesta pobres. En las empresas grandes, el DBA está asistido por un equipo de personas que llevan a cabo estas funciones.

### 1.4.2 Diseñadores de las bases de datos

Los **diseñadores de las bases de datos** son los responsables de identificar los datos que se almacenarán en la base de datos y de elegir las estructuras apropiadas para representar y almacenar esos datos. Estas tareas se acometen principalmente antes de implementar y rellenar la base de datos. Es responsabilidad de los diseñadores comunicarse con todos los presuntos usuarios de la base de datos para conocer sus requisitos, a fin de crear un diseño que satisfaga sus necesidades. En muchos casos, los diseñadores forman parte de la plantilla del DBA y se les pueden asignar otras responsabilidades una vez completado el diseño de la base de datos. Estos diseñadores normalmente interactúan con los grupos de usuarios potenciales y desarrollan **vistas** de la base de datos que satisfacen los requisitos de datos y procesamiento de esos grupos. Cada vista se analiza después y se *integra* con las vistas de los otros grupos de usuarios. El diseño final de la base de datos debe ser capaz de soportar los requisitos de todos los grupos de usuarios.

### 1.4.3 Usuarios finales

Los usuarios finales son las personas cuyos trabajos requieren acceso a la base de datos para realizar consultas, actualizaciones e informes; la base de datos existe principalmente para ser utilizada. Los usuarios finales se pueden clasificar en varias categorías:

- Los **usuarios finales casuales** acceden ocasionalmente a la base de datos, pero pueden necesitar una información diferente en cada momento. Utilizan un sofisticado lenguaje de consulta de bases de datos para especificar sus peticiones y normalmente son administradores de nivel medio o alto u otros usuarios interesados.
- Los **usuarios finales principiantes** o **paramétricos** constituyen una parte considerable de los usuarios finales de las bases de datos. Su labor principal gira entorno a la consulta y actualización constantes de la base de datos, utilizando tipos de consultas y actualizaciones estándar (denominadas **transacciones enlatadas**) que se han programado y probado cuidadosamente. Las tareas que estos usuarios llevan a cabo son variadas:
  - Los cajeros bancarios comprueban los balances de cuentas, así como las retiradas y los depósitos de fondos.
  - Los agentes de viajes que reservan en aerolíneas, hoteles y compañías de alquiler de automóviles comprueban la disponibilidad de una solicitud dada y hacen la reserva.
  - Los empleados de las estaciones receptoras de las compañías navieras introducen las identificaciones de los paquetes mediante códigos de barras y demás información descriptiva a través de botones para actualizar una base de datos central de paquetes recibidos y en tránsito.
- Entre los **usuarios finales sofisticados** se encuentran los ingenieros, los científicos, los analistas comerciales y otros muchos que están completamente familiarizados con el DBMS a fin de implementar sus aplicaciones y satisfacer sus complejos requisitos.
- Los **usuarios finales independientes** mantienen bases de datos personales utilizando paquetes de programas confeccionados que proporcionan unas interfaces fáciles de usar y basadas en menús o gráficos. Un ejemplo es el usuario de un paquete de impuestos que almacena sus datos financieros personales de cara a la declaración de la renta.

Un DBMS típico proporciona muchas formas de acceder a una base de datos. Los usuarios finales principiantes tienen que aprender muy poco sobre los servicios del DBMS; simplemente tienen que familiarizarse con las interfaces de usuario de las transacciones estándar diseñadas e implementadas para su uso. Los usuarios casuales sólo se aprenden unos cuantos servicios que pueden utilizar repetidamente. Los usuarios sofisticados intentan aprender la mayoría de los servicios del DBMS para satisfacer sus complejos requisitos. Los usuarios independientes normalmente llegan a ser expertos en un paquete de software específico.

### 1.4.4 Analistas de sistemas y programadores de aplicaciones (ingenieros de software)

Los analistas de sistemas determinan los requisitos de los usuarios finales, especialmente de los usuarios finales principiantes y paramétricos, así como las especificaciones de desarrollo para las transacciones enlatadas que satisfacen esos requisitos. Los **programadores de aplicaciones** implementan esas especificaciones como programas; después, verifican, depuran, documentan y mantienen esas transacciones enlatadas. Dichos analistas y programadores (normalmente conocidos como **desarrolladores de software** o **ingenieros de software**) deben familiarizarse con todas las posibilidades proporcionadas por el DBMS al objeto de desempeñar sus tareas.

## 1.5 Trabajadores entre bambalinas

Además de los que diseñan, utilizan y administran una base de datos, hay otros usuarios que están asociados con el diseño, el desarrollo y el funcionamiento de un *entorno de software y sistema DBMS*. Estas personas normalmente no están interesadas en la base de datos propiamente dicha. Los denominaremos *trabajadores entre bambalinas* y los dividiremos en las siguientes categorías:

- **Diseñadores e implementadores de sistemas DBMS.** Diseñan e implementan los módulos y las interfaces DBMS como un paquete software. Un DBMS es un sistema software muy complejo compuesto por muchos componentes, o **módulos**, incluyendo los destinados a implementar el catálogo, procesar el lenguaje de consulta, procesar la interfaz, acceder y almacenar los datos en un búfer, controlar la concurrencia, y manipular la recuperación y la seguridad de los datos. El DBMS debe interactuar con otro software de sistema, como el sistema operativo y los compiladores de diversos lenguajes de programación.
- **Desarrolladores de herramientas.** Diseñan e implementan **herramientas** (paquetes de software que facilitan el modelado y el diseño de la base de datos, el diseño del sistema de bases de datos y la mejora del rendimiento). Las herramientas son paquetes opcionales que a menudo se compran por separado. Entre ellas podemos citar los paquetes para el diseño de bases de datos, la monitorización del rendimiento, las interfaces gráficas o en otros idiomas, el prototipado, la simulación y la generación de datos de prueba. En muchos casos, los fabricantes de software independiente desarrollan y comercializan estas herramientas.
- **Operadores y personal de mantenimiento** (personal de administración del sistema). Son los responsables de la ejecución y el mantenimiento real del entorno hardware y software para el sistema de bases de datos.

Aunque estas categorías de trabajadores entre bambalinas se encargan de que el sistema de bases de datos esté disponible para los usuarios finales, normalmente no utilizan la base de datos para sus propios fines.

## 1.6 Ventajas de utilizar una metodología DBMS

En esta sección explicaremos algunas de las ventajas de utilizar un DBMS y las capacidades que un buen DBMS debe poseer. Estas capacidades se añaden a las cuatro características principales explicadas en la Sección 1.3. El DBA debe utilizar estas capacidades para acometer una variedad de objetivos relacionados con el diseño, la administración y el uso de una base de datos multiusuario grande.

### 1.6.1 Control de la redundancia

En el desarrollo tradicional de software que hace uso del procesamiento de archivos, cada grupo de usuarios mantiene sus propios archivos para manipular sus aplicaciones de procesamiento de datos. Por ejemplo,

vamos a retomar la base de datos UNIVERSIDAD de la Sección 1.2; aquí, el personal que registra los cursos y la oficina de contabilidad podrían ser los dos grupos de usuarios. En la metodología tradicional, cada grupo mantiene sus propios archivos de estudiantes. La oficina de contabilidad guarda datos sobre el registro y la información de facturación relacionada, mientras que la oficina de registro hace un seguimiento de los cursos y las calificaciones de los estudiantes. Aparte de estos dos grupos, puede haber otros que dupliquen parte o todos estos mismos datos en sus archivos propios.

La **redundancia** resultante de almacenar los mismos datos varias veces conduce a serios problemas. En primer lugar, las actualizaciones lógicas sencillas (como la introducción de los datos de un estudiante nuevo) hay que hacerlas varias veces: una por cada archivo donde se almacenen los datos de los estudiantes. Esto lleva a una *duplicación del esfuerzo*. En segundo lugar, *se derrocha espacio de almacenamiento* al guardar repetidamente los mismos datos, y este problema puede llegar a ser muy serio en las bases de datos más grandes. En tercer lugar, los archivos que representan los mismos datos pueden acabar siendo incoherentes, lo que puede ocurrir cuando una determinada actualización se aplica a unos archivos y a otros no. Incluso si una actualización (por ejemplo, la adición de un estudiante nuevo) se aplica a todos los archivos adecuados, los datos relacionados con ese estudiante pueden ser *incoherentes* porque las actualizaciones han sido aplicadas por los distintos grupos de usuarios. Por ejemplo, un grupo de usuarios puede introducir erróneamente la fecha de nacimiento del estudiante (‘19-ENE-1988’), mientras que otro grupo la introduce correctamente (‘29-ENE-1988’).

En la metodología de bases de datos, las vistas de los diferentes grupos de usuarios se integran durante el diseño de la base de datos. Idealmente, debemos tener un diseño que almacene cada elemento de datos lógico (como el nombre o la fecha de nacimiento del estudiante) *sólo en un lugar* de la base de datos. Este hecho garantiza la coherencia y ahorra espacio de almacenamiento. Sin embargo, en la práctica, a veces es necesario recurrir a una **redundancia controlada** para mejorar el rendimiento de las consultas. Por ejemplo, podemos almacenar NombreEstudiante y NumCurso de forma redundante en un archivo INFORME_CALIF (véase la Figura 1.6[a]) porque siempre que recuperemos un registro de este último, queremos recuperar el nombre del estudiante y el número del curso, junto con la calificación, el número de estudiante y el identificador de la sección. Al colocar todos los datos juntos, no tenemos que buscar en varios archivos para recopilarlos. En estos casos, el DBMS debe tener la capacidad de *controlar* esta redundancia para evitar las incoherencias entre

**Figura 1.6.** Almacenamiento redundante de NombreEstudiante y NumCurso en INFORME_CALIF. (a) Datos coherentes. (b) Registro incoherente.

**(a) INFORME_CALIF**

| NumEstudiante | NombreEstudiante | IDSeccion | NumCurso | Nota |
|---|---|---|---|---|
| 17 | Luis | 112 | MAT2410 | B |
| 17 | Luis | 119 | CC1310 | C |
| 8 | Carlos | 85 | MAT2410 | A |
| 8 | Carlos | 92 | CC1310 | A |
| 8 | Carlos | 102 | CC3320 | B |
| 8 | Carlos | 135 | CC3380 | A |

**(b) INFORME_CALIF**

| NumEstudiante | NombreEstudiante | IDSeccion | NumCurso | Nota |
|---|---|---|---|---|
| 17 | Carlos | 112 | MAT2410 | B |

archivos. Esto se puede hacer automáticamente comprobando que los valores NombreEstudiante-NumEstudiante de cualquier registro de INFORME_CALIF de la Figura 1.6(a) coincide con alguno de los valores Nombre-NumEstudiante del registro ESTUDIANTE (véase la Figura 1.2). De forma parecida, los valores IDSeccion-NumCurso de INFORME_CALIF pueden compararse con los registros de SECCIÓN. Estas comprobaciones pueden especificarse en el DBMS durante el diseño de la base de datos y que el DBMS las ejecute automáticamente siempre que se actualice el archivo INFORME_CALIF. La Figura 1.6(b) muestra un registro de INFORME_CALIF que es incoherente con el archivo ESTUDIANTE de la Figura 1.2, que puede introducirse incorrectamente de *no controlarse la redundancia*.

### 1.6.2 Restricción del acceso no autorizado

Cuando varios usuarios comparten una base de datos grande, es probable que la mayoría de los mismos no tengan autorización para acceder a toda la información de la base de datos. Por ejemplo, los datos financieros se consideran a menudo confidenciales, y sólo las personas autorizadas pueden acceder a ellos. Además, algunos usuarios sólo pueden recuperar datos, mientras que otros pueden recuperarlos y actualizarlos. Así pues, también hay que controlar el tipo de operación de acceso (recuperación o actualización). Normalmente, los usuarios o grupos de usuarios tienen números de cuenta protegidos mediante contraseñas, que pueden utilizar para tener acceso a la base de datos. Un DBMS debe proporcionar **seguridad y un subsistema de autorización**, que el DBA utiliza para crear cuentas y especificar las restricciones de las mismas. Después, el DBMS debe implementar automáticamente esas restricciones. Podemos aplicar controles parecidos al software DBMS. Por ejemplo, sólo el personal del DBA puede utilizar cierto **software privilegiado**, como el que permite crear cuentas nuevas. De forma parecida, los usuarios paramétricos pueden acceder a la base de datos sólo a través de transacciones enlatadas desarrolladas para su uso.

### 1.6.3 Almacenamiento persistente para los objetos del programa

Las bases de datos se pueden utilizar para proporcionar **almacenamiento persistente** a los objetos de programa y las estructuras de datos. Es una de las principales razones de los **sistemas de bases de datos orientados a objetos**. Normalmente, los lenguajes de programación tienen estructuras de datos complejas, como tipos de registro en Pascal o definiciones de clase en C++ o Java. Los valores de las variables de un programa se descartan una vez que termina ese programa, a menos que el programador los almacene explícitamente en archivos permanentes, lo que a menudo implica convertir esas estructuras complejas en un formato adecuado para el almacenamiento del archivo. Cuando surge la necesidad de leer estos datos una vez más, el programador debe convertir el formato del archivo a la estructura variable del programa. Los sistemas de bases de datos orientados a objetos son compatibles con lenguajes de programación como C++ y Java, y el software DBMS realiza automáticamente las conversiones necesarias. Por tanto, un objeto complejo de C++ se puede almacenar de forma permanente en un DBMS orientado a objetos. Se dice que dicho objeto es **persistente**, porque sobrevive a la terminación de la ejecución del programa y otro programa C++ lo puede recuperar más tarde.

El almacenamiento persistente de objetos de programas y estructuras de datos es una función importante de los sistemas de bases de datos. Los sistemas de bases de datos tradicionales a menudo adolecían de lo que se denominó **problema de incompatibilidad de impedancia**, puesto que las estructuras de datos proporcionadas por el DBMS eran incompatibles con las estructuras de datos del lenguaje de programación. Los sistemas de bases de datos orientados a objetos normalmente ofrecen la **compatibilidad** de la estructura de datos con uno o más lenguajes de programación orientados a objetos.

### 1.6.4 Suministro de estructuras de almacenamiento para un procesamiento eficaz de las consultas

Los sistemas de bases de datos deben proporcionar capacidades para *ejecutar eficazmente consultas y actualizaciones*. Como la base de datos normalmente se almacena en el disco, el DBMS debe proporcionar estruc-

turas de datos especializadas para acelerar la búsqueda en el disco de los registros deseados. Con este fin se utilizan unos archivos auxiliares denominados **índices**, que están basados casi siempre en el árbol de estructuras de datos o en las estructuras de datos dispersas, convenientemente modificados para la búsqueda en disco. A fin de procesar los registros necesarios de la base de datos para una consulta en particular, estos registros deben copiarse del disco a la memoria. Por consiguiente, el DBMS a menudo tiene un módulo de búfer que mantiene partes de la base de datos en los búferes de la memoria principal. En otros casos, el DBMS puede utilizar el sistema operativo para realizar el volcado de los datos del disco en el búfer.

El **módulo de procesamiento y optimización de consultas** del DBMS es el responsable de elegir un plan eficaz de ejecución de consultas para cada consulta basándose en las estructuras de almacenamiento existentes. La elección de qué índices crear y mantener es parte del diseño y refinamiento de la base de datos física, que es una de las responsabilidades del personal del DBA. En los Capítulos 15 y 16 explicaremos en profundidad el procesamiento, la optimización y el refinamiento de las consultas.

### 1.6.5 Copia de seguridad y recuperación

Un DBMS debe ofrecer la posibilidad de recuperarse ante fallos del hardware o del software. El **subsistema de copia de seguridad y recuperación** del DBMS es el responsable de la recuperación. Por ejemplo, si el computador falla en medio de una transacción compleja de actualización, el subsistema de recuperación es responsable de garantizar la restauración de la base de datos al estado anterior a que comenzase la ejecución de la transacción. Como alternativa, el subsistema de recuperación podría asegurarse de retomar la transacción en el punto en que se interrumpió para que todo su efecto se grabe en la base de datos.

### 1.6.6 Suministro de varias interfaces de usuario

Como una base de datos la utilizan muchos tipos de usuarios con distintos niveles de conocimiento técnico, un DBMS debe proporcionar distintas interfaces de usuario, entre las que podemos citar los lenguajes de consulta para los usuarios casuales, las interfaces de lenguaje de programación para los programadores de aplicaciones, formularios y códigos de comando para los usuarios paramétricos, e interfaces por menús y en el idioma nativo para los usuarios independientes. Tanto las interfaces al estilo de los formularios como las basadas en menús se conocen normalmente como **interfaces gráficas de usuario (GUI, *graphical user interfaces*)**. Existen muchos entornos y lenguajes especializados para especificar las GUIs. También son muy comunes las capacidades de proporcionar interfaces GUI web a una base de datos.

### 1.6.7 Representación de relaciones complejas entre los datos

Una base de datos puede incluir numerosas variedades de datos que se interrelacionan entre sí de muchas formas. Considerando el ejemplo de la Figura 1.2, el registro de 'Carlos' del archivo ESTUDIANTE está relacionado con cuatro registros del archivo INFORME_CALIF. Del mismo modo, cada registro de sección está relacionado con un registro de curso y con varios registros de INFORME_CALIF (uno por cada estudiante que haya completado esa sección). Un DBMS debe tener la capacidad de representar las relaciones complejas entre los datos, definir las nuevas relaciones que surgen, y recuperar y actualizar fácil y eficazmente los datos relacionados.

### 1.6.8 Implementación de las restricciones de integridad

La mayoría de las aplicaciones de bases de datos tienen ciertas **restricciones de integridad** que deben mantenerse para los datos. Un DBMS debe proporcionar servicios para definir e implementar esas restricciones. El tipo de restricción de integridad más simple consiste en especificar un tipo de datos por cada elemento de datos. Por ejemplo, en la Figura 1.3 especificamos que el valor del elemento de datos Clase dentro de cada

registro ESTUDIANTE debe ser un dígito entero, y que el valor de Nombre debe ser una cadena de no más de 30 caracteres alfanuméricos. Para restringir el valor de Clase entre 1 y 5 debe haber una restricción adicional que no se muestra en el catálogo actual. Un tipo de restricción más compleja que se da a menudo implica especificar que un registro de un archivo debe estar relacionado con registros de otros archivos. Por ejemplo, en la Figura 1.2 podemos especificar que *cada registro de sección debe estar relacionado con un registro de curso*. Otro tipo de restricción especifica la unicidad en los valores del elemento de datos, como que *cada registro de curso debe tener un único valor para NumCurso*. Estas restricciones se derivan del significado o la **semántica** de los datos y del minimundo que representan. Los diseñadores tienen la responsabilidad de identificar las restricciones de integridad durante el diseño de la base de datos. Algunas restricciones pueden especificarse en el DBMS e implementarse automáticamente. Otras restricciones pueden tener que ser comprobadas por los programas de actualización o en el momento de introducir los datos. En las aplicaciones grandes es costumbre denominar estas restricciones como **reglas de negocio**.

Aun cuando se introduce erróneamente un elemento de datos, éste puede satisfacer las restricciones de integridad especificadas. Por ejemplo, si un estudiante recibe una calificación de 'A' pero se introduce una calificación de 'C' en la base de datos, el DBMS *no puede* descubrir automáticamente este error porque 'C' es un valor correcto para el tipo de datos Nota. Estos errores en la introducción de los datos sólo se pueden descubrir manualmente (cuando el estudiante recibe la calificación y reclama) y corregirse más tarde mediante la actualización de la base de datos. No obstante, una calificación de 'Z' debería rechazarla automáticamente el DBMS, porque no se trata de un valor correcto para el tipo de datos Nota. Cuando expliquemos cada modelo de datos en los siguientes capítulos, introduciremos reglas que pertenecen implícitamente a ese modelo. Por ejemplo, en el modelo Entidad-Relación del Capítulo 3, una relación debe implicar como mínimo a dos entidades. Estas reglas son **reglas inherentes** del modelo de datos y se asumen automáticamente para garantizar la validez del modelo.

### 1.6.9 Inferencia y acciones usando reglas

Algunos sistemas de bases de datos ofrecen la posibilidad de definir *reglas de deducción* para *inferir* información nueva a partir de los hechos guardados en la base de datos. Estos sistemas se denominan **sistemas de bases de datos deductivos**. Por ejemplo, puede haber reglas complejas en la aplicación del minimundo para determinar si un estudiante está a prueba. Éstas se pueden especificar *declarativamente* como **reglas**, de modo que cuando el DBMS las compila y mantiene pueden determinar todos los estudiantes que están en periodo de prueba. En un DBMS tradicional habría que escribir un *código de programa procedimental* explícito para soportar dichas aplicaciones. Pero si cambian las reglas del minimundo, generalmente es mejor cambiar las reglas de deducción declaradas que volver a codificar los programas procedurales. En los sistemas de bases de datos relacionales actuales es posible asociar *triggers* a las tablas. Un ***trigger*** es una forma de regla que se activa con las actualizaciones de la tabla, lo que conlleva la ejecución de algunas operaciones adicionales sobre otras tablas, el envío de mensajes, etcétera. Los procedimientos más implicados en la implementación de reglas se conocen popularmente como **procedimientos almacenados**; se convierten en parte de la definición global de la base de datos y se les invoca correctamente cuando se dan ciertas condiciones. Los **sistemas de bases de datos activos** ofrecen la funcionalidad más potente; estos sistemas proporcionan reglas activas que pueden iniciar automáticamente acciones cuando ocurren ciertos eventos y condiciones.

### 1.6.10 Implicaciones adicionales de utilizar la metodología de bases de datos

Esta sección explica algunas implicaciones adicionales de usar la metodología de bases de datos que pueden beneficiar a la mayoría de las empresas.

**Potencial para implementar estándares.** La metodología de bases de datos permite al DBA definir e implementar estándares entre los usuarios de la base de datos en una empresa grande. Esto facilita la comu-

nicación y la cooperación entre varios departamentos, proyectos y usuarios dentro de la empresa. Los estándares se pueden definir para los nombres y los formatos de los elementos de datos, los formatos de visualización, las estructuras de los informes, la terminología, etcétera. El DBA puede implementar los estándares en un entorno de base de datos centralizado más fácilmente que en un entorno donde cada grupo de usuarios tiene el control de sus propios archivos y software.

**Tiempo de desarrollo de aplicación reducido.** Uno de los principales reclamos de venta de la metodología de bases de datos es que se necesita muy poco tiempo para desarrollar una aplicación nueva (como la recuperación de ciertos datos de la base de datos para imprimir un informe nuevo). El diseño y la implementación de una base de datos nueva desde el principio puede llevar más tiempo que escribir una aplicación de archivos especializada. No obstante, una vez que la base de datos está operativa y en ejecución, por lo general se necesita mucho menos tiempo para crear aplicaciones nuevas utilizando los servicios del DBMS. Se estima que el tiempo de desarrollo utilizando un DBMS es de una sexta a una cuarta parte del necesario para un sistema de archivos tradicional.

**Flexibilidad.** Puede ser necesario cambiar la estructura de una base de datos a medida que cambian los requisitos. Por ejemplo, puede surgir un nuevo grupo de usuarios que necesita información que actualmente no hay en la base de datos. En respuesta, puede que sea necesario añadir un archivo a la base de datos o extender los elementos de datos de un archivo existente. Los DBMS modernos permiten ciertos tipos de cambios evolutivos en la estructura de la base de datos sin que ello afecte a los datos almacenados y a los programas de aplicación existentes.

**Disponibilidad de la información actualizada.** Un DBMS hace que la base de datos esté disponible para todos los usuarios. Tan pronto como se aplica la actualización de un usuario a la base de datos, todos los demás usuarios pueden ver esa actualización inmediatamente. Esta disponibilidad de información actualizada es esencial para muchas de las aplicaciones de procesamiento de transacciones, como las bases de datos de los sistemas de reservas o bancarios, y esto es posible a los subsistemas de control de la concurrencia y de recuperación de un DBMS.

**Economías de escala.** La metodología DBMS permite la consolidación de los datos y las aplicaciones, lo que reduce el derroche de superposición entre las actividades del personal de procesamiento de datos en diferentes proyectos o departamentos, así como las redundancias entre las aplicaciones. Esto permite que toda la organización invierta en procesadores más potentes, dispositivos de almacenamiento o aparatos de comunicación, en lugar de que cada departamento compre sus propios equipos (menos potentes). De este modo se reducen los costes globales de funcionamiento y administración.

## 1.7 Breve historia de las aplicaciones de bases de datos

Esta sección ofrece una breve historia de las aplicaciones que utilizan DBMSs y cómo estas aplicaciones supusieron el impulso de nuevos tipos de sistemas de bases de datos.

### 1.7.1 Las primeras aplicaciones de bases de datos que utilizaron sistemas jerárquicos y de red

Muchas de las primeras aplicaciones de bases de datos almacenaban registros en grandes organizaciones, como corporaciones, universidades, hospitales y bancos. En muchas de esas aplicaciones había muchos registros de estructura parecida. Por ejemplo, en una aplicación para universidades, era preciso mantener información parecida por cada estudiante, cada curso y cada especialidad, etcétera. También había muchos tipos de registros y muchas interrelaciones entre ellos.

Uno de los principales problemas con los primeros sistemas de bases de datos era la mezcla de relaciones conceptuales con el almacenamiento físico y la ubicación de los registros en el disco. Por ejemplo, los registros de especialidad de un estudiante en particular podían guardarse físicamente a continuación del registro del estudiante. Aunque esto ofrecía un acceso muy eficaz para las consultas y las transacciones originales para las que fue diseñada la base de datos, no proporcionaba suficiente flexibilidad para acceder eficazmente a los registros cuando se identificaban consultas y transacciones nuevas. En particular, era muy difícil implementar con eficacia las consultas nuevas que requerían una organización diferente del almacenamiento para un procesamiento eficaz. También era muy laborioso reorganizar la base de datos cuando había cambios en los requisitos de la aplicación.

Otro defecto de los primeros sistemas era que sólo proporcionaban interfaces de lenguaje de programación. La implementación de consultas y transacciones nuevas llevaba mucho tiempo y era costosa, pues había que escribir, probar y depurar programas nuevos. La mayoría de esos sistemas de bases de datos se implantaron en grandes y costosos computadores *mainframe* a mediados de la década de 1960, y a lo largo de las décadas de 1970 y 1980. Los principales tipos de esos sistemas estaban basados en tres paradigmas principales: sistemas jerárquicos, sistemas basados en un modelo de red y sistemas de archivos inversos.

### 1.7.2 Flexibilidad de aplicación con las bases de datos relacionales

Las bases de datos relacionales se propusieron originalmente para separar el almacenamiento físico de los datos de su representación conceptual, así como para proporcionar una base matemática para el almacenamiento de contenidos. El modelo de datos relacional también introdujo lenguajes de consulta de alto nivel que proporcionaban una alternativa a las interfaces de lenguaje de programación; por tanto, era mucho más rápido escribir consultas nuevas. La representación relacional de los datos se parece al ejemplo presentado en la Figura 1.2. Los sistemas relacionales estaban destinados inicialmente a las mismas aplicaciones que los primitivos sistemas, pero estaban pensados para ofrecer flexibilidad en el desarrollo de nuevas consultas y para reorganizar la base de datos cuando cambiaran los requisitos.

Los sistemas relacionales experimentales desarrollados a finales de la década de 1970 y los sistemas de administración de bases de datos relacionales (RDBMS) comerciales que aparecieron a principios de la década de 1980 eran muy lentos, pues no utilizaban punteros de almacenamiento físico o la ubicación del registro para acceder a los registros de datos relacionados. Su rendimiento mejoró con el desarrollo de nuevas técnicas de almacenamiento e indexación y unas técnicas mejores de procesamiento y optimización. Eventualmente, las bases de datos relacionales se convirtieron en el tipo de sistema de bases de datos predominante para las aplicaciones de bases de datos tradicionales. En casi todos los tipos de computadores, desde los pequeños computadores personales hasta los grandes servidores, existen bases de datos relacionales.

### 1.7.3 Aplicaciones orientadas a objetos y la necesidad de bases de datos más complejas

El surgimiento de los lenguajes de programación orientados a objetos en la década de 1980 y la necesidad de almacenar y compartir objetos estructurados complejos induce al desarrollo de las bases de datos orientadas a objetos (OODB). Inicialmente, las OODB estaban consideradas como competidoras de las bases de datos relacionales, porque proporcionaban más estructuras de datos generales. También incorporaban muchos de los útiles paradigmas de la orientación a objetos, como los tipos de datos abstractos, la encapsulación de operaciones, la herencia y la identidad de objeto. No obstante, la complejidad del modelo y la carencia de un estándar contribuyó a su limitado uso. Ahora se utilizan principalmente en las aplicaciones especializadas (por ejemplo, en ingeniería, publicación multimedia y sistemas de fabricación). A pesar de las expectativas de que iban a provocar un gran impacto, lo cierto es que su penetración global en el mercado de productos de bases de datos permanece aún hoy por debajo del 50%.

### 1.7.4 Intercambio de datos en la Web para el comercio electrónico

La World Wide Web proporciona una gran red de computadores interconectados. Los usuarios pueden crear documentos utilizando un lenguaje de publicación web, como HTML (Lenguaje de marcado de hipertexto, *HyperText Markup Language*), y almacenar esos documentos en servidores web desde los que otros usuarios (clientes) pueden acceder a ellos. Los documentos se pueden enlazar mediante **hipervínculos**, que son punteros a otros documentos. En la década de 1990 apareció el comercio electrónico (*e-commerce*) como una aplicación trascendental en la Web. Cada vez iba siendo más evidente que parte de la información que aparecía en las páginas web de *e-commerce* a menudo eran datos que se extraían dinámicamente de unos DBMSs. Se desarrollaron varias técnicas que permitían el intercambio de datos en la Web. Actualmente, XML (Lenguaje de marcado extendido, *eXtended Markup Language*) está considerado como el principal estándar para el intercambio de datos entre varios tipos de bases de datos y páginas web. XML combina conceptos de los modelos utilizados en los sistemas de documentación con conceptos de modelado de bases de datos. El Capítulo 27 está dedicado a la explicación de XML.

### 1.7.5 Capacidades extendidas de las bases de datos para las nuevas aplicaciones

El éxito de los sistemas de bases de datos en las aplicaciones tradicionales animó a los desarrolladores de otros tipos de aplicaciones a intentar utilizarlos. Dichas aplicaciones, de las que se ofrecen unos ejemplos a continuación, utilizaban tradicionalmente sus propias estructuras de archivos y datos especializadas:

- **Aplicaciones científicas.** Almacenan grandes cantidades de datos resultado de los experimentos científicos en áreas como la física o el mapa del genoma humano.
- **Almacenamiento y recuperación de imágenes,** desde noticias escaneadas y fotografías personales, hasta imágenes de satélite o las procedentes de procedimientos médicos, como los rayos X o el MRI (procesamiento de imágenes de resonancia magnética).
- **Almacenamiento y recuperación de vídeos,** como películas, o **videoclips**, procedentes de noticias o cámaras digitales personales.
- **Aplicaciones de minado de datos,** que analizan grandes cantidades de datos buscando ocurrencias de patrones específicos o relaciones.
- **Aplicaciones espaciales,** que almacenan las ubicaciones espaciales de datos como la información meteorológica o los mapas que se utilizan en los sistemas de información geográfica.
- **Aplicaciones de series cronológicas** que almacenan información como datos económicos a intervalos regulares de tiempo (por ejemplo, gráficos de las ventas diarias o del producto nacional bruto mensual).

Es evidente que los sistemas relacionales básicos no eran muy adecuados para muchas de estas aplicaciones, normalmente por una o más de las siguientes razones:

- Se necesitaban estructuras de datos más complejas para modelar la aplicación que la simple representación relacional.
- Se necesitaron nuevos tipos de datos, **además** de los tipos numérico y de cadena de caracteres básicos.
- Para manipular los nuevos tipos de datos eran necesarias operaciones y construcciones de lenguaje de consulta nuevas.
- Se necesitaban nuevas estructuras de almacenamiento e indexación.

Esto llevó a que los desarrolladores de DBMS añadieran funcionalidad a sus sistemas. Parte de esa funcionalidad era de propósito general, como la incorporación de conceptos de las bases de datos orientadas a objetos

en los sistemas relacionales. Otra parte de esa funcionalidad era de propósito especial, en forma de módulos opcionales que se podían utilizar para aplicaciones específicas. Por ejemplo, los usuarios podrían comprar un módulo de series cronológicas para utilizarlo con su DBMS relacional para su aplicación de series cronológicas.

Actualmente, la mayoría de las organizaciones grandes utilizan distintos paquetes que funcionan en estrecha colaboración con **bases de datos *back-ends*.** Una base de datos *back-end* representa una o más bases de datos, seguramente de distintos fabricantes y diferentes modelos de datos, encaminado todo ello a almacenar los datos que esos paquetes manipulan para las transacciones, la generación de informes y dar respuesta a las consultas específicas. Uno de los sistemas que más se utiliza es **ERP (Planificación de recursos empresariales, *Enterprise Resource Planning*)**, que se utiliza para consolidar diferentes áreas funcionales dentro de una organización, como, por ejemplo, la producción, las ventas, la distribución, el marketing, las finanzas, los recursos humanos, etcétera. Otro tipo muy conocido de sistema es el software **CRM (Administración de las relaciones con el cliente, *Customer Relationship Management*)**, que abarca el procesamiento de pedidos y las funciones de marketing y soporte de clientes. Estas aplicaciones son compatibles con la Web para aquellos usuarios internos y externos a los que se dota de diferentes interfaces de portal web para interactuar con la base de datos *back-end*.

### 1.7.6 Bases de datos frente a recuperación de información

Tradicionalmente, la tecnología de bases de datos se aplica a los datos estructurados y formateados que se originan en las aplicaciones rutinarias gubernamentales, comerciales e industriales. Esta tecnología se utiliza mucho en la fabricación, las ventas, la banca, los seguros, las finanzas y la salud, donde los datos estructurados originan formularios como las facturas o los documentos de registro de pacientes. Ha habido un desarrollo concurrente de un campo denominado recuperación de información (IR, *information retrieval*) que tiene que ver con los libros, los manuscritos y distintos formularios de artículos basados en bibliotecas. Los datos se indexan, catalogan y anotan utilizando palabras clave. IR tiene que ver con la búsqueda de material basada en esas palabras clave, y con muchos de los problemas relacionados con el procesamiento de documentos y el procesamiento de texto de forma libre. Se ha realizado una cantidad considerable de trabajo en buscar texto basándose en palabras clave, buscar documentos y clasificarlos por su relevancia, clasificar el texto automáticamente, clasificar el texto por temas, etcétera. Con la llegada de la Web y la proliferación de las páginas HTML ejecutándose por miles de millones, es necesario aplicar muchas de las técnicas de IR para procesar los datos en la Web. Los datos de las páginas web son normalmente imágenes, texto y objetos que se activan y modifican dinámicamente. La recuperación de información en la Web es un problema nuevo que requiere la aplicación de técnicas de bases de datos e IR en variedad de nuevas combinaciones.

## 1.8 Cuándo no usar un DBMS

A pesar de las ventajas de usar un DBMS, hay algunas situaciones en las que su uso puede suponer unos sobrecostes innecesarios en los que no se incurriría con el procesamiento tradicional de archivos. Los sobrecostes de utilizar un DBMS se deben a lo siguiente:

- Inversión inicial muy alta en hardware, software y formación.
- La generalidad de que un DBMS ofrece definición y procesamiento de datos.
- Costes derivados de las funciones de seguridad, control de la concurrencia, recuperación e integridad.

Es posible que surjan otros problemas si los diseñadores y el DBA no diseñan correctamente la base de datos o si las aplicaciones de sistemas de bases de datos no se implantan correctamente. Por tanto, puede ser más deseable utilizar archivos normales en las siguientes circunstancias:

- Aplicaciones de bases de datos sencillas y bien definidas que no es previsible que cambien.

- Requisitos estrictos y en tiempo real para algunos programas que no podrían satisfacerse debido al sobrecoste de un DBMS.
- Inexistencia del acceso multiusuario a los datos.

Algunas industrias y aplicaciones prefieren no utilizar DBMSs de propósito general. Por ejemplo, muchas de las herramientas de diseño asistido por computador (CAD) que los ingenieros mecánicos y civiles utilizan, tienen archivos propietarios y software de administración de datos destinados a las manipulaciones internas de dibujos y objetos 3D. De forma parecida, los sistemas de comunicación y conmutación diseñados por empresas como AT&T eran manifestaciones precoces de software de bases de datos que se desarrolló para ejecutarse muy rápidamente con datos organizados jerárquicamente, al objeto de obtener un acceso rápido y el enrutamiento de llamadas. Asimismo, las implementaciones GIS a menudo implantaban sus propios esquemas de organización de datos para implementar eficazmente funciones relacionadas con el procesamiento de mapas, los contornos físicos, las líneas, los polígonos, etcétera. Los DBMSs de propósito general no son adecuados para su propósito.

# 1.9 Resumen

En este capítulo hemos definido una base de datos como una colección de datos relacionados, donde los *datos* son hechos grabados. Una base de datos típica representa algún aspecto del mundo real y es utilizada por uno o más grupos de usuarios con fines específicos. Un DBMS es un paquete de software generalizado destinado a implementar y mantener una base de datos computerizada. La base de datos y el software juntos forman un sistema de bases de datos. Hemos identificado algunas características que distinguen la metodología de bases de datos de las aplicaciones tradicionales de procesamiento de archivos, y hemos explicado las principales categorías de usuarios de las bases de datos, o *actores de la escena*. Además de los usuarios de las bases de datos, el personal de soporte, o *trabajadores entre bambalinas*, se pueden clasificar en varias categorías.

El capítulo también ofrece una lista de las capacidades que un software de DBMS debe ofrecer al DBA, los diseñadores y los usuarios para que les ayude en el diseño, la administración y el uso de una base de datos. Después, se ha ofrecido una perspectiva histórica de la evolución de las aplicaciones de bases de datos. Hemos apuntado al matrimonio de la tecnología de bases de datos con la tecnología de recuperación de información, que jugará un papel muy importante debido a la popularidad de la Web. Por último, hemos hablado de los sobrecostes de utilizar un DBMS y de algunas situaciones en las que no es ventajoso utilizar uno.

## Preguntas de repaso

**1.1.** Defina los siguientes términos: datos, base de datos, DBMS, sistema de bases de datos, catálogo de la base de datos, independencia programa-datos, vista de usuario, DBA, usuario final, transacción enlatada, sistema de bases de datos deductivo, objeto persistente, metadatos y aplicación de procesamiento de transacciones.

**1.2.** ¿Qué cuatro tipos de acciones implican bases de datos? Explique brevemente cada uno de ellos.

**1.3.** Explique las principales características de la metodología de bases de datos y cómo difiere de los sistemas de archivos tradicionales.

**1.4.** ¿Cuáles son las responsabilidades del DBA y de los diseñadores de bases de datos?

**1.5.** ¿Cuáles son los diferentes tipos de bases de datos y usuarios? Explique las actividades principales de cada uno.

**1.6.** Explique las capacidades que un DBMS debe proporcionar.

**1.7.** Explique las diferencias entre los sistemas de bases de datos y los sistemas de recuperación de información.

## Ejercicios

**1.8.** Identifique algunas operaciones de actualización y consultas informales que esperaría aplicar a la base de datos de la Figura 1.2.

**1.9.** ¿Cuál es la diferencia entre la redundancia controlada y la descontrolada? Ilustre su explicación con ejemplos.

**1.10.** Denomine todas las relaciones entre los registros de la base de datos de la Figura 1.2.

**1.11.** Ofrezca algunas vistas adicionales que otros grupos de usuarios podrían necesitar para la base de datos de la Figura 1.2.

**1.12.** Cite algunos ejemplos de restricciones de integridad que piense que podrían darse en la base de datos de la Figura 1.2.

**1.13.** Ofrezca ejemplos de sistemas en los que tenga sentido utilizar el procesamiento tradicional de archivos en lugar de una base de datos.

**1.14.** Considerando la Figura 1.2:

**a.** Si el nombre del departamento 'CC' (Ciencias de la Computación) cambia a 'CCIS' (Ciencias de la computación e Ingeniería de Software), y también cambia el prefijo correspondiente para el curso, identifique las columnas de la base de datos que deben actualizarse.

**b.** ¿Es posible reestructurar las columnas de las tablas CURSO, SECCIÓN y PRERREQUISITO para que sólo sea necesario modificar una columna?

## Bibliografía seleccionada

El ejemplar de octubre de 1991 de *Communications of the ACM and Kim* (1995) incluye varios artículos que describen los DBMSs de la siguiente generación; muchas de las características de las bases de datos explicadas en el pasado están ahora disponibles comercialmente. El ejemplar de marzo de 1976 de *ACM Computing Surveys* ofrece una introducción a los sistemas de bases de datos; al lector interesado le puede proporcionar una perspectiva histórica.

CAPÍTULO 2

# Conceptos y arquitectura de los sistemas de bases de datos

La arquitectura de los paquetes DBMS ha evolucionado desde los antiguos sistemas monolíticos, en los que todo el paquete de software DBMS era un sistema integrado, hasta los modernos paquetes DBMS con un diseño modular y una arquitectura de sistema cliente/servidor. Esta evolución es reflejo de las tendencias en computación, donde los grandes computadores *mainframe* centralizados se han sustituido por cientos de estaciones de trabajo distribuidas y computadores personales conectados a través de redes de comunicaciones a distintos tipos de servidores (servidores web, servidores de bases de datos, servidores de archivos, servidores de aplicaciones, etc.).

En una arquitectura DBMS cliente/servidor básica, la funcionalidad del sistema se distribuye entre dos tipos de módulos.[1] Un **módulo cliente** se diseña normalmente para que se pueda ejecutar en la estación de trabajo de un usuario o en un computador personal. Normalmente, las aplicaciones y las interfaces de usuario que acceden a las bases de datos se ejecutan en el módulo cliente. Por tanto, el módulo cliente manipula la interacción del usuario y proporciona interfaces amigables para el usuario, como formularios o GUIs basadas en menús. El otro tipo de módulo, denominado **módulo servidor**, manipula normalmente el almacenamiento de los datos, el acceso, la búsqueda y otras funciones. En la Sección 2.5 explicaremos más en detalle las arquitecturas cliente/servidor. En primer lugar, estudiaremos más conceptos básicos, que le permitirán tener un mayor conocimiento de las modernas arquitecturas de bases de datos.

En este capítulo veremos la terminología y los conceptos básicos que utilizaremos en todo el libro. La Sección 2.1 explica los modelos de datos y define los conceptos de esquema e instancia, que son fundamentales para el estudio de los sistemas de bases de datos. Después, explicaremos la arquitectura DBMS de tres esquemas y la independencia de los datos en la Sección 2.2; esto proporciona la perspectiva que tiene un usuario de lo que se supone que un DBMS debe hacer. En la Sección 2.3 se describen los tipos de interfaces y lenguajes que un DBMS normalmente proporciona. La Sección 2.4 ofrece un estudio del entorno software de un sistema de bases de datos. La Sección 2.5 ofrece una panorámica de distintos tipos de arquitecturas cliente/servidor. Por último, la Sección 2.6 ofrece una clasificación de los tipos de paquetes DBMS. La Sección 2.7 resume el capítulo.

El material de las Secciones 2.4 a 2.6 ofrece conceptos más detallados que pueden considerarse como complementarios del material de introducción básico.

---

[1] Como veremos en la Sección 2.5 existen variaciones de esta arquitectura cliente/servidor de dos capas sencillas.

# 2.1 Modelos de datos, esquemas e instancias

Una característica fundamental de la metodología de bases de datos es que ofrece algún nivel de abstracción de los datos. La **abstracción de datos** se refiere generalmente a la supresión de detalles de la organización y el almacenamiento de datos y a la relevancia de las características fundamentales para un conocimiento mejorado de los datos. Una de las características principales de la metodología de bases de datos es soportar la abstracción de datos para que diferentes usuarios puedan percibir esos datos con el nivel de detalle que prefieren. Un **modelo de datos** (colección de conceptos que se pueden utilizar para describir la estructura de una base de datos) proporciona los medios necesarios para conseguir esa abstracción.[2] Por *estructura de una base de datos* nos referimos a los tipos de datos, relaciones y restricciones que deben mantenerse para los datos. La mayoría de modelos de datos también incluyen un conjunto de **operaciones básicas** para especificar las recuperaciones y actualizaciones en la base de datos.

Además de las operaciones básicas proporcionadas por el modelo de datos, es cada vez más común incluir conceptos en el modelo de datos para especificar el **aspecto dinámico** o **comportamiento** de una aplicación de base de datos. Esto permite al diseñador de la base de datos especificar un conjunto de operaciones válidas definidas por el usuario que son permitidas en los objetos de la base de datos.[3] Un ejemplo de operación definida por usuario puede ser COMPUTE_GPA, que se puede aplicar al objeto ESTUDIANTE. Por el contrario, las operaciones genéricas para insertar, borrar, modificar o recuperar cualquier clase de objeto se incluyen a menudo en las *operaciones básicas del modelo de datos*. Los conceptos para especificar el comportamiento son fundamentales para los modelos de datos orientados a objetos (consulte los Capítulos 20 y 21), pero también se están incorporando en los modelos de datos más tradicionales. Por ejemplo, los modelos de objetos relacionales (consulte el Capítulo 22) extienden el modelo relacional básico para incluir dichos conceptos, además de otros. En el modelo de datos relacional hay una cláusula para adjuntar el comportamiento a las relaciones en forma de módulos almacenados persistentes, popularmente conocidos como procedimientos almacenados (consulte el Capítulo 9).

## 2.1.1 Categorías de modelos de datos

Se han propuesto muchos modelos de datos, que podemos clasificar conforme a los tipos de conceptos que utilizan para describir la estructura de la base de datos. Los **modelos de datos de alto nivel** o **conceptuales** ofrecen conceptos muy cercanos a como muchos usuarios perciben los datos, mientras que los **modelos de datos de bajo nivel** o **físicos** ofrecen conceptos que describen los detalles de cómo se almacenan los datos en el computador. Los conceptos ofrecidos por los modelos de datos de bajo nivel están pensados principalmente para los especialistas en computadores, no para los usuarios finales normales. Entre estos dos extremos hay una clase de **modelos de datos representativos** (o de **implementación**),[4] que ofrecen conceptos que los usuarios finales pueden entender pero que no están demasiado alejados de cómo se organizan los datos dentro del computador. Los modelos de datos representativos ocultan algunos detalles relativos al almacenamiento de los datos, pero pueden implementarse directamente en un computador.

Los modelos de datos conceptuales utilizan conceptos como entidades, atributos y relaciones. Una entidad representa un objeto o concepto del mundo real, como un empleado o un proyecto que se describe en la base de datos. Un atributo representa alguna propiedad de interés que describe a una entidad, como, por ejemplo,

---

[2] A veces, la palabra *modelo* se utiliza para denotar una descripción de base de datos específica, o esquema (por ejemplo, *el modelo de datos de marketing*). No utilizaremos esta interpretación.

[3] La inclusión de conceptos para describir el comportamiento refleja una tendencia según la cual las actividades de diseño de bases de datos y software se combinan cada vez más en una sola actividad. Tradicionalmente, la declaración de un comportamiento se asocia con el diseño de software.

[4] El término *modelo de datos de implementación* no es un término estándar; lo hemos introducido para hacer referencia a los modelos de datos disponibles en los sistemas de bases de datos comerciales.

el nombre o el salario de un empleado. Una relación entre dos o más entidades representa una asociación entre dos o más entidades; por ejemplo, una relación de trabajo entre un empleado y un proyecto. El Capítulo 3 presenta el modelo Entidad-Relación, un conocido modelo de datos conceptual de alto nivel. El Capítulo 4 describe las abstracciones adicionales que se utilizan para el modelado avanzado, como la generalización, la especialización y las categorías.

Los modelos de datos representativos o de implementación son los más utilizados en los DBMS comerciales tradicionales. Incluyen los modelos de datos relacionales ampliamente utilizados, así como los modelos de datos heredados (los modelos de red y **jerárquicos**) que tanto se han utilizado en el pasado. La Parte 2 está dedicada al modelo de datos relacional, sus operaciones y lenguajes, y algunas técnicas de programación de aplicaciones de bases de datos relacionales.[5] En los Capítulos 8 y 9 se describe el estándar SQL para las bases de datos relacionales. Los modelos de datos representativos representan los datos mediante estructuras de registro y, por tanto, se los conoce a veces como **modelos de datos basados en registros**.

Podemos considerar que el **grupo de modelos de datos de objetos (ODMG, *object data model group*)** es una nueva familia de modelos de datos de implementación de alto nivel que está más cercana a los modelos de datos conceptuales. En los Capítulos 20 y 21 se describen las características generales de las bases de datos de objetos y del estándar ODMG propuesto. Los modelos de datos de objetos también se utilizan a menudo como modelos conceptuales de alto nivel, generalmente en el ámbito de la ingeniería de software.

Los modelos de datos físicos describen cómo se almacenan los datos en el computador en forma de archivos, representando la información como formatos de registro, ordenación de registros y rutas de acceso. Una ruta de acceso es una estructura que hace más eficaz la búsqueda de registros en una base de datos. En los Capítulos 13 y 14 explicaremos las técnicas de almacenamiento físico y las estructuras de acceso. Un **índice** es un ejemplo de ruta de acceso que permite el acceso directo a los datos que utilizan un término del índice o una palabra clave. Es parecido al índice final de este libro, excepto que se puede organizar lineal o jerárquicamente, o de algún otro modo.

### 2.1.2 Esquemas, instancias y estado de la base de datos

En cualquier modelo de datos es importante distinguir entre la *descripción* de la base de datos y la *misma base de datos*. La descripción de una base de datos se denomina **esquema de la base de datos**, que se especifica durante la fase de diseño y no se espera que cambie con frecuencia.[6] La mayoría de los modelos de datos tienen ciertas convenciones para la visualización de los esquemas a modo de diagramas. Un esquema visualizado se denomina **diagrama del esquema**. La Figura 2.1 muestra un diagrama del esquema para la base de datos de la Figura 1.2; el diagrama muestra la estructura de cada tipo de registro, pero no las instancias reales de los registros. A cada objeto del esquema (como ESTUDIANTE o CURSO) lo denominamos **estructura de esquema**.

Un diagrama del esquema sólo muestra *algunos aspectos* de un esquema, como los nombres de los tipos de registros y los elementos de datos, y algunos tipos de restricciones. Otros aspectos no se especifican; por ejemplo, la Figura 2.1 no muestra los tipos de datos de cada elemento de datos, ni las relaciones entre los distintos archivos. En los diagramas de esquemas no se representan muchos de los tipos de restricciones. Una restricción como, por ejemplo, "*los estudiantes que se especializan en ciencias de la computación deben terminar CC1310 antes de finalizar su curso de segundo año*", es muy difícil de representar.

Los datos reales de una base de datos pueden cambiar con mucha frecuencia. Por ejemplo, la base de datos de la Figura 1.2 cambia cada vez que se añade un estudiante o se introduce una calificación nueva. Los

[5] En los Apéndices E y F se incluye un resumen de los modelos de datos de red y jerárquicos. Son accesibles desde el sitio web *www.librosite.net/elmasri.*

[6] Normalmente, es necesario hacer cambios en el esquema cuando cambian los requisitos de las aplicaciones de bases de datos. Los sistemas de bases de datos más nuevos incluyen operaciones para permitir cambios en el esquema, aunque el proceso de cambio del esquema es más complejo que las sencillas actualizaciones de la base de datos.

**Figura 2.1.** Diagrama del esquema para la base de datos de la Figura 1.2.

datos de la base de datos en un momento concreto se denominan **estado de la base de datos** o ***snapshot* (captura)**. También reciben el nombre de conjunto *actual* de **ocurrencias o instancias** de la base de datos. En un estado dado de la base de datos, cada estructura de esquema tiene su propio *conjunto actual* de instancias; por ejemplo, la construcción ESTUDIANTE contendrá el conjunto de entidades estudiante individuales (registros) como sus instancias. Es posible construir muchos estados de la base de datos para que se correspondan con un esquema de bases de datos particular. Cada vez que se inserta o borra un registro, o cambia el valor de un elemento de datos de un registro, cambia el estado de la base de datos por otro.

Esta distinción entre esquema de la base de datos y estado de la base de datos es muy importante. Cuando **definimos** una base de datos nueva, sólo especificamos su esquema al DBMS. A estas alturas, el estado correspondiente de la base de datos es el *estado vacío*, sin datos. El *estado inicial* de la base de datos se da cuando ésta se **rellena o carga** por primera vez con los datos iniciales. Desde ese momento, cada vez que sobre la base de datos se aplica una operación de actualización, obtenemos otro estado de la base de datos. En cualquier momento en el tiempo, la base de datos tiene un *estado actual*.[7] El DBMS es en parte responsable de garantizar que cada estado de la base de datos sea un **estado válido**; es decir, un estado que satisfaga la estructura y las restricciones especificadas en el esquema. Por tanto, especificar un esquema correcto al DBMS es sumamente importante, por lo que el esquema debe diseñarse con sumo cuidado. El DBMS almacena las descripciones de las construcciones de esquema y las restricciones (también denominadas **metadatos**) en el catálogo del DBMS, para que el software DBMS pueda dirigirse al esquema siempre que lo necesite. En ocasiones, el esquema recibe el nombre de **intención**, y el estado de la base de datos **extensión** del esquema.

Aunque, como ya mencionamos, no se supone que el esquema cambie con frecuencia, no es raro que ocasionalmente haya que introducir algún cambio en él al cambiar los requisitos de la aplicación. Por ejemplo, podemos decidir que es necesario almacenar otro elemento de datos por cada registro del archivo, como añadir Fecha_nac al esquema ESTUDIANTE de la Figura 2.1. Esto se conoce como evolución del esquema. Los DBMS más modernos incluyen algunas operaciones para la **evolución del esquema** que se pueden aplicar mientras la base de datos es operativa.

---

[7] El estado actual también se denomina *snapshot actual* de la base de datos.

# 2.2 Arquitectura de tres esquemas e independencia de los datos

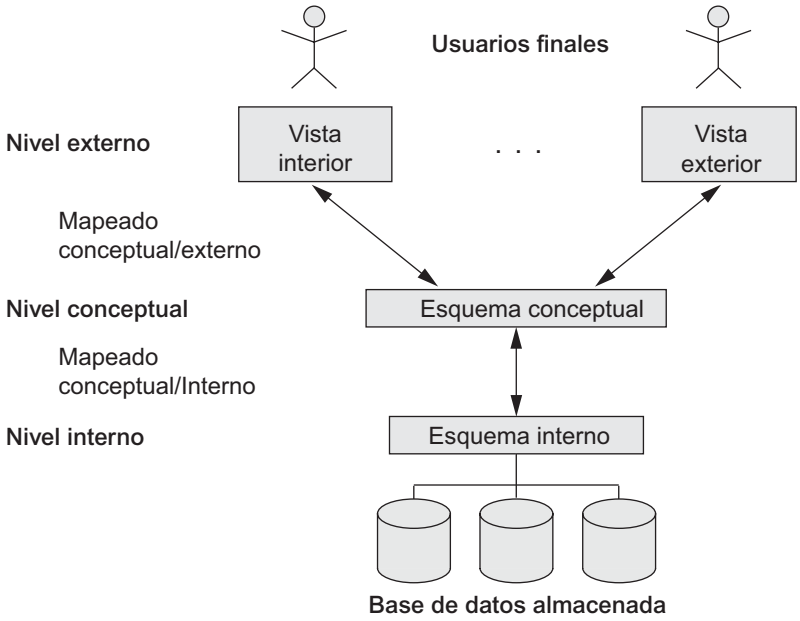
Tres de las cuatro importantes características de la metodología de bases de datos que se mencionaron en la Sección 1.3 son: (1) aislamiento de los programas y los datos (independencia programa-datos y programa-operación), (2) soporte de varias vistas de usuario y (3) uso de un catálogo para almacenar la descripción de la base de datos (esquema). En esta sección especificamos una arquitectura para los sistemas de bases de datos, denominada **arquitectura de tres esquemas**,[8] que se propuso para ayudar a conseguir y visualizar estas características. Después explicaremos el concepto de independencia de los datos.

## 2.2.1 Arquitectura de tres esquemas

El objetivo de la arquitectura de tres esquemas, ilustrada en la Figura 2.2, es separar las aplicaciones de usuario y las bases de datos físicas. En esta arquitectura se pueden definir esquemas en los siguientes tres niveles:

1. El **nivel interno** tiene un **esquema interno**, que describe la estructura de almacenamiento físico de la base de datos. El esquema interno utiliza un modelo de datos físico y describe todos los detalles del almacenamiento de datos y las rutas de acceso a la base de datos.
2. El **nivel conceptual** tiene un **esquema conceptual**, que describe la estructura de toda la base de datos para una comunidad de usuarios. El esquema conceptual oculta los detalles de las estructuras de almacenamiento físico y se concentra en describir las entidades, los tipos de datos, las relaciones, las operaciones de los usuarios y las restricciones. Normalmente, el esquema conceptual se describe con un modelo de datos representativo cuando se implementa un sistema de bases de datos. Este *esquema conceptual de implementación* se basa a menudo en un *diseño de esquema conceptual* en un modelo de datos de alto nivel.

**Figura 2.2**. Arquitectura de tres esquemas.




[8] También se conoce como arquitectura ANSI/SPARC, según el comité que la propuso (Tsichritzis and Klug 1978).

3. El **nivel de vista** o **externo** incluye una cierta cantidad de **esquemas externos** o **vistas de usuario**. Un esquema externo describe la parte de la base de datos en la que un grupo de usuarios en particular está interesado y le oculta el resto de la base de datos. Como en el caso anterior, cada esquema externo se implementa normalmente mediante un modelo de datos representativo, posiblemente basado en un diseño de esquema externo de un modelo de datos de alto nivel.

La arquitectura de tres esquemas es una buena herramienta con la que el usuario puede visualizar los niveles del esquema de un sistema de bases de datos. La mayoría de los DBMSs no separan completa y explícitamente los tres niveles, pero soportan esta arquitectura en cierta medida. Algunos DBMSs pueden incluir en el esquema conceptual detalles a nivel físico. La arquitectura de tres niveles ANSI ocupa un lugar importante en el desarrollo de tecnologías de bases de datos porque separa el nivel externo de los usuarios, el nivel conceptual del sistema y el nivel de almacenamiento interno para diseñar una base de datos. Incluso hoy en día se aplica mucho al diseño de DBMSs. En la mayoría de los DBMSs que soportan vistas de usuario, los esquemas externos se especifican en el mismo modelo de datos que describe la información a nivel conceptual (por ejemplo, un DBMS relacional como Oracle utiliza SQL para esto). Algunos DBMSs permiten el uso de diferentes modelos de datos en los niveles conceptual y externo. Un ejemplo es Base de datos universal (UDB, *Universal Data Base*), un DBMS de IBM que utiliza el modelo relacional para describir el esquema conceptual, pero puede utilizar un modelo orientado a objetos para describir un esquema externo.

Observe que los tres esquemas sólo son *descripciones* de datos; los datos almacenados que existen en realidad están en el nivel físico. En un DBMS basado en la arquitectura de tres esquemas, cada grupo de usuarios sólo se refiere a su propio esquema externo. Por tanto, el DBMS debe transformar una solicitud especificada en un esquema externo en una solicitud contra el esquema conceptual, y después en una solicitud en el esquema interno para el procesamiento sobre la base de datos almacenada. Si la solicitud es para una recuperación de la base de datos, es preciso reformatear los datos extraídos de la base de datos almacenada para que concuerden o encajen en la vista externa del usuario. Los procesos para transformar solicitudes y resultados entre niveles se denominan mapeados. Estos mapeados pueden consumir bastante tiempo, por lo que algunos DBMS (sobre todo los que están pensados para bases de datos pequeñas) no soportan las vistas externas. No obstante, incluso en dichos sistemas se necesita algo de mapeado para transformar las solicitudes entre los niveles conceptual e interno.

### 2.2.2 Independencia de los datos

La arquitectura de tres esquemas se puede utilizar para explicar el concepto de **independencia de los datos**, que puede definirse como la capacidad de cambiar el esquema en un nivel de un sistema de bases de datos sin tener que cambiar el esquema en el siguiente nivel más alto. Se pueden definir dos tipos de independencia de datos:

1. **Independencia lógica de datos.** Es la capacidad de cambiar el esquema conceptual sin tener que cambiar los esquemas externos o los programas de aplicación. Es posible cambiar el esquema conceptual para expandir la base de datos (añadiendo un tipo de registro o un elemento de datos), para cambiar las restricciones o para reducir la base de datos (eliminando un tipo de registro o un elemento de datos). En el último caso, no deben verse afectados los esquemas externos que sólo se refieren a los datos restantes. Por ejemplo, el esquema externo de la Figura 1.5(a) no debe verse afectado por cambiar el archivo INFORME_CALIF (o tipo de registro) de la Figura 1.2 por el mostrado en la Figura 1.6(a). Sólo es necesario cambiar la definición de la vista y los mapeados en un DBMS que soporta la independencia lógica de datos. Una vez que el esquema conceptual sufre una reorganización lógica, los programas de aplicación que hacen referencia a las estructuras de esquema externo deben funcionar como antes. En el esquema conceptual se pueden introducir cambios en las restricciones sin que se vean afectados los esquemas externos o los programas de aplicación.
2. **Independencia física de datos.** Es la capacidad de cambiar el esquema interno sin que haya que cambiar el esquema conceptual. Por tanto, tampoco es necesario cambiar los esquemas externos.

Puede que haya que realizar cambios en el esquema interno porque algunos archivos físicos fueran reorganizados (por ejemplo, por la creación de estructuras de acceso adicionales) de cara a mejorar el rendimiento de las recuperaciones o las actualizaciones. Si en la base de datos permanecen los mismos datos que antes, no hay necesidad de cambiar el esquema conceptual. Por ejemplo, el suministro de una ruta de acceso para mejorar la velocidad de recuperación de los registros de sección (véase la Figura 1.2) por semestre y año no debe requerir modificar una consulta del tipo "*listar todas las secciones ofrecidas en otoño de 2004*", aunque el DBMS ejecutará la consulta con más eficacia utilizando la ruta de acceso nueva.

Por regla general, la independencia física de datos existe en la mayoría de las bases de datos y de los entornos de archivos en los que al usuario se le ocultan la ubicación exacta de los datos en el disco, los detalles hardware de la codificación del almacenamiento, la colocación, la compresión, la división, la fusión de registros, etcétera. Las aplicaciones siguen obviando estos detalles. Por el contrario, la independencia lógica de datos es muy difícil de conseguir porque permite los cambios estructurales y restrictivos sin afectar a los programas de aplicación (un requisito mucho más estricto).

Siempre que tengamos un DBMS de varios niveles, su catálogo debe ampliarse para incluir información de cómo mapear las consultas y los datos entre los diferentes niveles. El DBMS utiliza software adicional para acometer estos mapeados refiriéndose a la información de mapeado que hay en el catálogo. La independencia de datos ocurre porque cuando el esquema cambia a algún nivel, el esquema en el siguiente nivel más alto permanece inalterado; sólo cambia el *mapeado* entre los dos niveles. Por tanto, las aplicaciones que hacen referencia al esquema del nivel más alto no tienen que cambiar.

La arquitectura de tres esquemas puede facilitar el conseguir la independencia de datos verdadera, tanto física como lógica. Sin embargo, los dos niveles de mapeados crean un sobrecoste durante la compilación o la ejecución de una consulta o un programa, induciendo a deficiencias en el DBMS. Debido a esto, pocos DBMSs han implementado la arquitectura de tres esquemas completa.

# 2.3 Lenguajes e interfaces de bases de datos

En la Sección 1.4 explicamos la variedad de usuarios que un DBMS soporta. El DBMS debe proporcionar los lenguajes e interfaces apropiados para cada categoría de usuarios. En esta sección explicamos los tipos de lenguajes e interfaces proporcionados por un DBMS y las categorías de usuarios a las que se dirige cada interfaz.

## 2.3.1 Lenguajes DBMS

Una vez completado el diseño de una base de datos y elegido un DBMS para implementarla, el primer paso es especificar los esquemas conceptual e interno para la base de datos y cualesquiera mapeados entre los dos. En muchos DBMSs donde no se mantiene una separación estricta de niveles, el DBA y los diseñadores de la base de datos utilizan un lenguaje, denominado **lenguaje de definición de datos (DDL, *data definition language*)**, para definir los dos esquemas. El DBMS tendrá un compilador DDL cuya función es procesar las sentencias DDL a fin de identificar las descripciones de las estructuras del esquema y almacenar la descripción del mismo en el catálogo del DBMS.

En los DBMSs donde hay una clara separación entre los niveles conceptual e interno, se utiliza DDL sólo para especificar el esquema conceptual. Para especificar el esquema interno se utiliza otro lenguaje, el **lenguaje de definición de almacenamiento (SDL, *storage definition language*)**. Los mapeados entre los dos esquemas se pueden especificar en cualquiera de estos lenguajes. En la mayoría de los DBMSs relacionales actuales, no hay un lenguaje específico que asuma el papel de SDL. En cambio, el esquema interno se especifica mediante una combinación de parámetros y especificaciones relacionadas con el almacenamiento: el personal del DBA normalmente controla la indexación y la asignación de datos al almacenamiento. Para conseguir una

arquitectura de tres esquemas real se necesita un tercer lenguaje, el **lenguaje de definición de vistas (VDL, *view definition language*)**, a fin de especificar las vistas de usuario y sus mapeados al esquema conceptual, pero en la mayoría de los DBMSs se utiliza el DDL para definir tanto el esquema conceptual como el externo. En los DBMSs relacionales se utiliza SQL actuando como VDL para definir las vistas de usuario o de aplicación como resultado de las consultas predefinidas (consulte los Capítulos 8 y 9).

Una vez compilados los esquemas de la base de datos y rellenada ésta con datos, los usuarios deben disponer de algunos medios para manipularla. Entre las manipulaciones típicas podemos citar la recuperación, la inserción, el borrado y la modificación de datos. El DBMS proporciona un conjunto de operaciones o un lenguaje denominado **lenguaje de manipulación de datos (DML, *data manipulation language*)** para todas estas tareas.

En los DBMSs actuales, los tipos de lenguajes anteriormente citados normalmente *no están considerados como lenguajes distintos*; más bien, se utiliza un lenguaje integrado comprensivo que incluye construcciones para la definición del esquema conceptual, la definición de vistas y la manipulación de datos. La definición del almacenamiento normalmente se guarda aparte, ya que se utiliza para definir las estructuras de almacenamiento físico a fin de refinar el rendimiento del sistema de bases de datos, que normalmente lo lleva a cabo el personal del DBA. El lenguaje de bases de datos relacionales SQL es un ejemplo típico de lenguaje de bases de datos comprensible (consulte los Capítulos 8 y 9). SQL representa una combinación de DDL, VDL y DML, así como sentencias para la especificación de restricciones, la evolución del esquema y otras características. El SDL era un componente de las primeras versiones de SQL, pero se ha eliminado del lenguaje para mantenerlo únicamente en los niveles conceptual y externo.

Hay dos tipos principales de DML. Se puede utilizar un DML de **alto nivel** o **no procedimental** para especificar de forma concisa las operaciones complejas con las bases de datos. Muchos DBMS admiten sentencias DML de alto nivel mediante la introducción interactiva desde el monitor o terminal, o incrustadas en un lenguaje de programación de propósito general. En el último caso, las sentencias DML deben identificarse dentro del programa para que el precompilador las pueda extraer y el DBMS las pueda procesar. Un DML de **bajo nivel** o **procedimental** *debe* incrustarse en un lenguaje de programación de propósito general. Normalmente, este tipo de DML recupera registros individuales u objetos de la base de datos, y los procesa por separado. Por consiguiente, es preciso utilizar construcciones de un lenguaje de programación, como los bucles, para recuperar y procesar cada registro de un conjunto de registros. Los DMLs de bajo nivel también se conocen con el nombre de DMLs ***record-at-a-time*** (registro de una sola vez), debido a esta propiedad. DL/1, un DML diseñado para el modelo jerárquico, es un DML de bajo nivel que utiliza comandos como GET UNIQUE, GET NEXT o GET NEXT WITHIN PARENT para navegar de un registro a otro dentro de la jerarquía de registros de una base de datos. Los DMLs de alto nivel, como SQL, pueden especificar y recuperar muchos registros con una sola sentencia DML; por tanto, también se conocen como DML ***set-at-a-time*** o ***set-oriented*** (un conjunto de una sola vez, u orientado a conjuntos). Una consulta en un DML de alto nivel a menudo especifica los datos que hay que recuperar, en lugar de cómo recuperarlos; en consecuencia, dichos lenguajes también se conocen como **declarativos**.

Siempre que hay comandos DML, de alto o de bajo nivel, incrustados en un lenguaje de programación de propósito general, ese lenguaje se denomina **lenguaje *host***, y el DML **sublenguaje de datos**.[9] Por el contrario, un DML de alto nivel utilizado de forma interactiva independiente se conoce como **lenguaje de consulta**. En general, tanto los comandos de recuperación como los de actualización de un DML de alto nivel se pueden utilizar interactivamente y, por tanto, se consideran como parte del lenguaje de consulta.[10]

---

[9] En las bases de datos de objetos, los sublenguajes de *host* y datos normalmente forman un lenguaje integrado (por ejemplo, C++ con algunas extensiones a fin de soportar la funcionalidad de bases de datos). Algunos sistemas relacionales también proporcionan lenguajes integrados (por ejemplo, PL/SQL de Oracle).

[10] Según el significado en inglés de la palabra "*query*" (consulta), realmente se debería utilizar para describir sólo las recuperaciones, no las actualizaciones.

Los usuarios finales casuales normalmente utilizan un lenguaje de consulta de alto nivel para especificar sus consultas, mientras que los programadores utilizan el DML en su forma incrustada. Los usuarios principiantes y paramétricos normalmente utilizan **interfaces amigables para el usuario** para interactuar con la base de datos; los usuarios casuales, u otros usuarios, que quieren aprender los detalles de un lenguaje de consulta de alto nivel también pueden utilizar estas interfaces. A continuación explicamos los tipos de interfaces.

### 2.3.2 Interfaces de los DBMSs

El DBMS puede incluir las siguientes interfaces amigables para el usuario:

**Interfaces basadas en menús para los clientes web o la exploración.** Estas interfaces presentan al usuario listas de opciones (denominadas **menús**) que le guían por la formulación de una consulta. Los menús eliminan la necesidad de memorizar los comandos específicos y la sintaxis de un lenguaje de consulta; en cambio, la consulta se compone paso a paso eligiendo opciones de los menús visualizados por el sistema. Los menús desplegables son una técnica muy popular en las **interfaces de usuario basadas en la Web**. También se utilizan a menudo en las **interfaces de exploración**, que permiten al usuario examinar los contenidos de una base de datos de forma indagatoria y desestructurada.

**Interfaces basadas en formularios.** Una interfaz basada en formularios muestra un formulario a cada usuario. Los usuarios pueden rellenar las entradas del **formulario** para insertar datos nuevos, o rellenar únicamente ciertas entradas, en cuyo caso el DBMS recuperará los datos coincidentes para el resto de entradas. Normalmente, los formularios se diseñan y programan para los usuarios principiantes como interfaces para las transacciones enlatadas. Muchos DBMSs tienen **lenguajes de especificación de formularios**, que son lenguajes especiales que ayudan a los programadores a especificar dichos formularios. SQL*Forms es un lenguaje basado en formularios que especifica consultas utilizando un formulario diseñado en combinación con el esquema de la base de datos relacional. Oracle Forms es un componente de la suite de productos de Oracle que proporciona un amplio conjunto de características para diseñar y construir aplicaciones mediante formularios. Algunos sistemas tienen utilidades que definen un formulario dejando que el usuario final construya interactivamente en pantalla un formulario de ejemplo.

**Interfaces gráficas de usuario.** Una GUI normalmente muestra un esquema al usuario de forma esquemática. El usuario puede especificar entonces una consulta manipulando el diagrama. En muchos casos, las GUIs utilizan tanto menús como formularios. La mayoría de las GUIs utilizan un **dispositivo apuntador**, como el ratón, para elegir distintas partes del diagrama esquemático visualizado.

**Interfaces de lenguaje natural.** Estas interfaces aceptan consultas escritas en inglés u otro idioma e intentan *entenderlas*. Una interfaz de este tipo normalmente tiene su propio *esquema*, que es parecido al esquema conceptual de la base de datos, así como un diccionario de palabras importantes. La interfaz de lenguaje natural se refiere a las palabras de su esquema, así como al conjunto de palabras estándar de su diccionario, para interpretar la consulta. Si la interpretación es satisfactoria, la interfaz genera una consulta de alto nivel correspondiente a la consulta de lenguaje natural y la envía al DBMS para que la procese; de lo contrario, se inicia un diálogo con el usuario para clarificar la consulta. Las capacidades de las interfaces de lenguaje natural no avanzan rápidamente. En la actualidad vemos motores de búsqueda que aceptan cadenas de palabras en lenguajes naturales (por ejemplo, inglés o español) y las intentan emparejar con documentos de sitios específicos (en el caso de motores de búsqueda locales) o con páginas de la Web (para motores como Google o AskJeeves). Utilizan índices de palabras predefinidos y funciones de clasificación para recuperar y presentar los documentos resultantes según un grado decreciente de coincidencia. Estas interfaces de consulta textual de "formato libre" no son muy comunes en las bases de datos estructuradas relacionales o heredadas.

**Entrada y salida de lenguaje hablado.** Cada vez es más común el uso limitado del habla como consulta de entrada y como respuesta a una pregunta o como resultado de una consulta. Las aplicaciones con vocabularios limitados como las búsquedas en los directorios telefónicos, la salida/llegada de vuelos y la informa-

ción sobre el estado de las cuentas bancarias permiten la entrada y salida en lenguaje hablado para que las personas tengan acceso a esta información. La entrada de lenguaje hablado se detecta mediante una librería de palabras predefinidas que se utilizan para configurar los parámetros que se suministran a las consultas. Para la salida, se realiza una conversión parecida del texto o los números al lenguaje hablado.

**Interfaces para los usuarios paramétricos.** Los usuarios paramétricos, como los cajeros automáticos, a menudo tienen un pequeño conjunto de operaciones que se deben llevar a cabo repetidamente. Por ejemplo, un cajero puede utilizar teclas de función para invocar transacciones rutinarias y repetitivas, como depósitos o retiradas de las cuentas, o consultas de saldo. Los analistas de sistemas y los programadores diseñan e implementan una interfaz especial por cada clase de usuarios principiantes. Normalmente, se incluye un pequeño conjunto de comandos abreviados, con el objetivo de minimizar el número de pulsaciones necesarias por cada consulta. Por ejemplo, las teclas de función de un terminal se pueden programar para iniciar determinados comandos. Esto permite que el usuario paramétrico proceda con una cantidad mínima de pulsaciones.

**Interfaces para el DBA.** La mayoría de los sistemas de bases de datos contienen comandos privilegiados que sólo puede utilizar el personal del DBA. Entre ellos hay comandos para crear cuentas, configurar los parámetros del sistema, conceder la autorización de una cuenta, cambiar un esquema y reorganizar las estructuras de almacenamiento de una base de datos.

# 2.4 Entorno de un sistema de bases de datos

Un DBMS es un sistema de software complejo. En esta sección explicamos los tipos de componentes software que constituyen un DBMS y los tipos de software de computador con el que el DBMS interactúa.

## 2.4.1 Módulos componentes de un DBMS

La Figura 2.3 ilustra, de una forma sencilla, los componentes típicos de un DBMS. La figura está dividida en dos niveles: la mitad superior se refiere a los diversos usuarios del entorno de base de datos y sus interfaces; la mitad inferior muestra las "entrañas" del DBMS responsables del almacenamiento de datos y el procesamiento de transacciones.

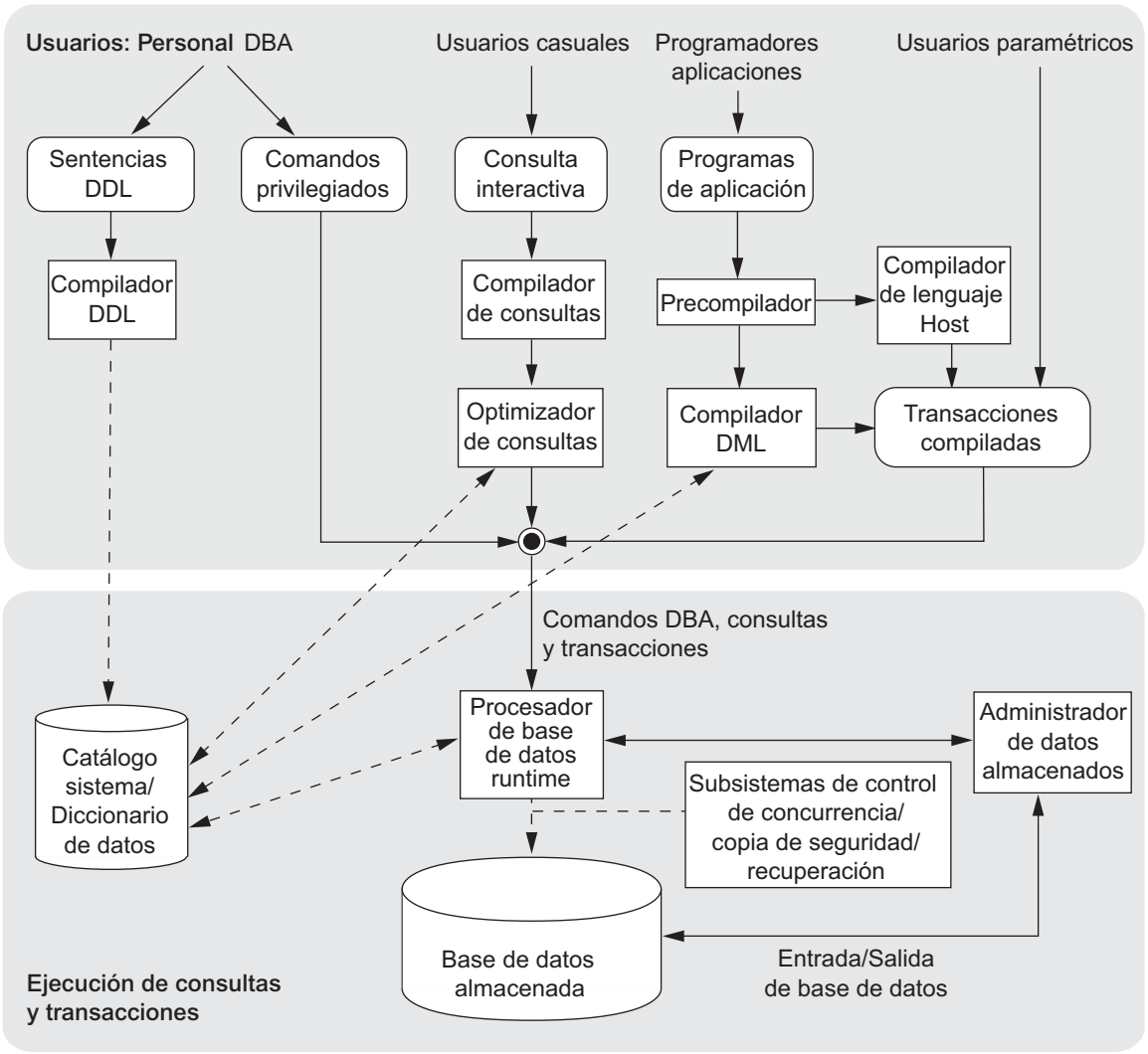
La base de datos y el catálogo del DBMS normalmente se almacenan en el disco. El acceso al disco está principalmente controlado por el **sistema operativo** (**SO**), que planifica la entrada/salida del disco. Un módulo **administrador de los datos almacenados** de alto nivel del DBMS controla el acceso a la información del DBMS almacenada en el disco, sea parte de la base de datos o del catálogo.

Vamos a ver primero la parte superior de la figura. Muestra las interfaces para el personal del DBA, los usuarios casuales que trabajan con interfaces interactivas para formular consultas, los programadores de aplicaciones que programan utilizando algunos lenguajes *host* y los usuarios paramétricos que realizan las entradas de datos suministrando parámetros a las transacciones predefinidas. El personal del DBA trabaja en definir la base de datos y refinarla introduciendo cambios en su definición mediante el DDL y otros comandos privilegiados.

El compilador DDL procesa las definiciones de esquema, especificadas en el DDL, y almacena las descripciones de los esquemas (metadatos) en el catálogo del DBMS. El catálogo incluye información como los nombres y los tamaños de los archivos, los nombres y los tipos de datos de los elementos de datos, los detalles del almacenamiento de cada archivo, la información de mapeado entre esquemas y las restricciones, además de muchos otros tipos de información que los módulos del DBMS necesitan. Los módulos software del DBMS buscan después en el catálogo la información que necesitan.

Los usuarios casuales y las personas con una necesidad ocasional de información de la base de datos interactúan utilizando alguna forma de interfaz, que mostramos, como la **interfaz de consulta interactiva**. No mostramos explícitamente ninguna interacción basada en menús o en formularios que puede utilizarse para

**Figura 2.3.** Módulos constituyentes de un DBMS y sus interacciones.




generar automáticamente la consulta interactiva. Un **compilador de consultas** analiza estas consultas sintácticamente para garantizar la corrección de las operaciones de los modelos, los nombres de los elementos de datos, etcétera, y luego lo compila todo en un formato interno. Esta consulta interna está sujeta a la optimización de la consulta que explicaremos en el Capítulo 15. Entre otras cosas, el **optimizador de consultas** se ocupa de la reconfiguración y la posible reordenación de operaciones, eliminación de redundancias y uso de los algoritmos e índices correctos durante la ejecución. Consulta el catálogo del sistema para información estadística y física acerca de los datos almacenados, y genera un código ejecutable que lleva a cabo las operaciones necesarias para la consulta y realiza las llamadas al procesador *runtime*.

Los programadores de aplicaciones escriben programas en lenguajes *host* como Java, C o COBOL, que son enviados a un **precompilador**. Éste extrae los comandos DML de un programa de aplicación escrito en un lenguaje de programación *host*.

Estos comandos se envían al compilador DML para su compilación en código objeto y así poder acceder a la base de datos. El resto del programa se envía al compilador de lenguaje *host*. Los códigos objeto para los

comandos DML y el resto del programa se enlazan, formando una transacción enlatada cuyo código ejecutable incluye llamadas al procesador de base de datos *runtime*. Estas transacciones enlatadas resultan de utilidad para los usuarios paramétricos que simplemente suministran los parámetros a dichas transacciones, de modo que pueden ejecutarse repetidamente como transacciones separadas. Un ejemplo es una transacción de retirada de fondos donde el número de cuenta y la cantidad pueden suministrarse como parámetros.

En la parte inferior de la Figura 2.3 aparece el procesador de bases de datos *runtime* para ejecutar (1) los comandos privilegiados, (2) los proyectos de consultas ejecutables y (3) las transacciones enlatadas con parámetros *runtime*. Trabaja con el diccionario del sistema y se puede actualizar con estadísticas. Funciona con el administrador de datos almacenados, que a su vez utiliza servicios básicos del sistema operativo para ejecutar operaciones de entrada/salida de bajo nivel entre el disco y la memoria principal. Se encarga de otros aspectos de la transferencia de datos como la administración de los búferes en la memoria principal. Algunos DBMSs tienen su propio módulo de administración de búfer, mientras que otros dependen del SO para dicha administración. En esta figura hemos puesto por separado el control de la concurrencia y los sistemas de copia de seguridad y recuperación como un módulo. Con propósitos de administración de transacciones, están integrados en el funcionamiento del procesador de bases de datos *runtime*.

Ahora es normal tener el **programa cliente** que accede al DBMS ejecutándose en un computador diferente al que alberga la base de datos. El primero se denomina **computador cliente** y ejecuta un cliente DBMS, y el último se denomina **servidor de bases de datos**. En algunos casos, el cliente accede a un computador intermedio, denominado **servidor de aplicaciones**, que a su vez accede al servidor de bases de datos. Elaboraremos este tema en la Sección 2.5.

La Figura 2.3 no tiene la intención de describir un DBMS específico; en cambio, ilustra los módulos DBMS típicos. El DBMS interactúa con el sistema operativo cuando se necesita acceso al disco (a la base de datos o al catálogo). Si varios usuarios comparten el computador, el SO planificará las peticiones de acceso al disco DBMS y el procesamiento DBMS junto con otros procesos. Por el contrario, si el computador está principalmente dedicado a ejecutar el servidor de bases de datos, el DBMS controlará el *buffering* de memoria principal de las páginas de disco. El DBMS también interactúa con los compiladores en el caso de lenguajes de programación *host* de propósito general, y con los servidores de aplicaciones y los programas cliente que se ejecutan en máquinas separadas a través de la interfaz de red del sistema.

## 2.4.2 Utilidades del sistema de bases de datos

Además de poseer los módulos software recientemente descritos, la mayoría de los DBMSs tienen utilidades de bases de datos que ayudan al DBA a administrar el sistema de bases de datos. Las funciones de las utilidades más comunes son las siguientes:

- **Carga**. La carga de los archivos de datos existentes (como archivos de texto o archivos secuenciales) en la base de datos se realiza con una utilidad de carga. Normalmente, a la utilidad se le especifican el formato (origen) actual del archivo de datos y la estructura del archivo de base de datos (destino) deseada; después, reformatea automáticamente los datos y los almacena en la base de datos. Con la proliferación de DBMSs, la transferencia de datos de un DBMS a otro es cada vez más común en muchas empresas. Algunos fabricantes están ofreciendo productos que generan los programas de carga apropiados, dando las descripciones de almacenamiento de base de datos de origen y de destino existentes (esquemas internos). Estas herramientas también se conocen como **herramientas de conversión**. Para el DBMS jerárquico denominado IMS (IBM) y para muchos DBMSs de red como IDMS (Computer Associates), SUPRA (Cincom) o IMAGE (HP), los fabricantes o terceros están desarrollando toda una variedad de herramientas de conversión (por ejemplo, SUPRA Server SQL de Cincom) para transformar los datos en el modelo relacional.
- **Copia de seguridad**. Una utilidad de copia de seguridad crea una copia de respaldo de la base de datos, normalmente descargando la base de datos entera en una cinta. La copia de seguridad se puede utilizar

para restaurar la base de datos en caso de un fallo desastroso. También se suelen utilizar las copias de seguridad incrementales, con las que sólo se hace copia de los cambios experimentados por la base de datos desde la última copia. La copia de seguridad incremental es más compleja, pero ahorra espacio.

- **Reorganización del almacenamiento de la base de datos**. Esta utilidad se puede utilizar para reorganizar un conjunto de archivos de bases de datos en una organización de archivos diferente a fin de mejorar el rendimiento.
- **Monitorización del rendimiento**. Una utilidad de este tipo monitoriza el uso de la base de datos y ofrece estadísticas al DBA. Este último utiliza las estadísticas para tomar decisiones, como si debe o no reorganizar los archivos, o si tiene que añadir o eliminar índices para mejorar el rendimiento.

Hay otras utilidades para ordenar archivos, manipular la compresión de datos, monitorizar el acceso de los usuarios, interactuar con la red y llevar a cabo otras funciones.

### 2.4.3 Herramientas, entornos de aplicación e instalaciones de comunicaciones

También existen otras herramientas para los diseñadores de bases de datos, usuarios y DBMS. Las herramientas CASE[11] se utilizan en la fase de diseño de los sistemas de bases de datos. Otra herramienta que puede resultar muy útil en empresas grandes es un **sistema de diccionario de datos** (o **almacén de datos**) ampliado. Además de almacenar información de catálogo sobre esquemas y restricciones, el diccionario de datos almacena otra información, como decisiones de diseño, uso de estándares, descripciones de las aplicaciones e información de usuario. Dicho sistema también se denomina **almacén de información**. Los usuarios o el DBA pueden acceder *directamente* a esta información siempre que lo necesiten. Una utilidad de diccionario de datos es parecida al catálogo del DBMS, pero incluye una amplia variedad de información a la que acceden principalmente los usuarios, más que el software de DBMS.

Los **entornos de desarrollo de aplicaciones**, como PowerBuilder (Sybase) o JBuilder (Borland), son cada vez más populares. Estos sistemas proporcionan un entorno para el desarrollo de aplicaciones de bases de datos e incluyen servicios que ayudan en muchos de los aspectos de los sistemas de bases de datos, como el diseño de la base de datos, el desarrollo de la GUI, las consultas y las actualizaciones, y el desarrollo de una aplicación.

El DBMS también necesita interactuar con **software de comunicaciones**, cuya función es permitir a los usuarios de ubicaciones alejadas (remotas) del sistema de bases de datos acceder a la base de datos a través de sus terminales, estaciones de trabajo o computadores personales. La conexión de estos usuarios al sitio de la base de datos se realiza a través de hardware de comunicaciones de datos, como las líneas telefónicas, redes de larga distancia, redes locales o dispositivos de comunicaciones por satélite. Muchos sistemas de bases de datos tienen paquetes de comunicaciones que trabajan con el DBMS. El sistema integrado por el DBMS y el sistema de comunicaciones de datos se conoce como sistema **DB/DC**. Además, algunos DBMSs distribuidos se distribuyen físicamente entre varias máquinas. En este caso, son necesarias las redes de comunicaciones para conectar esas máquinas. En ocasiones se trata de **redes de área local** (**LAN**), pero también pueden ser redes de otros tipos.

[11] Aunque CASE significa "*computer-aided software engineering*" (ingeniería de software asistida por computador), muchas de las herramientas CASE se utilizan principalmente para el diseño de bases de datos.

# 2.5 Arquitecturas cliente/servidor centralizadas para los DBMSs

## 2.5.1 Arquitectura centralizada de los DBMSs

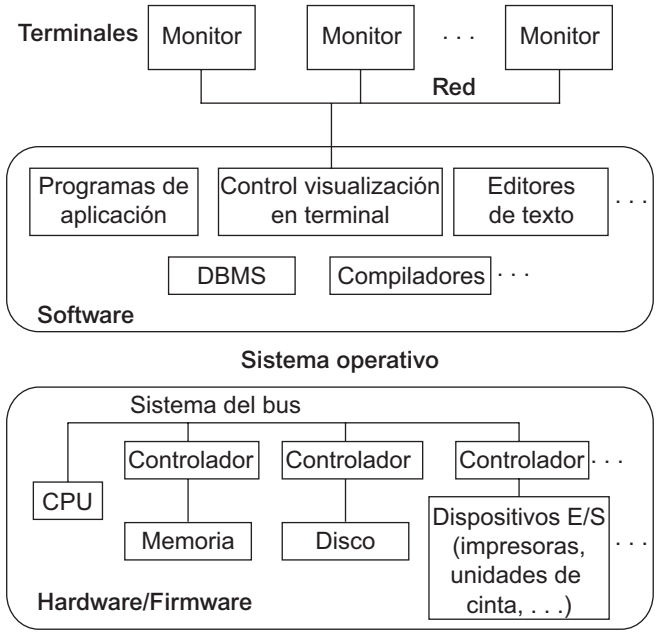
Las arquitecturas de los DBMSs han seguido tendencias parecidas a las arquitecturas de los sistemas de computación generales. Las arquitecturas primigenias utilizaban *mainframes* para proporcionar el procesamiento principal a todas las funciones del sistema, incluyendo las aplicaciones de usuario y los programas de interfaz de usuario, así como a toda la funcionalidad del DBMS. La razón era que la mayoría de los usuarios accedía a esos sistemas a través de terminales de computador que no tenían potencia de procesamiento y sólo ofrecían capacidades de visualización. Por tanto, todo el procesamiento se realizaba remotamente en el sistema computador, y sólo se enviaba la información de visualización y los controles desde el computador a los terminales de visualización, que estaban conectados con el computador central a través de diferentes tipos de redes de comunicaciones.

A medida que bajaban los precios del hardware, la mayoría de los usuarios reemplazaban sus terminales por PCs y estaciones de trabajo. Al principio, los sistemas de bases de datos utilizaban esos computadores de un modo parecido a como utilizaban los terminales de visualización, de modo que el DBMS seguía siendo un DBMS **centralizado** en el que toda la funcionalidad DBMS, ejecución de aplicaciones e interacción con el usuario se llevaba a cabo en una máquina. La Figura 2.4 ilustra los componentes físicos de una arquitectura centralizada. Gradualmente, los sistemas DBMS empezaron a aprovecharse de la potencia de procesamiento disponible en el lado del usuario, lo que llevó a las arquitecturas DBMS cliente/servidor.

## 2.5.2 Arquitecturas cliente/servidor básicas

En primer lugar vamos a ver la arquitectura cliente/servidor en general, para luego ver cómo se aplica a los DBMSs. La **arquitectura cliente/servidor** se desarrolló para ocuparse de los entornos de computación en los

**Figura 2.4.** Arquitectura centralizada física.



que una gran cantidad de PCs, estaciones de trabajo, servidores de archivos, impresoras, servidores de bases de datos, servidores web y otros equipos están conectados a través de una red. La idea es definir **servidores especializados** con funcionalidades específicas. Por ejemplo, es posible conectar varios PCs o estaciones de trabajo pequeñas como clientes a un servidor de archivos que mantiene los archivos de las máquinas cliente. Otra máquina puede designarse como **servidor de impresión** conectándola a varias impresoras; después, todas las peticiones de impresión procedentes de los clientes se envían a esta máquina. Los **servidores web** o **servidores de e-mail** también han caído en la categoría de servidores especializados. De este modo, muchas máquinas cliente pueden acceder a los recursos proporcionados por servidores especializados. Las **máquinas cliente** proporcionan al usuario las interfaces apropiadas para utilizar estos servidores, así como potencia de procesamiento local para ejecutar aplicaciones locales. Este concepto se puede llevar al software, donde los programas especializados (como un DBMS o un paquete CAD [diseño asistido por computador]) se almacenan en servidores específicos a los que acceden multitud de clientes. La Figura 2.5 ilustra una arquitectura cliente/servidor en el nivel lógico; la Figura 2.6 es un diagrama simplificado que muestra la arquitectura física. Algunas máquinas sólo serían sitios cliente (por ejemplo, estaciones de trabajo sin discos o estaciones/PCs con discos que sólo tienen instalado el software cliente). Otras máquinas serían servidores dedicados, y otras tendrían funcionalidad de cliente y servidor.

El concepto de arquitectura cliente/servidor asume una estructura subyacente consistente en muchos PCs y estaciones de trabajo, así como una pequeña cantidad de máquinas *mainframe*, conectadas a través de LANs y otros tipos de redes de computadores. En esta estructura, un cliente es normalmente la máquina de un usua-

**Figura 2.5.** Arquitectura cliente/servidor lógica de dos capas.

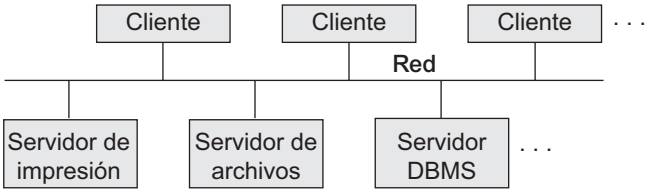
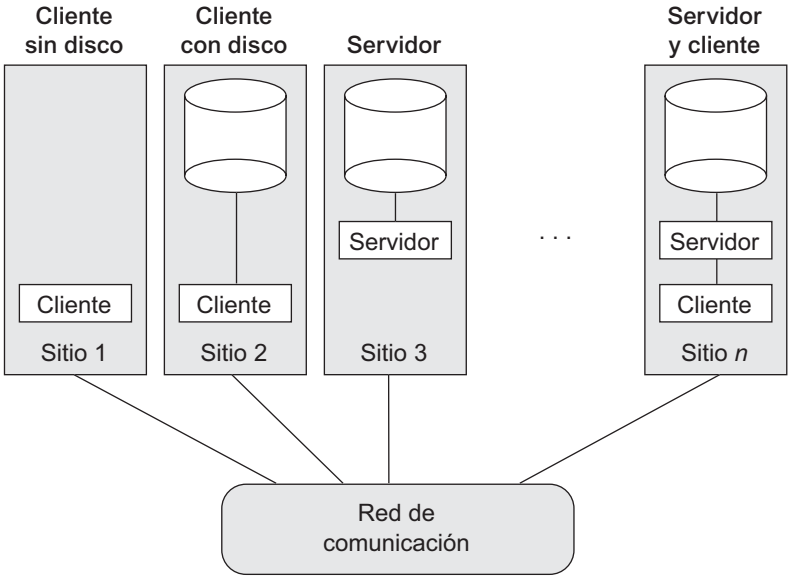



**Figura 2.6.** Arquitectura cliente/servidor física de dos capas.



rio que proporciona capacidad de interfaz de usuario y procesamiento local. Cuando un cliente requiere acceso a funcionalidad adicional (por ejemplo, acceso a una base de datos) que no existe en esa máquina, conecta con un servidor que ofrece la funcionalidad necesaria. Un servidor es un sistema que contiene hardware y software que pueden proporcionar servicios a los computadores cliente, como acceso a archivos, impresión, archivado o acceso a bases de datos. En el caso general, algunas máquinas sólo instalan el software cliente, mientras otras sólo instalan el software servidor, y otras pueden incluir los dos (véase la Figura 2.6). No obstante, lo más normal es que el software cliente y el software servidor se ejecuten en máquinas separadas. Los dos tipos principales de arquitecturas DBMS básicas se crearon sobre esta estructura cliente/servidor fundamental: dos capas y tres capas.[12] Las explicamos a continuación.

### 2.5.3 Arquitecturas cliente/servidor de dos capas para los DBMSs

La arquitectura cliente/servidor se está incorporando progresivamente a los paquetes DBMS comerciales. En los sistemas de administración de bases de datos relacionales (RDBMSs), muchos de los cuales empezaron como sistemas centralizados, los primeros componentes del sistema que se movieron al lado del cliente fueron la interfaz de usuario y las aplicaciones. Como SQL (consulte los Capítulos 8 y 9) ofrecía un lenguaje estándar para los RDBMSs, se creó un punto de división lógica entre cliente y servidor. Por tanto, la funcionalidad de consulta y transacción relacionada con el procesamiento SQL permanece en el lado del servidor. En semejante estructura, el servidor se denomina a menudo **servidor de consultas** o **servidor de transacciones** porque proporciona estas dos funcionalidades. En un RDBMS el servidor también se conoce como **servidor SQL**.

En una arquitectura cliente/servidor, los programas de interfaz de usuario y los programas de aplicación se pueden ejecutar en el lado del cliente. Cuando se necesita acceso DBMS, el programa establece una conexión con el DBMS (que se encuentra en el lado del servidor); una vez establecida la conexión, el programa cliente puede comunicarse con el DBMS. El estándar **Conectividad abierta de bases de datos (ODBC, *Open Database Connectivity*)** proporciona una **interfaz de programación de aplicaciones (API, *application programming interface*)**, que permite a los programas del lado del cliente llamar al DBMS, siempre y cuando las máquinas cliente y servidor tengan instalado el software necesario. La mayoría de los fabricantes de DBMSs proporcionan controladores ODBC para sus sistemas. Un programa cliente puede conectar realmente con varios RDBMSs y enviar solicitudes de consulta y transacción utilizando la API ODBC, que después son procesadas en los sitios servidor. Los resultados de una consulta se envían de regreso al programa cliente, que procesará o visualizará los resultados según las necesidades. También se ha definido un estándar relacionado con el lenguaje de programación Java, **JDBC**, que permite a los programas Java cliente acceder al DBMS a través de una interfaz estándar.

Algunos DBMSs orientados a objetos adoptaron la segunda metodología de arquitectura cliente/servidor: los módulos software del DBMS se dividían entre cliente y servidor de un modo más integrado. Por ejemplo, el **nivel servidor** puede incluir la parte del software DBMS responsable de manipular los datos en las páginas del disco, controlar la concurrencia local y la recuperación, almacenar en búfer y caché las páginas de disco, y otras funciones parecidas. Entretanto, el **nivel cliente** puede manipular la interfaz de usuario; las funciones del diccionario de datos; las interacciones DBMS con los compiladores de lenguajes de programación; la optimización de consultas globales, el control de la concurrencia y la recuperación entre varios servidores; la estructuración de objetos complejos a partir de los datos almacenados en los búferes; y otras funciones parecidas. En esta metodología, la interacción cliente/servidor es más estrecha y la realizan internamente los módulos DBMS (algunos de los cuales residen en el cliente y otros en el servidor), en lugar de los usuarios. La división exacta de la funcionalidad varía de un sistema a otro. En semejante arquitectura cliente/servidor, el servidor se llama **servidor de datos** porque proporciona datos de las páginas de disco al cliente. El software DBMS del lado del cliente estructura después estos datos como objetos para los programas cliente.

[12] Hay otras muchas variaciones de las arquitecturas cliente/servidor. Explicaremos las dos más básicas.

Las arquitecturas descritas aquí se llaman **arquitecturas de dos capas** porque los componentes software están distribuidos en dos sistemas: cliente y servidor. Las ventajas de esta arquitectura son su simplicidad y su perfecta compatibilidad con los sistemas existentes. La aparición de la Web cambió los roles de clientes y servidor, lo que condujo a la arquitectura de tres capas.

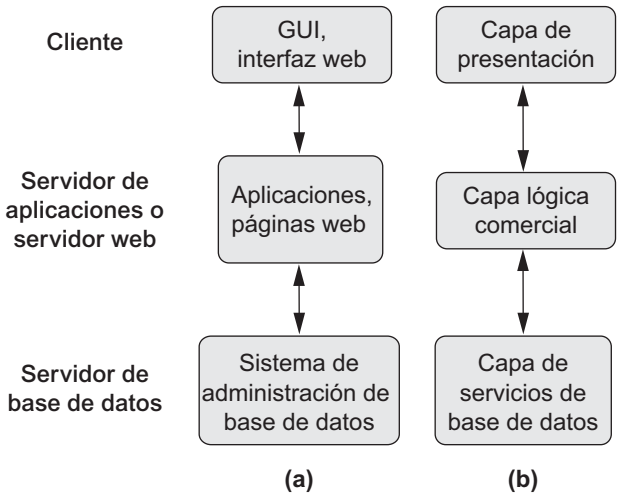
### 2.5.4 Arquitecturas de tres capas y *n* capas para las aplicaciones web

Muchas aplicaciones web utilizan una arquitectura denominada de tres capas, que añade una capa intermedia entre el cliente y el servidor de la base de datos, como se ilustra en la Figura 2.7(a).

Esta capa **intermedia** se denomina a veces **servidor de aplicaciones** y, en ocasiones, **servidor web**, en función de la aplicación. Este servidor juega un papel intermedio almacenando las reglas comerciales (procedimientos o restricciones) que se utilizan para acceder a los datos del servidor de bases de datos. También puede mejorar la seguridad de la base de datos al comprobar las credenciales del cliente antes de enviar una solicitud al servidor de la base de datos.

Los clientes contienen interfaces GUI y algunas reglas comerciales adicionales específicas de la aplicación. El servidor intermedio acepta solicitudes del cliente, las procesa y envía comandos de bases de datos al servidor de bases de datos, y después actúa como un conducto para pasar datos procesados (parcialmente) desde el servidor de bases de datos a los clientes, donde son procesados de forma más avanzada para su presentación en formato GUI a los usuarios. De este modo, la interfaz de usuario, las reglas de aplicación y el acceso de datos actúan como las tres capas. La Figura 2.7(b) muestra otra arquitectura utilizada por las bases de datos y otros fabricantes de paquetes de aplicaciones. La capa de presentación muestra información al usuario y permite la entrada de datos. La capa lógica comercial manipula las reglas intermedias y las restricciones antes de que los datos sean pasados hacia arriba hasta el usuario, o hacia abajo, hasta el DBMS. La capa inferior incluye todos los servicios de administración de datos. Si la capa inferior está dividida en dos capas (un servidor web y un servidor de bases de datos), entonces tenemos una arquitectura de cuatro capas. Es costumbre dividir las capas entre el usuario y los datos almacenados en componentes aún más sutiles, para de este modo llegar a arquitecturas de *n* capas, donde *n* puede ser cuatro o cinco. Normalmente, la capa lógica comercial está dividida en varias capas. Además de distribuir la programación y los datos por la red, las aplicaciones de *n*

**Figura 2.7.** Arquitectura cliente/servidor lógica de tres capas, con un par de nomenclaturas comúnmente utilizadas.



capas ofrecen la ventaja de que cualquiera de las capas se puede ejecutar en un procesador adecuado o plataforma de sistema operativo, además de poderse manipular independientemente. Otra capa que los fabricantes de paquetes ERP (planificación de recursos empresariales) y CRM (administración de la relación con el cliente) suelen utilizar es la *capa middleware*, que da cuenta de los módulos *front-end* que comunican con una determinada cantidad de bases de datos *back-end*.

Los avances en la tecnología de cifrado y descifrado hace más segura la transferencia de datos sensibles cifrados desde el servidor hasta el cliente, donde se descifran. Esto último lo puede hacer hardware o software avanzado. Esta tecnología otorga unos niveles altos de seguridad en los datos, aunque los problemas de seguridad en las redes siguen siendo la principal inquietud. Distintas tecnologías de compresión de datos también ayudan a transferir grandes cantidades de datos desde los servidores hasta los clientes a través de redes cableadas e inalámbricas.

## 2.6 Clasificación de los sistemas de administración de bases de datos

Normalmente se utilizan varios criterios para clasificar los DBMSs. El primero es el **modelo de datos** en el que el DBMS está basado. El **modelo de datos relacional** es el modelo de datos principal que se utiliza en muchos de los DBMSs comerciales actuales. En algunos sistemas comerciales se ha implantado el **modelo de datos de objetos**, pero su uso no se ha extendido. Muchas aplicaciones heredadas todavía se ejecutan en sistemas de bases de datos basados en los **modelos de datos jerárquicos** y **de red**. IMS (IBM) y algunos otros sistemas como System 2K (SAS Ic.) o TDMS son ejemplos de DBMS jerárquicos, que no tuvieron mucho éxito comercial. IMS continúa siendo un actor muy importante entre los DBMSs en uso en instalaciones gubernamentales e industriales, incluyendo hospitales y bancos. Muchos fabricantes utilizaron el modelo de datos de red, y los productos resultantes, como IDMS (Cullinet; ahora, Computer Associates), DMS 1100 (Univac; ahora, Unisys), IMAGE (Hewlett-Packard), VAX-DBMS (Digital; ahora, Compaq) y SUPRA (Cincom) todavía tienen partidarios y sus grupos de usuarios cuentan con organizaciones propias activas. Si a esta lista añadimos el popular sistema de archivos VSAM de IBM, podemos decir que (en el momento de escribir esto) más del 50% de los datos computerizados en todo el mundo se encuentran en estos así llamados **sistemas de bases de datos heredados**.

Los DBMSs relacionales están evolucionando constantemente y, en particular, han ido incorporando muchos de los conceptos que se desarrollaron en las bases de datos de objetos. Esto ha conducido a una nueva clase de DBMSs denominados DBMSs objeto-relacional (objetos relacionales). Los DBMSs se pueden clasificar basándose en el modelo de datos: relacional, objeto, objeto-relacional, jerárquico, de red, y otros.

El segundo criterio que se utiliza para clasificar los DBMSs es el **número de usuarios** soportado por el sistema. Los **sistemas de un solo usuario** sólo soportan un usuario al mismo tiempo y se utilizan principalmente con los PCs. Los **sistemas multiusuario**, que incluyen la mayoría de los DBMSs, soportan varios usuarios simultáneamente.

El tercer criterio es el **número de sitios** sobre los que se ha distribuido la base de datos. Un DBMS es **centralizado** si los datos están almacenados en un solo computador. Un DBMS centralizado puede soportar varios usuarios, pero el DBMS y la base de datos residen en su totalidad en un solo computador. Un **DBMS distribuido** (DDBMS) puede tener la base de datos y el software DBMS distribuidos por muchos sitios, conectados por una red de computadores. Los **DBMSs homogéneos** utilizan el mismo software DBMS en varios sitios. Una tendencia reciente es desarrollar software para acceder a varias bases de datos autónomas preexistentes almacenadas en DBMSs homogéneos. Esto lleva a un **DBMS federado** (o **sistema multibase de datos**), en el que los DBMSs participantes se acoplan y tienen cierto grado de autonomía local. Muchos DBMSs utilizan una arquitectura cliente/servidor.

El cuarto criterio es el coste. Es muy difícil proponer una clasificación de los DBMSs atendiendo al coste. Actualmente, tenemos los productos DBMS de código fuente abierto (gratuito), como MYSQL y PostgreSQL, soportados por terceros con servicios adicionales. Los principales productos RDBMS están disponibles como versiones de prueba válidas para 30 días, así como versiones personales que pueden costar menos de 100 dólares y permitir una funcionalidad considerable. Los sistemas gigantes se están vendiendo de forma modular con componentes para la distribución, duplicación, procesamiento paralelo, capacidad móvil, etcétera; su configuración se realiza mediante la definición de una gran cantidad de parámetros. Además, se venden como licencias (las licencias para un sitio permiten un uso ilimitado del sistema de bases de datos con cualquier número de copias ejecutándose en el sitio cliente). Otro tipo de licencia limita el número de usuarios simultáneos o el número de usuarios sentados en una ubicación. Las versiones para un solo usuario de algunos sistemas como ACCESS se venden por copia, o se incluyen en la configuración global de un escritorio o portátil. Además, el almacenamiento de datos y el minado, así como el soporte de tipos adicionales de datos, tienen un coste adicional. Es muy corriente pagar millones anualmente por la instalación y el mantenimiento de los sistemas de bases de datos.

También podemos clasificar un DBMS según los **tipos de rutas de acceso** para almacenar archivos. Una familia de DBMSs bien conocida está basada en estructuras de archivos inversas. Por último, un DBMS puede ser de **propósito general** o de **propósito especial**. Cuando la principal consideración es el rendimiento, se puede diseñar y construir un DBMS de propósito especial para una aplicación específica; dicho sistema no se puede utilizar para otras aplicaciones sin introducir cambios importantes. Muchos sistemas de reservas en aerolíneas y de directorios telefónicos desarrollados en el pasado son DBMSs de propósito especial. Esto cae en la categoría de los sistemas de **procesamiento de transacciones en línea** (**OLTP**, ***online transaction processing***), que deben soportar una gran cantidad de transacciones simultáneas sin imponer unos retrasos excesivos.

Permítanos elaborar brevemente el criterio principal para clasificar los DBMSs: el modelo de datos. El modelo de datos relacional básico representa una base de datos como una colección de tablas, donde cada tabla se puede almacenar como un archivo separado. La base de datos de la Figura 1.2 parece una representación relacional. La mayoría de las bases de datos relacionales utilizan el lenguaje de consulta de alto nivel SQL y soportan un formato limitado de vistas de usuario. En los Capítulos 5 a 9 explicaremos el modelo relacional, sus lenguajes y operaciones, así como las técnicas para programar aplicaciones relacionales.

El modelo de datos de objetos define una base de datos en términos de objetos, sus propiedades y sus operaciones. Los objetos con la misma estructura y comportamiento pertenecen a una **clase**, y las clases están organizadas en **jerarquías** (o **gráficos acíclicos**). Las operaciones de cada clase son específicas en términos de procedimientos predefinidos denominados **métodos**. Los DBMSs relacionales han ido ampliando sus modelos para incorporar conceptos de bases de datos de objetos y otras capacidades; estos sistemas se conocen como **sistemas objeto-relacional** o **sistemas relacionales extendidos**. En los Capítulos 20 a 22 explicaremos los sistemas de bases de datos de objetos y los sistemas objeto-relacional.

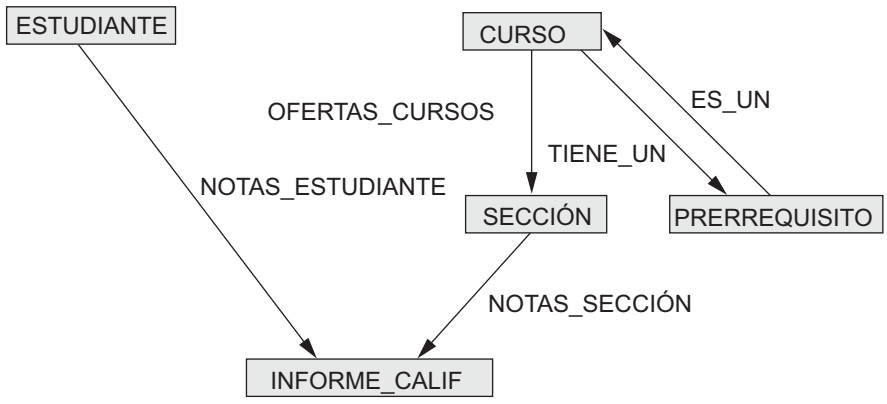
Dos modelos de datos más antiguos e históricamente importantes, ahora conocidos como modelos de datos heredados, son los modelos de red y jerárquico. El **modelo de red** representa los datos como tipos de registros, y también representa un tipo limitado de relación 1:N, denominado **tipo conjunto**. Una relación 1:N, o uno-a-muchos, relaciona una instancia de un registro con muchas instancias de registro mediante algún mecanismo de punteros en esos modelos. La Figura 2.8 muestra el diagrama del esquema de una red para la base de datos de la Figura 1.2, donde los tipos de registros se muestran como rectángulos y los tipos conjunto como flechas de dirección etiquetadas. El modelo de red, también conocido como modelo CODASYL DBTG,[13] tiene un lenguaje *record-at-a-time* asociado que debe incrustarse en un lenguaje de programación *host*. El DML de red se propuso en el informe Database Task Group (DBTG) de 1971 como una extensión del

[13] CODASYL DBTG significa "*Conference on Data Systems Languages Database Task Group*", que es el comité que especificó el modelo de red y su lenguaje.

lenguaje COBOL. Proporciona comandos para localizar registros directamente (por ejemplo, FIND ANY <*tipo-registro*> USING <*lista-campos*>, o FIND DUPLICATE <*tipo-registro*> USING <*lista-campos*>). Tiene comandos para soportar conversiones dentro de los tipos conjunto (por ejemplo, GET OWNER, GET {FIRST, NEXT, LAST} MEMBER WITHIN <*tipo-conjunto*> WHERE <*condición*>). También tiene comandos para almacenar datos nuevos (por ejemplo, STORE <*tipo-registro*>) y convertirlos en parte de un tipo conjunto (por ejemplo, CONNECT <*tipo-registro*> TO <*tipo-conjunto*>). El lenguaje también manipula muchas consideraciones adicionales como los tipos de registro y los tipos conjunto, que están definidos por la posición actual del proceso de navegación dentro de la base de datos. Actualmente, lo utilizan principalmente los DBMSs IDMS, IMAGE y SUPRA. El **modelo jerárquico** representa los datos como estructuras en forma de árboles jerárquicos. Cada jerarquía representa una cantidad de registros relacionados. No hay ningún lenguaje estándar para el modelo jerárquico. Un DML jerárquico popular es DL/1 del sistema IMS, que dominó el mercado de los DBMSs durante más de 20 años, entre 1965 y 1985, y es un DBMS que incluso hoy en día sigue utilizándose ampliamente, manteniendo un porcentaje muy alto de datos en bases de datos gubernamentales, sanitarias, bancarias y aseguradoras. Su DML, denominado DL/1, fue el estándar industrial de facto durante mucho tiempo. DL/1 tiene comandos para localizar un registro (por ejemplo, GET {UNIQUE, NEXT} <*tipo-registro*> WHERE <*condición*>). También ofrece funciones de navegación para navegar por las jerarquías (por ejemplo, GET NEXT WITHIN PARENT o GET {FIRST, NEXT} PATH <*especificación-ruta-jerárquica*> WHERE <*condición*>). Tiene los medios apropiados para almacenar y actualizar registros (por ejemplo, INSERT <*tipo-registro*>, REPLACE <*tipo-registro*>). En los Apéndices E y F veremos una breve panorámica de los modelos de red y jerárquico.[14]

El **modelo de lenguaje de marcado extendido** (**XML**, ***eXtended Markup Language***), ahora considerado el estándar para el intercambio de datos por Internet, también utiliza estructuras en forma de árbol jerárquico. Combina los conceptos de bases de datos con conceptos procedentes de los modelos de representación de documentos. Los datos se representan como elementos; con el uso de etiquetas, los datos se pueden anidar para crear estructuras jerárquicas complejas. Este modelo se parece conceptualmente al modelo de objetos, pero utiliza una terminología diferente. En el Capítulo 27 explicaremos XML y veremos cómo se relaciona con las bases de datos.

**Figura 2.8.** El esquema de la Figura 2.1 en la notación del modelo de red.




[14] Los capítulos completos sobre los modelos de red y jerárquico de la segunda edición de este libro están disponibles en el sitio web complementario: www.librosite.net/elmasri.

## 2.7 Resumen

En este capítulo hemos introducido los principales conceptos que se utilizan en los sistemas de bases de datos. Hemos definido un modelo de datos y distinguido tres categorías principales:

- Modelos de datos de alto nivel o conceptuales (basados en entidades y relacionales).
- Modelos de datos de nivel bajo o físicos.
- Modelos de datos representativos o de implementación (basados en registros, orientados a objetos).

Distinguimos el esquema, o descripción de una base de datos, de la propia base de datos. El esquema no cambia muy a menudo, mientras que el estado de la base de datos cambia cada vez que se insertan, eliminan o modifican datos. Después describimos la arquitectura DBMS de tres esquemas, que permite tres niveles de esquema:

- Un esquema interno describe la estructura del almacenamiento físico de la base de datos.
- Un esquema conceptual es una descripción de nivel alto de toda la base de datos.
- Los esquemas externos describen las vistas de los diferentes grupos de usuarios.

Un DBMS que separa limpiamente los tres niveles debe tener mapeados entre los esquemas para transformar las solicitudes y los resultados de un nivel al siguiente. La mayoría de los DBMSs no separan completamente los tres niveles. Utilizamos la arquitectura de tres esquemas para definir los conceptos de la independencia lógica y física de datos.

Después explicamos los principales tipos de lenguajes e interfaces que los DBMSs soportan. Se utiliza un lenguaje de definición de datos (DDL) para definir el esquema conceptual de la base de datos. En la mayoría de los DBMSs, el DDL también define las vistas de los usuarios y, algunas veces, las estructuras de almacenamiento; en otros DBMSs, pueden existir lenguajes separados (VDL, SDL) para especificar las vistas y las estructuras de almacenamiento. Esta distinción desaparece en las implementaciones relacionales actuales con SQL haciendo el papel de lenguaje común para muchos papeles, como la definición de vistas. La parte de definición del almacenamiento (SDL) se incluyó en muchas versiones de SQL, pero ahora ha quedado relegado a comandos especiales para el DBA en los DBMSs relacionales. El DBMS compila todas las definiciones de esquema y almacena sus descripciones en el catálogo del DBMS. Se utiliza un lenguaje de manipulación de datos (DML) para especificar las recuperaciones y las actualizaciones de la base de datos. Los DMLs pueden ser de alto nivel (orientados a conjunto, no procedimentales) o de bajo nivel (orientados a registros, procedimentales). Un DML de alto nivel se puede incrustar en un lenguaje *host*, y también se puede utilizar como un lenguaje independiente; en el último caso se denomina con frecuencia lenguaje de consulta.

Hemos explicado los diferentes tipos de interfaces proporcionadas por los DBMSs, así como los tipos de usuarios de DBMS con los que cada interfaz está asociada. Después, hemos explicado el entorno del sistema de bases de datos, los módulos software DBMS típicos y las utilidades DBMS destinadas a ayudar a los usuarios y al DBA en sus tareas. Continuamos con una panorámica de las arquitecturas de dos y tres capas para las aplicaciones de bases de datos, moviéndonos progresivamente hasta la arquitectura de *n* capas, mucho más común en la mayoría de las aplicaciones modernas, particularmente las basadas en la Web.

Por último, clasificamos los DBMS según varios criterios: modelo de datos, número de usuarios, número de sitios, tipos de rutas de acceso y en general. Hablamos de la disponibilidad de los DBMSs y de los módulos adicionales (desde la ausencia de coste en forma de software de código abierto, hasta las configuraciones con un coste anual de varios millones por mantenimiento). También nos referimos a la variedad de acuerdos de licencia para los DBMSs y los productos relacionados. La principal clasificación de los DBMSs está basada en los modelos de datos. Explicamos brevemente los principales modelos que se utilizan en los DBMSs comerciales actuales y ofrecimos un ejemplo de modelo de datos de red.

## Preguntas de repaso

**2.1.** Defina los siguientes términos: modelo de datos, esquema de base de datos, estado de la base de datos, esquema interno, esquema conceptual, esquema externo, independencia de datos, DDL, DML, SDL, VDL, lenguaje de consulta, lenguaje host, sublenguaje de datos, utilidad de base de datos, catálogo, arquitectura cliente/servidor, arquitectura de tres capas y arquitectura de *n* capas.

**2.2.** Explique las principales categorías de modelos de datos.

**2.3.** ¿Cuál es la diferencia entre un esquema de base de datos y un estado de base de datos?

**2.4.** Describa la arquitectura de tres esquemas. ¿Por qué son necesarios los mapeos entre los niveles de esquema? ¿Cómo soportan los diferentes lenguajes de definición de esquemas esta arquitectura?

**2.5.** ¿Cuál es la diferencia entre la independencia lógica de datos y la independencia física de datos? ¿Cuál es más difícil de conseguir? ¿Por qué?

**2.6.** ¿Cuál es la diferencia entre DMLs procedimentales y no procedimentales?

**2.7.** Explique los diferentes tipos de interfaces amigables con el usuario y los tipos de usuarios que normalmente utilizan cada una de ellas.

**2.8.** ¿Con qué otro software de computación interactúa un DBMS?

**2.9.** ¿Cuál es la diferencia entre las arquitecturas cliente/servidor de dos y tres capas?

**2.10.** Explique algunos tipos de utilidades y herramientas de bases de datos, así como sus funciones.

**2.11.** ¿Cuál es la funcionalidad adicional que se incorpora en la arquitectura de *n* capas ($n > 3$)?

## Ejercicios

**2.12.** Piense en diferentes usuarios para la base de datos de la Figura 1.2. ¿Qué tipos de aplicaciones necesitaría cada uno? ¿A qué categoría de usuario pertenecería cada uno y qué tipo de interfaz necesitaría cada uno de ellos?

**2.13.** Elija una aplicación de bases de datos con la que esté familiarizado. Diseñe un esquema y muestre una base de datos de ejemplo para esa aplicación, utilizando la notación de las Figuras 1.2 y 2.1. ¿Qué tipos de información adicional y de restricciones tendrá que representar en el esquema? Piense en los distintos usuarios de su base de datos y diseñe una vista para cada uno.

**2.14.** Si estuviera diseñando un sistema basado en la Web para realizar reservas en aerolíneas y vender billetes de avión, ¿qué arquitectura DBMS elegiría de las presentadas en la Sección 2.5? ¿Por qué? ¿Por qué las demás arquitecturas no serían una buena elección?

**2.15.** Considere la Figura 2.1. Además de las restricciones que relacionan los valores de las columnas de una tabla con las columnas de otra, también hay otras restricciones que restringen los valores de una columna o de una combinación de columnas de una tabla. Una de esas restricciones estipula que una columna o un grupo de columnas debe ser único a través de todas las filas de la tabla. Por ejemplo, en la tabla ESTUDIANTE, la columna NumEstudiante debe ser única (para evitar que dos estudiantes diferentes tengan el mismo NumEstudiante). Identifique la columna o el grupo de columnas de las otras tablas que deben ser únicos por todas las filas de la tabla.

## Bibliografía seleccionada

Muchos libros de bases de datos, incluyendo Date (2004), Silberschatz y otros (2005), Ramakrishnan and Gehrke (2003), García-Molina y otros (2000, 2002), y Abiteboul y otros (1995), ofrecen una explicación de los diferentes conceptos de bases de datos aquí explicados. Tsichritzis and Lochovsky (1982) es un libro antiguo dedicado a los modelos de datos. Tsichritzis and Klug (1978) y Jardine (1977) presentan la arquitectura de tres esquemas, que se sugirió por primera vez en el informe DBTG CODASYL (1971) y, posteriormente, en un informe del American National Standards Institute (ANSI) en 1975. En Codd (1992) se ofrece un aná-

lisis en profundidad del modelo de datos relacional y de algunas de sus posibles extensiones. El estándar propuesto para las bases de datos orientadas a objetos se describe en Cattell y otros (2000). En la Web hay muchos documentos que describen XML, como, por ejemplo, XML (2005).

ETI Extract Toolkit (*http://www.eti.com*) y la herramienta de administración de bases de datos, DB Artisan, de Embarcadero Technologies (*http://www.embarcadero.com*), son ejemplos de utilidades de bases de datos.

CAPÍTULO 3

# Modelado de datos con el modelo Entidad-Relación (ER)

El modelado conceptual es una fase muy importante para diseñar correctamente una aplicación de base de datos. Por lo general, el término **aplicación de base de datos** se refiere a una base de datos concreta y a los programas asociados encargados de implementar las consultas y las actualizaciones de la base de datos. Por ejemplo, una aplicación de base de datos BANCO que realiza el seguimiento de las cuentas de los clientes debe incluir programas encargados de implementar las actualizaciones de la base de datos correspondientes a los depósitos y las retiradas de fondos de los clientes. Estos programas ofrecen a los usuarios finales de la aplicación interfaces gráficas de usuario (GUIs) compuestas por formularios y menús. Por tanto, parte de la aplicación de base de datos requerirá el diseño, la implementación y la comprobación de esos programas de aplicación. Tradicionalmente, el diseño y la comprobación de los **programas de aplicación** se han considerado más como parte del dominio de la ingeniería de software que del dominio de las bases de datos. Como las metodologías de diseño de bases de datos incluyen cada vez más conceptos que sirven para especificar las operaciones con los objetos de bases de datos, y como las metodologías de ingeniería de software especifican la estructura de las bases de datos que los programas utilizarán y a las que accederán, es evidente que estas actividades están estrechamente relacionadas. En la Sección 3.8 explicaremos brevemente algunos de los conceptos que sirven para especificar las operaciones con las bases de datos, y de nuevo los veremos cuando expliquemos la metodología de diseño de una base de datos con las aplicaciones de ejemplo del Capítulo 12.

En este capítulo seguiremos la metodología tradicional de concentrarse en las estructuras y las restricciones de la base de datos durante el diseño de esta última. Presentaremos los conceptos de modelado del **modelo Entidad-Relación (ER)**, que es un modelo de datos conceptual de alto nivel. Este modelo y sus variaciones se utilizan con frecuencia para el diseño conceptual de las aplicaciones de base de datos, y muchas herramientas de diseño emplean estos conceptos. Describimos los conceptos básicos de la estructura de datos y las restricciones del modelo ER, así como su uso en el diseño de esquemas conceptuales para las aplicaciones de base de datos. También presentamos la notación esquemática asociada con el modelo ER, conocida como **diagramas ER**.

Las metodologías de modelado de objetos, como el **Lenguaje de modelado universal** (**UML**, **Universal *Modeling Language***), son cada vez más populares en el diseño y la ingeniería del software. Estas metodologías van más allá del diseño de bases de datos a fin de especificar un diseño específico de los módulos software y sus interacciones mediante varios tipos de diagramas. Una parte importante de estas metodologías

(antiguamente conocidas como *diagramas de clase*[1]) son parecidas a los diagramas ER. En los diagramas de clase se especifican *operaciones* sobre los objetos, además de especificar la estructura del esquema de la base de datos. Las operaciones se pueden utilizar para especificar los *requisitos funcionales* durante el diseño de la base de datos, como se explica en la Sección 3.1. La Sección 3.8 presenta algunos de los conceptos y notaciones UML para los diagramas de clase que están relacionados con el diseño de la base de datos, y los compara brevemente con la notación y los conceptos ER. En la Sección 4.6 y el Capítulo 12 se explican otros conceptos y notaciones UML.

Este capítulo está organizado de este modo: la Sección 3.1 explica el papel de los modelos de datos conceptuales de alto nivel en el diseño de las bases de datos. En la Sección 3.2 se exponen los requisitos de una aplicación de base de datos de ejemplo para ilustrar el uso de los conceptos del modelo ER. Esta base de datos de ejemplo también se utiliza en capítulos posteriores. En la Sección 3.3 se presentan los conceptos de entidades y atributos, y se va introduciendo gradualmente la técnica diagramática para visualizar un esquema ER. En la Sección 3.4 se introducen los conceptos de las relaciones binarias y sus roles y restricciones estructurales. La Sección 3.5 introduce los tipos de entidad débiles. La Sección 3.6 muestra cómo se refina el diseño de un esquema para incluir las relaciones. La Sección 3.7 repasa la notación de los diagramas ER, resume los problemas que pueden surgir en el diseño del esquema, y explica cómo elegir los nombres de las estructuras del esquema de la base de datos. La Sección 3.8 introduce algunos conceptos diagramáticos de clase UML, los compara con los conceptos de modelo ER y los aplica a la misma base de datos de ejemplo. La Sección 3.9 explica los tipos más complejos de relaciones, mientras que la Sección 3.10 resume el capítulo.

El material de las Secciones 3.8 y 3.9 se puede excluir de un curso introductorio; si desea una explicación más detallada de los conceptos de modelado de datos y del diseño de bases de datos conceptual, debe pasar de la Sección 3.7 al Capítulo 4, donde se describen las extensiones del modelo ER que conducen al modelo ER mejorado (EER), el cual incluye conceptos como la especialización, la generalización, la herencia y los tipos de unión (categorías). En el Capítulo 4 también se incluyen algunos conceptos UML y notaciones adicionales.

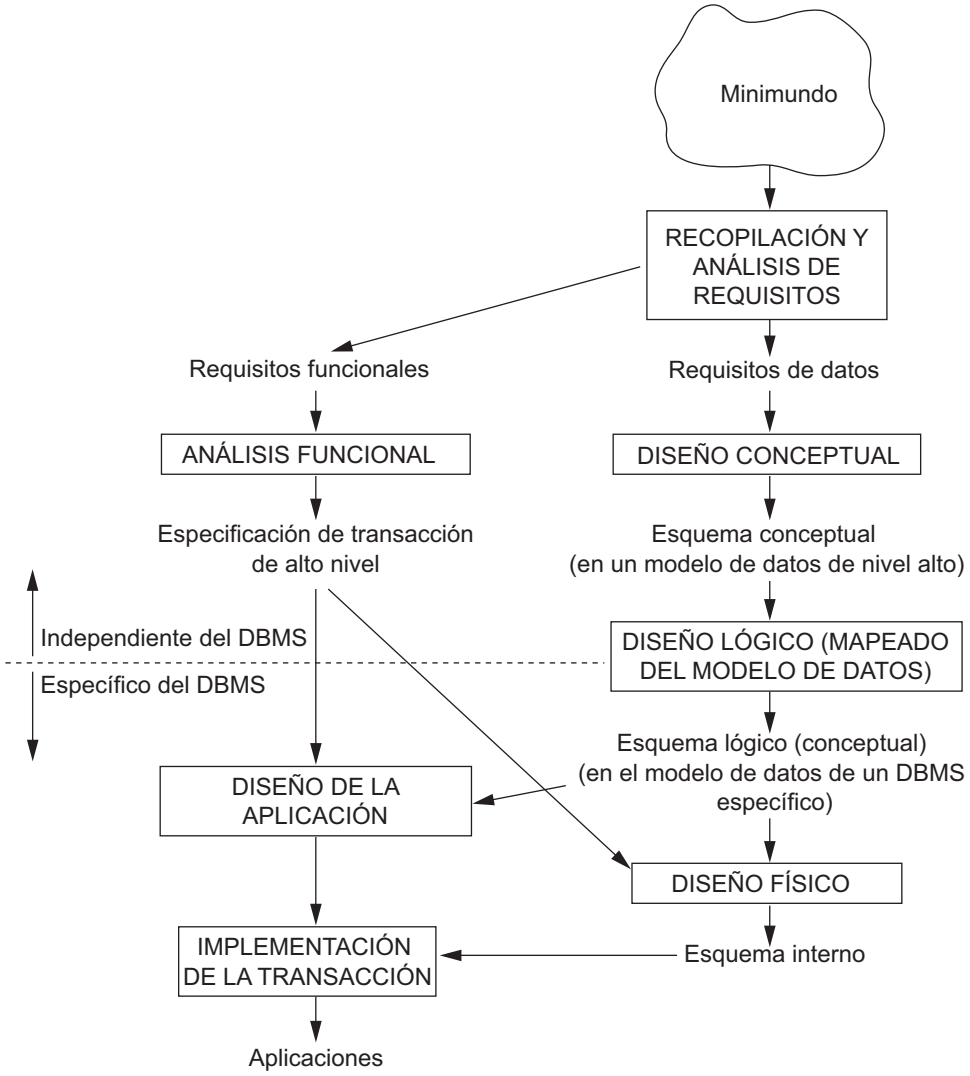
## 3.1 Uso de modelos de datos conceptuales de alto nivel para el diseño de bases de datos

La Figura 3.1 muestra una descripción simplificada del proceso de diseño de una base de datos. El primer paso es la **recopilación de requisitos y el análisis**. Durante este paso, los diseñadores de bases de datos entrevistan a los potenciales usuarios de la base de datos para comprender y documentar sus **requisitos en cuanto a datos**. El resultado de este paso es un conjunto por escrito de los requisitos del usuario. Estos requisitos se deben plasmar en un formulario lo más detallado y completo posible. En paralelo al estudio de estos requisitos, resulta útil especificar los **requisitos funcionales** de la aplicación, que consisten en las **operaciones** (o **transacciones**) definidas por el usuario que se aplicarán a la base de datos, incluyendo las recuperaciones y las actualizaciones. En el diseño de software, es frecuente utilizar los *diagramas de flujo de datos*, *diagramas de secuencia*, *escenarios* y otras técnicas para especificar los requisitos funcionales. No vamos a explicar ninguna de estas técnicas; normalmente se explican en profundidad en los textos de ingeniería de software. En el Capítulo12 ofreceremos una visión general de algunas de estas técnicas.

Una vez recopilados y analizados todos los requisitos, el siguiente paso es crear un **esquema conceptual** para la base de datos, mediante un modelo de datos conceptual de alto nivel. Este paso se denomina **diseño conceptual**. El esquema conceptual es una descripción concisa de los requisitos de datos por parte de los usuarios e incluye descripciones detalladas de los tipos de entidades, relaciones y restricciones; se expresan utilizando los conceptos proporcionados por el modelo de datos de alto nivel. Como estos conceptos no incluyen detalles de implementación, normalmente son más fáciles de entender y se pueden utilizar para comuni-

[1] Una **clase** se parece en muchos casos a un tipo de entidad.

**Figura 3.1.** Diagrama simplificado para ilustrar las fases principales del diseño de una base de datos.




car con usuarios no técnicos. El esquema conceptual de alto nivel también se puede utilizar como referencia para garantizar que se han reunido todos los requisitos de datos de los usuarios y que esos requisitos no entran en conflicto. Esta metodología permite a los diseñadores de bases de datos concentrarse en especificar las propiedades de los datos, sin tener que preocuparse por los detalles del almacenamiento. En consecuencia, es más fácil para ellos crear un buen diseño conceptual de bases de datos.

Durante o después del diseño del esquema conceptual, se pueden utilizar las operaciones básicas del modelo de datos para especificar las operaciones de usuario de alto nivel identificadas durante el análisis funcional. Esto también sirve para confirmar que el esquema conceptual satisface todos los requisitos funcionales identificados. Es posible realizar modificaciones en el esquema conceptual si con el esquema inicial no se pueden especificar algunos de los requisitos funcionales.

El siguiente paso del diseño de una base de datos es la implementación real de la misma mediante un DBMS comercial. La mayoría de los DBMSs comerciales actuales utilizan un modelo de datos de implementación

(como el modelo de base de datos relacional u objeto-relación), de modo que el esquema conceptual se transforma de modelo de datos de alto nivel en modelo de datos de implementación. Este paso se conoce como **diseño lógico** o **asignación de modelo de datos**; su resultado es un esquema de base de datos en el modelo de datos de implementación del DBMS.

El último paso es la fase de **diseño físico**, durante la cual se especifican las estructuras de almacenamiento interno, los índices, las rutas de acceso y la organización de los archivos para la base de datos. En paralelo a estas actividades, se diseñan e implementan los programas de aplicación como transacciones de bases de datos correspondientes a las especificaciones de transacción de alto nivel. En el Capítulo 12 explicaremos más en profundidad el proceso de diseño de una base de datos.

En este capítulo sólo presentamos los conceptos básicos del modelo ER para el diseño del esquema conceptual. Los conceptos adicionales sobre modelado se explican en el Capítulo 4, donde se introduce el modelo EER.

## 3.2 Un ejemplo de aplicación de base de datos

En esta sección se describe un ejemplo de aplicación de base de datos, denominada EMPRESA, que sirve para ilustrar los conceptos del modelo ER básico y su uso en el diseño del esquema. En primer lugar se enumeran los requisitos de datos para la base de datos, y después se crea su esquema conceptual paso a paso tras introducir los conceptos de modelado del modelo ER. La base de datos EMPRESA sirve como seguimiento de los empleados, los departamentos y los proyectos de una empresa. Suponga que después de la fase de recopilación de requisitos y análisis, los diseñadores de la base de datos proporcionan la siguiente descripción del *minimundo* (la parte de la empresa que se va a representar en la base de datos):

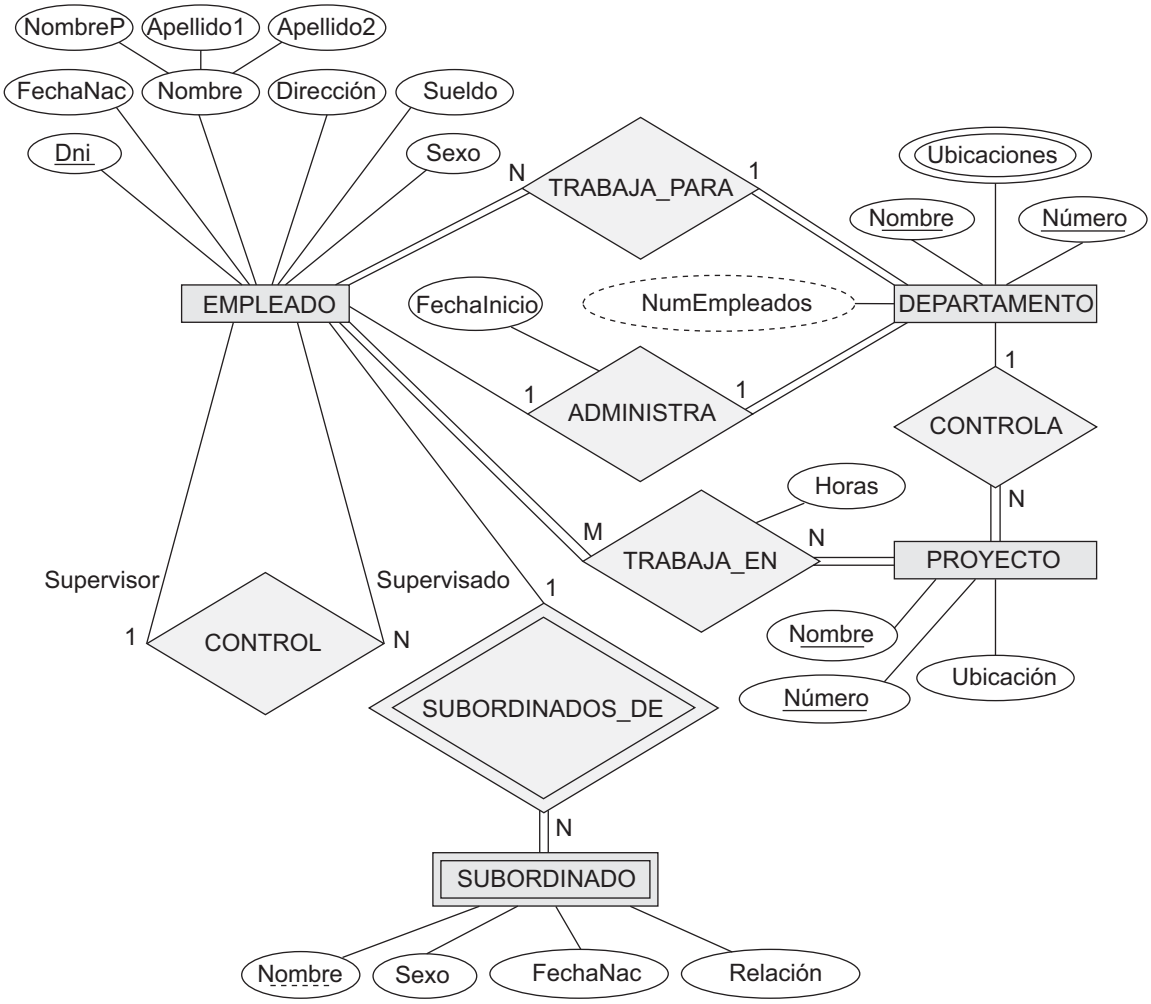
- La empresa está organizada en departamentos. Cada uno tiene un nombre único, un número único y un empleado concreto que lo administra. Se realizará un seguimiento de la fecha en que ese empleado empezó a administrar el departamento. Un departamento puede tener varias ubicaciones.
- Un departamento controla una cierta cantidad de proyectos, cada uno de los cuales tiene un nombre único, un número único y una sola ubicación.
- Almacenaremos el nombre, el documento nacional de identidad,[2] la dirección, el sueldo, el sexo y la fecha de nacimiento de cada empleado. Un empleado está asignado a un departamento, pero puede trabajar en varios proyectos, que no están controlados necesariamente por el mismo departamento. Se hará un seguimiento del número de horas por semana que un empleado trabaja en cada proyecto. También se realizará el seguimiento del supervisor directo de cada empleado.
- También se desea realizar un seguimiento de las personas a cargo de cada empleado por el tema de los seguros. Por cada persona a cargo o subordinado, se registrará su nombre de pila, sexo, fecha de nacimiento y relación con el empleado.

La Figura 3.2 muestra cómo se puede visualizar el esquema de esta aplicación de base de datos mediante la notación gráfica conocida como **diagramas ER**. Esta figura se explicará gradualmente a medida que se vayan presentando los conceptos del modelo ER. Describiremos el proceso por pasos para deducir este esquema a partir de los requisitos indicados (y de la explicación de la notación diagramática ER) a medida que vayamos introduciendo los conceptos del modelo ER.

---

[2] En Estados Unidos se utiliza el número de la seguridad social, que es un identificador de nueve dígitos único asignado a cada persona, para hacer un seguimiento de su empleo, sus beneficios y sus impuestos. En el resto de países hay esquemas de identificación parecidos, como, por ejemplo, el número del DNI (Documento Nacional de Identidad) español.

**Figura 3.2.** Diagrama de un esquema ER para la base de datos EMPRESA. La notación diagramática se introduce gradualmente a lo largo de este capítulo.




# 3.3 Tipos de entidad, conjuntos de entidades, atributos y claves

El modelo ER describe los datos como entidades, relaciones y atributos. En la Sección 3.3.1 introducimos los conceptos de entidades y sus atributos. En la Sección 3.3.2 explicamos los tipos de entidad y los atributos clave. Después, ya en la Sección 3.3.3, concretamos el diseño conceptual inicial de los tipos de entidad para la base de datos EMPRESA. Las relaciones se describen en la Sección 3.4.

## 3.3.1 Entidades y atributos

**Entidades y sus atributos.** El objeto básico representado por el modelo ER es una **entidad**, que es una *cosa* del mundo real con una existencia independiente. Una entidad puede ser un objeto con una existencia física (por ejemplo, una persona en particular, un coche, una casa o un empleado) o puede ser un objeto con

una existencia conceptual (por ejemplo, una empresa, un trabajo o un curso universitario). Cada entidad tiene **atributos** (propiedades particulares que la describen). Por ejemplo, una entidad EMPLEADO se puede describir mediante el nombre, la edad, la dirección, el sueldo y el trabajo que desempeña. Una entidad en particular tendrá un valor para cada uno de sus atributos. Los valores de los atributos que describen cada entidad se convierten en la parte principal de los datos almacenados en la base de datos.

La Figura 3.3 muestra dos entidades y los valores de sus atributos. La entidad EMPLEADO *e*1 tiene cuatro atributos: Nombre, Dirección, Edad y TlfCasa; sus valores son 'José Pérez', 'Ribera del Sena, 915. Getafe, Madrid 28903', '55' y '91-123-4567', respectivamente. La entidad EMPRESA *c*1 tiene tres atributos: Nombre, SedeCentral y Presidente; sus valores son 'Sunco Oil', 'Madrid' y 'José Pérez', respectivamente.

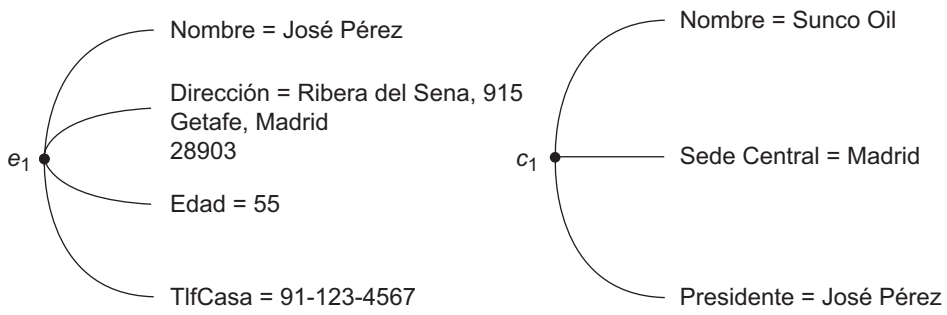
En el modelo ER se dan varios tipos de atributos: *simple* frente a *compuesto*, *monovalor* frente a *multivalor*, y *almacenado* frente a *derivado*. En primer lugar, definimos estos tipos de atributos e ilustramos su uso mediante ejemplos. Después, introducimos el concepto de *valor NULL* (nulo) para un atributo.

**Atributos compuestos frente a atributos simples (atómicos).** Los **atributos compuestos** se pueden dividir en subpartes más pequeñas, que representan atributos más básicos con significados independientes. Por ejemplo, el atributo Dirección de la entidad EMPLEADO de la Figura 3.3 se puede subdividir en DirCalle, Ciudad, Provincia y CP,[3] con los valores 'Ribera del Sena, 915', 'Getafe', 'Madrid' y '28903'. Los atributos que no son divisibles se denominan **atributos simples** o **atómicos**. Los atributos compuestos pueden formar una jerarquía. Por ejemplo, DirCalle se puede subdividir en tres atributos simples: Número, Calle y NumApto, como se muestra en la Figura 3.4. El valor de un atributo compuesto es la concatenación de los valores de sus atributos simples.

Los atributos son útiles para modelar situaciones en las que un usuario se refiere a veces al atributo compuesto como una unidad, pero otras veces se refiere específicamente a sus componentes. Si se hace referencia al atributo compuesto como un todo, no hay necesidad de subdividirlo en atributos componentes. Por ejemplo, si no hay necesidad de referirse a los componentes individuales de una dirección (código postal, calle, etcétera), entonces la dirección entera se puede designar como un atributo simple.

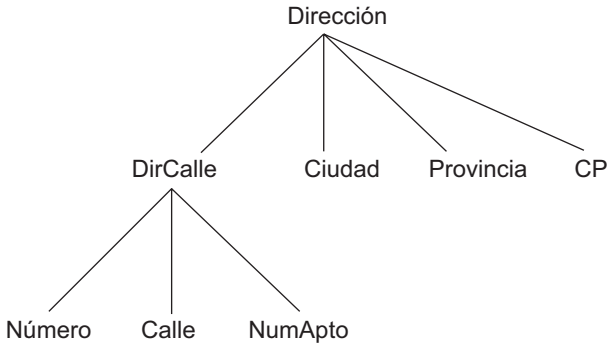
**Atributos monovalor y multivalor.** La mayoría de los atributos tienen un solo valor para una entidad en particular; dichos atributos reciben el nombre de **monovalor** o **de un solo valor**. Por ejemplo, Edad es un atributo monovalor de una persona. En algunos casos, un atributo puede tener un conjunto de valores para la misma entidad (por ejemplo, un atributo Colores para un coche, o un atributo Licenciaturas para una persona). Los coches con un solo color tiene un solo valor, mientras que los coches de dos tonos tienen dos valores de color. De forma parecida, puede que una persona no tenga ninguna licenciatura, otra puede que tenga una, y una tercera persona puede que tenga dos o más; por consiguiente, diferentes personas pueden

**Figura 3.3.** Dos entidades, EMPLEADO *e*1 y EMPRESA *c*1, y sus atributos.




[3] CP es la abreviatura que se utiliza en España para el código postal de cinco dígitos.

**Figura 3.4.** Una jerarquía de atributos compuestos.




tener una *cantidad de valores* diferente para el atributo Licenciaturas. Dichos atributos se denominan **multivalor**. Un atributo multivalor puede tener límites superior e inferior para restringir el número de valores permitidos para cada entidad individual. Por ejemplo, el atributo Colores de un coche puede tener entre uno y tres valores, si asumimos que un coche puede tener tres colores a lo sumo.

**Atributos almacenados y derivados.** En algunos casos, dos (o más) valores de atributo están relacionados (por ejemplo, los atributos Edad y FechaNac de una persona). Para una entidad de persona en particular, el valor de Edad puede determinarse a partir de la fecha actual (el día de hoy) y el valor de FechaNac de esa persona. El atributo Edad se denomina entonces **atributo derivado** y se dice que se ha **derivado** del atributo FechaNac, que es el denominado **atributo almacenado**. Algunos valores de atributo se pueden derivar de *entidades relacionadas*; por ejemplo, un atributo NumEmpleados de una entidad DEPARTAMENTO puede derivarse contando el número de empleados relacionados con (o que trabajan para) ese departamento.

**Valores NULL (nulos).** En algunos casos, es posible que una entidad en particular no tenga un valor aplicable para un atributo. Por ejemplo, el atributo NumApto de una dirección sólo se aplica a las direcciones correspondientes a edificios de apartamentos, y no a otros tipos de residencias, como las casas unifamiliares. De forma parecida, un atributo Licenciaturas sólo se aplica a las personas con carrera universitaria. Para estas situaciones se ha creado un valor especial denominado NULL (nulo). La dicción de una casa unifamiliar tendría el valor NULL para su atributo NumApto, y una persona sin carrera universitaria tendría NULL para Licenciaturas. NULL también se puede utilizar cuando no se conoce el valor de un atributo para una entidad en particular (por ejemplo, si no conocemos el número de teléfono de la casa de 'José Pérez' en la Figura 3.3). El significado del tipo anterior de NULL no es aplicable, mientras que el significado del último es *desconocido*. La categoría *desconocido* se puede clasificar en dos casos. El primero se da cuando se sabe que existe el valor del atributo pero *no se encuentra*: por ejemplo, si el atributo Altura de una persona aparece como NULL. El segundo caso se da cuando es *no conocido* si existe el valor del atributo: por ejemplo, si el atributo TlfCasa de una persona es NULL.

**Atributos complejos.** Los atributos compuestos y multivalor se pueden anidar arbitrariamente. Podemos representar el anidamiento arbitrario agrupando componentes de un atributo compuesto entre paréntesis () y separando los componentes con comas, y mostrando los atributos multivalor entre llaves {}. Dichos atributos se denominan **atributos complejos**. Por ejemplo, si una persona puede tener más de una residencia y cada residencia puede tener una sola dirección y varios teléfonos, el atributo TlfDir de una persona se puede especificar como en la Figura 3.5.[4] Los dos atributos, Tlf y Dir, son compuestos.

[4] Los que están familiarizados con XML verán que los atributos complejos son parecidos a los elementos complejos de XML (consulte el Capítulo 27).

**Figura 3.5.** Un atributo complejo: TlfDir.

{TlfDir({Tlf(CodÁrea,NumTlf)},Dir(DirCalle(Número,Calle,NumApto),Ciudad,Provincia,CP))}

### 3.3.2 Tipos de entidades, conjuntos de entidades, claves y conjuntos de valores

**Tipos de entidades y conjuntos de entidades.** Una base de datos normalmente contiene grupos de entidades que son parecidas. Por ejemplo, una compañía que da trabajo a cientos de empleados puede querer almacenar información parecida relacionada con cada uno de ellos. Estas entidades de empleado comparten los mismos atributos, pero cada entidad tiene su(s) *propio(s) valor(es)* para cada atributo. Un **tipo de entidad** define una *colección* (o *conjunto*) de entidades que tienen los mismos atributos. La Figura 3.6 muestra dos tipos de entidades: EMPLEADO y EMPRESA, y una lista de atributos de cada una. También se ilustran unas cuantas entidades individuales de cada tipo, junto con los valores de sus atributos. La colección de todas las entidades de un tipo de entidad en particular de la base de datos en cualquier momento del tiempo se denomina conjunto de entidades; al conjunto de entidades normalmente se hace referencia utilizando el mismo nombre que para el tipo de entidad. Por ejemplo, EMPLEADO se refiere tanto al *tipo de entidad* como al conjunto actual de *todas las entidades de empleado* de la base de datos.

Un tipo de entidad se representa en los diagramas ER[5] (*véase* la Figura 3.2) como un rectángulo con el nombre del tipo de entidad en su interior. Los nombres de los atributos se encierran en óvalos y están unidos a su tipo de entidad mediante líneas rectas. Los atributos compuestos están unidos a sus atributos componente mediante líneas rectas. Los atributos multivalor se muestran en óvalos dobles. La Figura 3.7(a) muestra un tipo de entidad COCHE en esta notación.

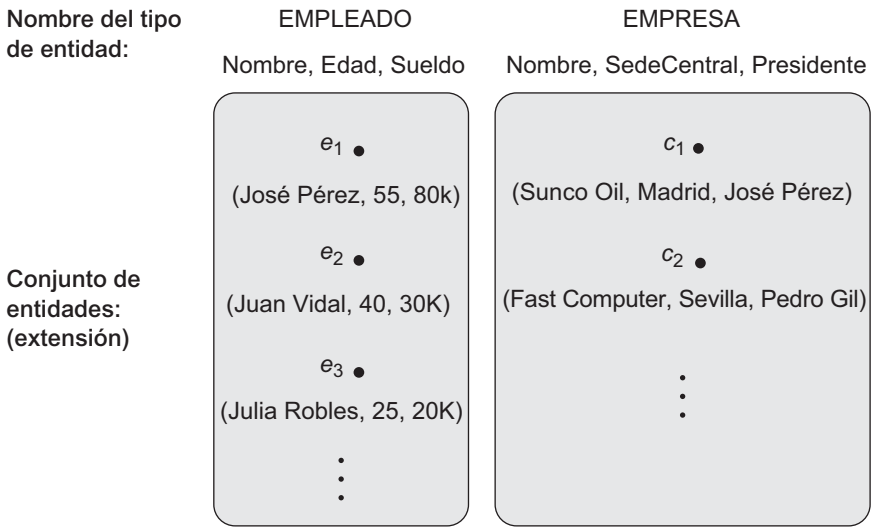
Un tipo de entidad describe el **esquema** o la **intención** de un *conjunto de entidades* que comparten la misma estructura. La colección de entidades de un tipo de entidad en particular está agrupada en un conjunto de entidades, que también se denomina **extensión** del tipo de entidad.

**Atributos clave de un tipo de entidad.** Una restricción importante de las entidades de un tipo de entidad es la **clave** o **restricción de unicidad** de los atributos. Un tipo de entidad normalmente tiene un atributo cuyos valores son distintos para cada entidad individual del conjunto de entidades. Dicho atributo se denomina **atributo clave**, y sus valores se pueden utilizar para identificar cada entidad sin lugar a dudas. Por ejemplo, en la Figura 3.6 el atributo Nombre es una clave del tipo de entidad EMPRESA porque no está permitido que dos empresas tengan el mismo nombre. Para el tipo de entidad PERSONA, un atributo clave típico es DNI. En ocasiones, una clave está formada por varios atributos juntos, lo que da a entender que la *combinación* de los valores de atributo debe ser distinta para cada entidad. Si un conjunto de atributos posee esta propiedad, la forma correcta de representar esto en el modelo ER que aquí describimos es definir un *atributo compuesto* y designarlo como atributo clave del tipo de entidad. Una clave compuesta debe ser mínima; es decir, en el atributo compuesto se deben incluir todos los atributos componente para tener una propiedad de unicidad. En una clave no se deben incluir atributos superfluos. En la notación diagramática ER, cada atributo clave tiene su nombre **subrayado** dentro del óvalo, como muestra la Figura 3.7(a).

Especificar que un atributo es una clave de un tipo de entidad significa que debe mantenerse la propiedad de unicidad anterior para *cada conjunto de entidades* del tipo de entidad. Por tanto, es una restricción que prohíbe que dos entidades tengan el mismo valor para el atributo clave al mismo tiempo. No es la propiedad de una extensión en particular; en cambio, es una restricción de *todas las extensiones* del tipo de entidad. Esta restricción clave (y otras restricciones que veremos más tarde) se derivan de las restricciones del minimundo representado por la base de datos.

5. Para los diagramas ER utilizamos una notación cercana a la notación original (Chen 1976). Se utilizan muchas otras notaciones; cuando hablemos de los diagramas de clase UML en este capítulo ofreceremos algunas de ellas, así como en el Apéndice A.

**Figura 3.6.** Dos tipos de entidades, EMPLEADO y EMPRESA, y algunas entidades miembro de cada una de ellas.




Algunos tipos de entidad tienen *más de un atributo clave*. Por ejemplo, cada uno de los atributos IDVehículo y Matrícula del tipo de entidad COCHE (*véase* la Figura 3.7) es una clave por propio derecho. El atributo Matrícula es un ejemplo de clave compuesta formada en España por dos atributos componente simples, un Número y unas Letras, ninguno de los cuales es una clave por sí mismo. Un tipo de entidad también puede *carecer de clave*, en cuyo caso se denomina *tipo de entidad débil* (consulte la Sección 3.5).

**Conjuntos de valores (dominios) de atributos.** Cada atributo simple de un tipo de entidad está asociado con un **conjunto de valor** (o **dominio** de valores), que especifica el conjunto de los valores que se pueden asignar a ese atributo por cada entidad individual. En la Figura 3.6, si el rango de edades permitido para los empleados está entre 16 y 70, podemos especificar el conjunto de valores del atributo Edad de EMPLEADO como un conjunto de números enteros entre 16 y 70. De forma parecida, podemos especificar el conjunto de valores para el atributo Nombre como un conjunto de cadenas de caracteres alfabéticos separados por espacios en blanco, etcétera. Los conjuntos de valores no se muestran en los diagramas ER; normalmente se especifican mediante los **tipos de datos** básicos disponibles en la mayoría de los lenguajes de programación, como entero, cadena, booleano, flotante, tipo enumerado, subrango, etcétera. También se emplean otros tipos de datos adicionales para representar la fecha, la hora y otros conceptos.

Matemáticamente, un atributo $A$ de un tipo de entidad $E$ cuyo conjunto de valores es $V$ se puede definir como una **función** de $E$ al conjunto potencia[6] $P(V)$ de $V$:

$$A : E \rightarrow P(V)$$

Al valor del atributo $A$ para la entidad $e$ nos referimos como $A(e)$. La definición anterior abarca tanto los atributos de un solo valor como los multivalor, así como los nulos. Un valor NULL queda representado por el *conjunto vacío*. Para los atributos de un solo valor, $A(e)$ está restringido a ser un conjunto sencillo para cada entidad $e$ de $E$, mientras no haya una restricción en los atributos multivalor.[7] Para un atributo compuesto $A$, el conjunto de valores $V$ es el producto cartesiano de $P(V_1)$, $P(V_2)$, . . ., $P(V_n)$, donde $V_1$, $V_2$, . . ., $Vn$ son los conjuntos de valores de los atributos simples que forman $A$:

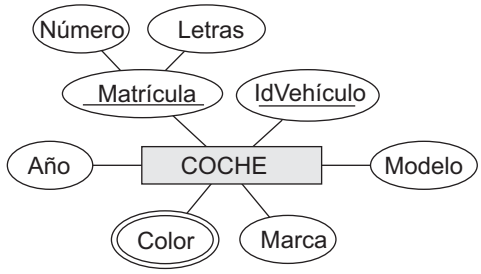
$$V = P(V_1) \times P(V_2) \times ... \times P(V_n)$$

6. El **conjunto potencia** $P(V)$ de un conjunto $V$ es el conjunto de todos los subconjuntos de $V$.

7. Un conjunto **sencillo** es un conjunto con un solo elemento (valor).

**Figura 3.7.** El tipo de entidad COCHE con dos atributos clave, Matrícula e IDVehículo. (a) Notación de diagrama ER. (b) Conjunto de entidades con tres entidades.

**(a)**




**(b)**

COCHE
Matrícula (Número, Letras), IdVehículo, Marca, Modelo, Año, {Color}

$COCHE_1$
((1234, ABC), TK629, Ford Mustang, convertible, 2004 {rojo, negro})

$COCHE_2$
((9876, CBA), WP9872, Nissan Máxima, 4 puertas, 2005, {azul})

$COCHE_3$
((7200, ABC), TD729, Chrysler LeBaron, 4 puertas, 2002, {blanco, azul})

. . .

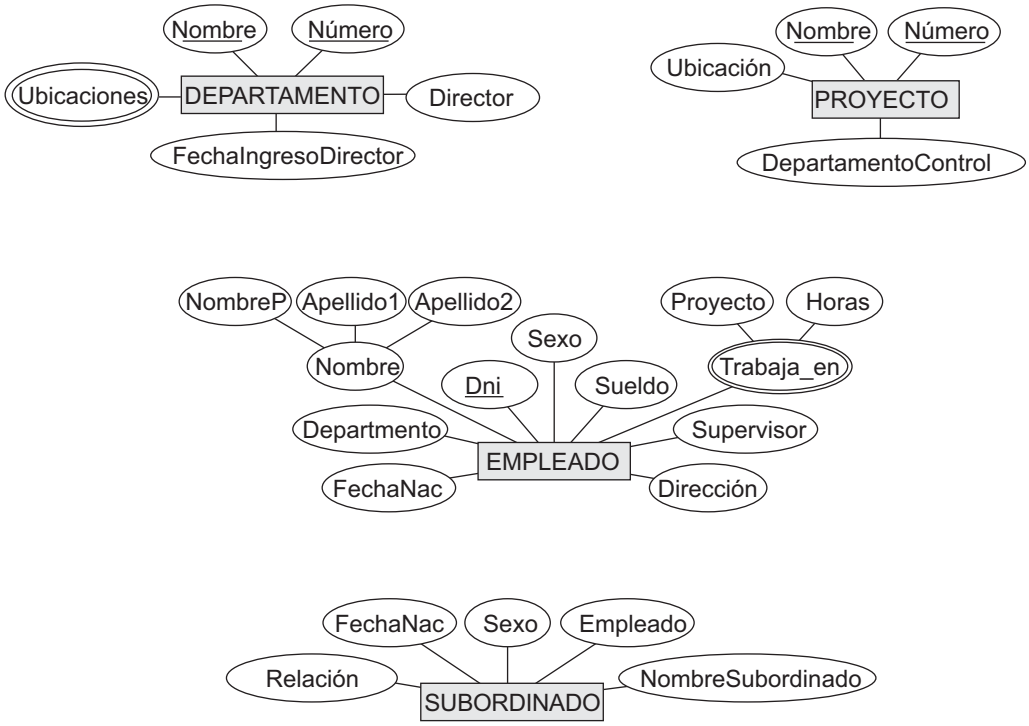
El conjunto de valores proporciona todos los valores posibles. Normalmente, en la base de datos sólo existen unos cuantos de estos valores. Estos valores representan los datos del estado del minimundo. Corresponden a los datos tal y como existen en el minimundo.

### 3.3.3 Diseño conceptual inicial de la base de datos EMPRESA

Ahora podemos definir los tipos de entidad para la base de datos EMPRESA, en base a los requisitos descritos en la Sección 3.2. Después de definir aquí varios tipos de entidad y sus atributos, refinaremos nuestro diseño en la Sección 3.4 después de introducir el concepto de una relación. De acuerdo con los requisitos enumerados en la Sección 3.2, podemos identificar cuatro tipos de entidades (uno por cada uno de los cuatro elementos de la especificación [véase la Figura 3.8]):

1. Tipo de entidad DEPARTAMENTO con los atributos NombreDpto, NúmeroDpto, Ubicaciones, Director y FechaIngresoDirector. Ubicaciones es el único atributo multivalor. Podemos especificar que Nombre y NúmeroDpto son atributos clave (separados) porque cada uno se especificó como único.
2. Tipo de entidad PROYECTO con los atributos Nombre, Número, Ubicación y DepartamentoControl. Tanto Nombre como Número son atributos clave (separados).
3. Tipo de entidad EMPLEADO con los atributos Nombre, Dni, Sexo, Dirección, Sueldo, FechaNac, Departamento y Supervisor. Nombre y Dirección pueden ser atributos compuestos; no obstante, esto no se especificó en los requisitos. Debemos volver a los usuarios para ver si alguno de ellos se referirá a los componentes individuales de Nombre (NombrePila, PrimerApellido, SegundoApellido) o de Dirección.

**Figura 3.8.** Diseño preliminar de los tipos de entidad para la base de datos EMPRESA. Algunos de los atributos mostrados se redefinirán como relaciones.




4. Tipo de entidad SUBORDINADO con los atributos Empleado, NombreSubordinado, Sexo, FechaNac y Relación (con el empleado).

Hasta ahora, no hemos representado el hecho de que un empleado pueda trabajar en varios proyectos, ni tampoco el número de horas por semana trabajadas por un empleado en cada proyecto. Esta característica se muestra como parte del tercer requisito en la Sección 3.2, y se puede representar mediante un atributo compuesto multivalor de EMPLEADO denominado TrabajaEn con los componentes Proyecto y Horas. De forma alternativa, puede representarse como un atributo compuesto multivalor de PROYECTO denominado Trabajadores con los componentes Empleado y Horas. En la Figura 3.8 elegimos la primera alternativa, que muestra cada uno de los tipos de entidad que acabamos de describir. El atributo Nombre de EMPLEADO se muestra como un atributo compuesto, probablemente después de consultar con los usuarios.

## 3.4 Tipos de relaciones, conjuntos de relaciones, roles y restricciones estructurales

En la Figura 3.8 hay varias *relaciones implícitas* entre los distintos tipos de entidades. De hecho, en cuanto un atributo de un tipo de entidad se refiere a otro tipo de entidad, decimos que existen algunas relaciones. Por ejemplo, el atributo Director de DEPARTAMENTO se refiere a un empleado que dirige el departamento; el atributo DepartamentoControl de PROYECTO se refiere al departamento que controla el proyecto; el atributo Supervisor de EMPLEADO se refiere a otro empleado (el que supervisa a este empleado); el atributo

Departamento de EMPLEADO se refiere al departamento para el que trabaja el empleado; etcétera. En el modelo ER, estas referencias no deben representarse como atributos, sino como **relaciones**, que explicaremos en esta sección. El esquema de la base de datos EMPRESA se refinará en la Sección 3.6 para representar explícitamente las relaciones. En el diseño inicial de los tipos de entidades, las relaciones se capturan normalmente en forma de atributos. Al depurar el diseño, estos atributos se convierten en relaciones entre los tipos de entidades.

Esta sección está organizada de este modo: la Sección 3.4.1 introduce los conceptos de tipos de relaciones, conjuntos de relaciones e instancias (también conocidas como ejemplares u ocurrencias) de relaciones. Además, en la Sección 3.4.2 definimos los conceptos de grado de relación, nombres de rol y relaciones recursivas. Después, ya en la Sección 3.4.3, explicamos las restricciones estructurales en las relaciones (por ejemplo, como las razones de cardinalidad y las dependencias existentes). La Sección 3.4.4 muestra cómo los tipos de relaciones también pueden tener atributos.

### 3.4.1 Tipos, conjuntos e instancias de relaciones

Un **tipo de relación** $R$ entre $n$ tipos de entidades $E_1, E_2, \ldots, E_n$ define un conjunto de asociaciones (o un **conjunto de relaciones**) entre las entidades de esos tipos de entidades. Como en el caso de los tipos de entidades y los conjuntos de entidades, normalmente se hace referencia a un tipo de relación y su correspondiente conjunto de relaciones con el *mismo nombre*, $R$. Matemáticamente, el conjunto de relaciones $R$ es un conjunto de instancias de relación $r_i$, donde cada $r_i$ asocia $n$ entidades individuales $(e_1, e_2, \ldots, e_n)$, y cada entidad $e_j$ de $r_i$ es un miembro del tipo de entidad $E_j$, $1 \leq j \leq n$. Por tanto, un tipo de relación es una relación matemática en $E_1, E_2, \ldots, E_n$; de forma alternativa, se puede definir como un subconjunto del producto cartesiano $E_1 \times E_2, \times \ldots \times E_n$. Se dice que cada uno de los tipos de entidad $E_1, E_2, \ldots, E_n$ participa en el tipo de relación $R$; de forma parecida, cada una de las entidades individuales $e_1, e_2, \ldots, e_n$ se dice que **participa** en la instancia de relación $r_i = (e_1, e_2, \ldots, e_n)$.

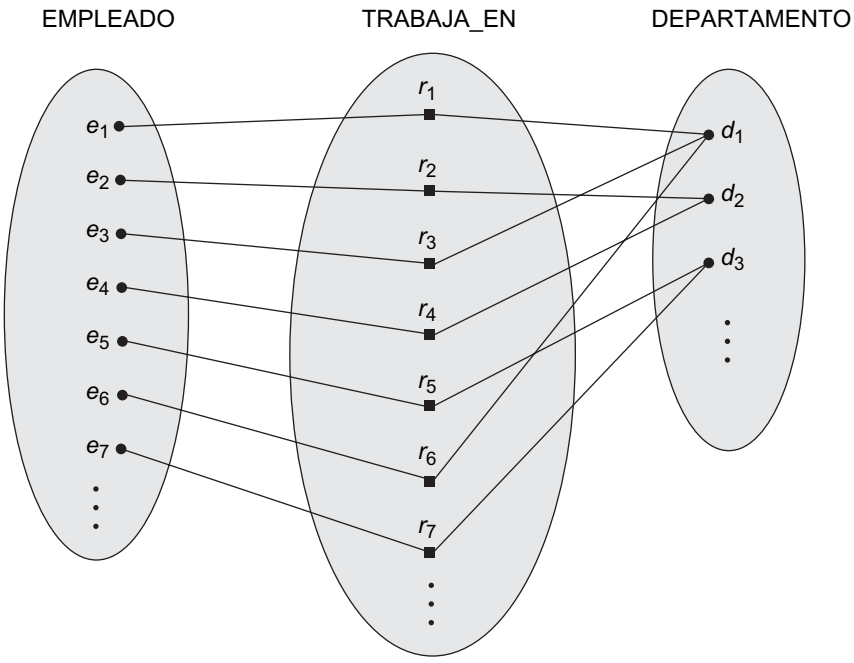
Informalmente, cada instancia de relación $r_i$ en $R$ es una asociación de entidades, donde la asociación incluye exactamente una entidad de cada tipo de entidad participante. Cada una de dichas instancias de relación $r_i$ representa el hecho de que las entidades que participan en $r_i$ están relacionadas de alguna forma en la situación correspondiente del minimundo. Por ejemplo, considere un tipo de relación TRABAJA_PARA asociado con una entidad EMPLEADO y una entidad DEPARTAMENTO. La Figura 3.9 ilustra este ejemplo, donde cada instancia de relación $r_i$ se muestra conectada a las entidades EMPLEADO y DEPARTAMENTO que participan en $r_i$. En el minimundo representado por la Figura 3.9, los empleados $e_1$, $e_3$ y $e_6$ trabajan para el departamento $d_1$; los empleados $e_2$ y $e_4$ trabajan para el departamento $d_2$; y los empleados $e_5$ y $e_7$ trabajan para el departamento $d_3$.

En los diagramas ER, los tipos de relaciones se muestran mediante rombos, conectados a su vez mediante líneas a los rectángulos que representan los tipos de entidad participantes. El nombre de la relación se muestra en el rombo (*véase* la Figura 3.2).

### 3.4.2 Grado de relación, nombres de rol y relaciones recursivas

**Grado de un tipo de relación.** El **grado** de un tipo de relación es el número de tipos de entidades participantes. Por tanto, la relación TRABAJA_PARA es de grado dos. Un tipo de relación de grado dos se denomina **binario**, y uno de grado tres, **ternario**. Un ejemplo de relación ternaria es SUMINISTRO, en la Figura 3.10, donde cada instancia de relación $r_i$ asocia tres entidades (un proveedor $s$, un repuesto $p$ y un proyecto $j$), siempre que $s$ suministre un repuesto $p$ al proyecto $j$. Las relaciones pueden ser generalmente de cualquier grado, pero las más comunes son las relaciones binarias. Las relaciones de grado más alto son, por lo general, más complejas que las relaciones binarias; las explicaremos más tarde, en la Sección 3.9.

**Figura 3.9.** Algunas instancias del conjunto de relación TRABAJA_PARA, que representa un tipo de relación TRABAJA_PARA entre EMPLEADO y DEPARTAMENTO.




**Relaciones y atributos.** A veces es conveniente imaginar un tipo de relación en términos de atributos, como explicamos en la Sección 3.3.3. Considere el tipo de relación TRABAJA_PARA de la Figura 3.9. Uno puede pensar en un atributo denominado Departamento del tipo de entidad EMPLEADO donde el valor de Departamento por cada entidad EMPLEADO es (una referencia a) la entidad DEPARTAMENTO para la que ese empleado trabaja. Por tanto, el conjunto de valores para este atributo Departamento es el conjunto de *todas* las entidades DEPARTAMENTO, que es el conjunto de entidades DEPARTAMENTO. Es lo que hacíamos en la Figura 3.8 cuando especificábamos el diseño inicial del tipo de entidad EMPLEADO para la base de datos EMPRESA. No obstante, cuando pensamos en una relación binaria como si fuera un atributo, siempre tenemos dos opciones. En este ejemplo, la alternativa es pensar en un atributo Empleado multivalor del tipo de entidad DEPARTAMENTO cuyos valores por cada entidad DEPARTAMENTO es el conjunto de entidades EMPLEADO que trabajan para ese departamento. El conjunto de valores de este atributo Empleado es el conjunto potencia del conjunto de entidades EMPLEADO. Cualquiera de estos dos atributos (Departamento de EMPLEADO o Empleado de DEPARTAMENTO) puede representar el tipo de relación TRABAJA_PARA. Si se representan ambas, se restringen para ser mutuamente inversas.[8]

**Nombres de rol y relaciones recursivas.** Cada tipo de entidad que participa en un tipo de relación juega un papel o rol particular en la relación. El **nombre de rol** hace referencia al papel que una entidad participante del tipo de entidad juega en cada instancia de relación, y ayuda a explicar el significado de la relación. Por ejemplo, en el tipo de relación TRABAJA_PARA, EMPLEADO juega el papel de *empleado* o *trabajador* y DEPARTAMENTO juega el papel de *departamento* o *empleador*.

Los nombres de rol no son técnicamente necesarios en los tipos de relación donde todos los tipos de entidad participantes son distintos, puesto que cada nombre de tipo de entidad participante se puede utilizar como

[8] Este concepto de representar los tipos de relación como atributos se utiliza en una clase de modelos de datos denominada **modelos de datos funcionales**. En las bases de datos de objetos (consulte el Capítulo 20), las relaciones se pueden representar mediante atributos de referencia, en una dirección o en ambas direcciones. En las bases de datos relacionales (consulte el Capítulo 5), las claves extrañas son un tipo de atributo de referencia que se utiliza para representar las relaciones.

**Figura 3.10.** Algunas instancias de relación en el conjunto de relaciones ternarias SUMINISTRO.

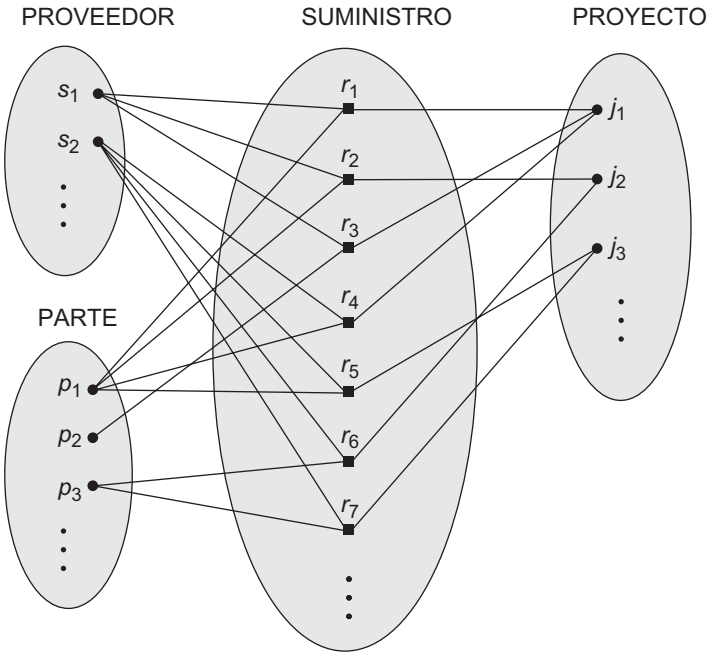
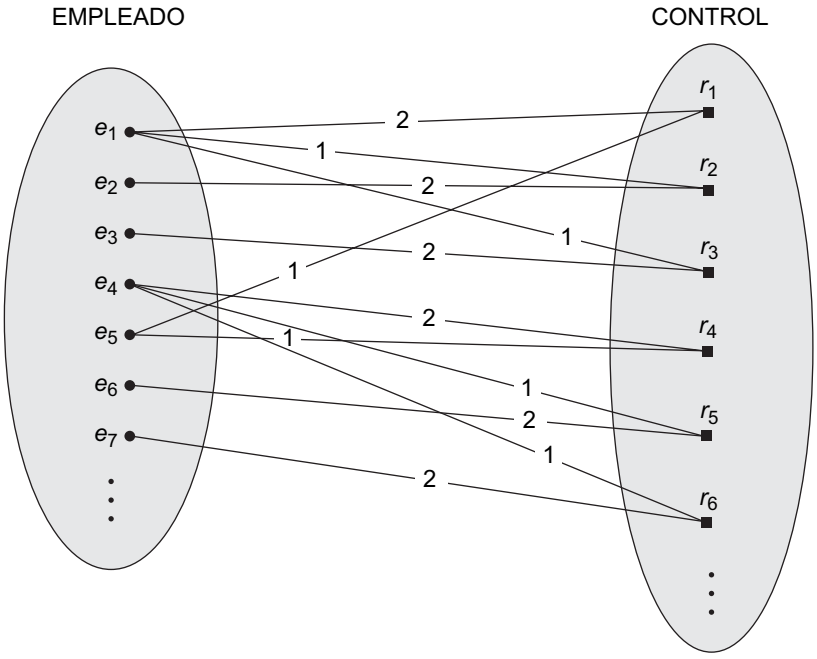



**Figura 3.11.** Una relación recursiva CONTROL entre EMPLEADO en el papel de *supervisor* (1) y EMPLEADO en el papel de *subordinado* (2).




nombre de rol. No obstante, en algunos casos el *mismo* tipo de entidad participa más de una vez en un tipo de relación con *diferentes roles*. En esos casos, el nombre de rol es esencial para distinguir el significado de cada

participación. Dichos tipos de relaciones se denominan **relaciones recursivas**. La Figura 3.11 muestra un ejemplo. El tipo de relación CONTROL relaciona un empleado con un supervisor, donde las entidades empleado y supervisor son miembros del mismo tipo de entidad EMPLEADO. Por tanto, el tipo de entidad EMPLEADO *participa* dos veces en CONTROL: una en el papel de *supervisor* (o *jefe*), y otra en el papel de *supervisado* (o *subordinado*). Cada instancia de relación $r_i$ en CONTROL asocial dos entidades de empleado, $e_j$ y $e_k$, una de las cuales desempeña el papel de supervisor y la otra el papel de supervisado.

En la Figura 3.11, las líneas marcadas como '1' representan el papel de supervisor, y las marcadas con el '2' representan el papel de supervisado; por tanto, $e_1$ supervisa a $e_2$ y $e_3$; $e_4$ supervisa a $e_6$ y $e_7$; y $e_5$ supervisa a $e_1$ y $e_4$. En este ejemplo, cada instancia de relación debe tener dos líneas, una marcada con el '1' (supervisión) y otra con el '2' (supervisado).

## 3.4.3 Restricciones en los tipos de relaciones

Los tipos de relaciones normalmente tienen ciertas restricciones que limitan las posibles combinaciones entre las entidades que pueden participar en el conjunto de relaciones correspondiente. Estas restricciones están determinadas por la situación del minimundo representado por las relaciones. Por ejemplo, en la Figura 3.9, si la empresa tiene por norma que cada empleado debe trabajar únicamente para un departamento, entonces tendríamos que describir esta restricción en el esquema. Podemos distinguir dos tipos principales de restricciones de relación: *razón de cardinalidad* y *participación*.

**Razones de cardinalidad para las relaciones binarias.** La **razón de cardinalidad** de una relación binaria especifica el número *máximo* de instancias de relación en las que una entidad puede participar. Por ejemplo, en el tipo de relación binaria TRABAJA_PARA, DEPARTAMENTO:EMPLEADO tiene una razón de cardinalidad de 1:N, que significa que cada departamento puede estar relacionado con (es decir, emplea a) cualquier cantidad de empleados,[9] pero un empleado puede estar relacionado con (trabajar para) un solo departamento. Las posibles razones de cardinalidad para los tipos de relación binaria son 1:1, 1:N, N:1 y M:N.

Un ejemplo de relación binaria 1:1 es ADMINISTRA (véase la Figura 3.12), que relaciona una entidad departamento con el empleado que dirige ese departamento. Esto representa las restricciones del minimundo, según las cuales, en cualquier momento del tiempo, un empleado puede dirigir un solo departamento y un departamento sólo puede tener un director. El tipo de relación TRABAJA_EN (véase la Figura 3.13) tiene una razón de cardinalidad de M:N, porque la norma del minimundo es que un empleado puede trabajar en varios proyectos, y un proyecto puede tener varios empleados. Las razones de cardinalidad de las relaciones binarias se representan en los diagramas ER mediante 1, M y N en los rombos (véase la Figura 3.2).

**Restricciones de participación y dependencias de existencia.** La **restricción de participación** especifica si la existencia de una entidad depende de si está relacionada con otra entidad a través de un tipo de relación. Esta restricción especifica el número *mínimo* de instancias de relación en las que puede participar cada entidad, y en ocasiones recibe el nombre de **restricción de cardinalidad mínima**. Hay dos tipos de restricciones de participación, total y parcial, que ilustramos con un ejemplo. Si una política de la empresa dice que *cada* empleado debe trabajar para un departamento, entonces una entidad de empleado sólo puede existir si participa en al menos una instancia de relación TRABAJA_PARA (véase la Figura 3.9). De este modo, la participación de EMPLEADO en TRABAJA_PARA se denomina **participación total**, es decir, cada entidad del *conjunto total* de entidades empleado debe estar relacionada con una entidad departamento a través de TRABAJA_PARA. La participación total también se conoce como **dependencia de existencia**. En la Figura 3.12 no esperamos que cada empleado dirija un departamento, de modo que la participación de EMPLEADO en el tipo de relación ADMINISTRA es parcial; esto significa que *algo* o *parte del conjunto* de entidades empleado está relacionado con alguna entidad departamento a través de ADMINISTRA, pero no necesariamente con todas. Nos referiremos a la razón de cardinalidad y a las restricciones de participación, en conjunto, como **restricciones estructurales** de un tipo de relación.

[9] N significa cualquier número de entidades relacionadas (cero o más).

**Figura 3.12.** Una relación 1:1, ADMINISTRA.

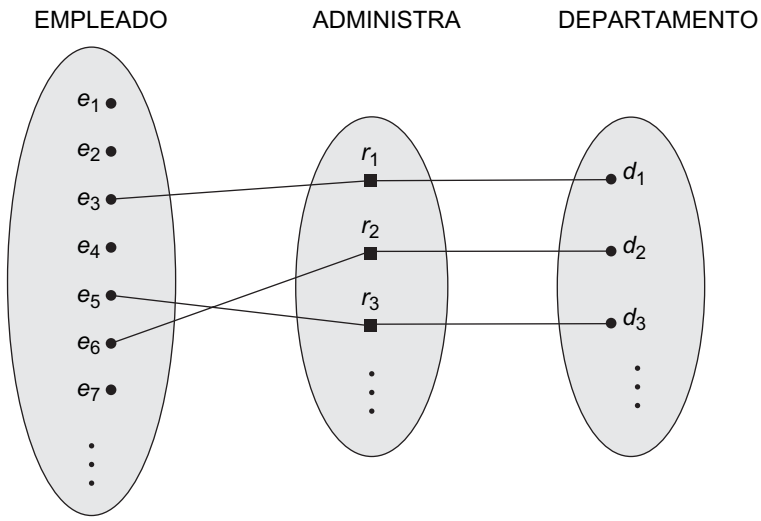
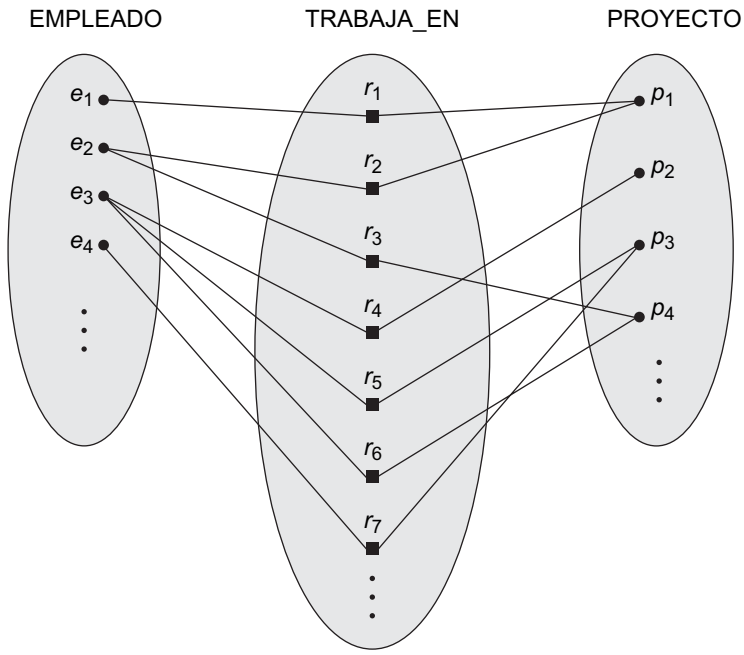



**Figura 3.13.** Una relación M:N, TRABAJA_EN.




En los diagramas ER, la participación total (o dependencia existente) se muestra como una *línea doble* que conecta el tipo de entidad participante con la relación, mientras que las participaciones parciales se representan mediante una *línea sencilla* (véase la Figura 3.2).

## 3.4.4 Atributos de los tipos de relación

Los tipos de relación también pueden tener atributos, parecidos a los de los tipos de entidad. Por ejemplo, para registrar el número de horas por semana que un empleado trabaja en un proyecto en particular, podemos

incluir un atributo Horas para el tipo de relación TRABAJA_EN de la Figura 3.13. Otro ejemplo es incluir la fecha en que el director empezó a dirigir un departamento, mediante un atributo FechaInicio para el tipo de relación ADMINISTRA de la Figura 3.1.

Los atributos de los tipos de relación 1:1 o 1:N se pueden trasladar a uno de los tipos de entidad participantes. Por ejemplo, el atributo FechaInicio para la relación ADMINISTRA puede ser un atributo de EMPLEADO o DEPARTAMENTO, aunque conceptualmente pertenece a ADMINISTRA. Esto se debe a que ADMINISTRA es una relación 1:1, por lo que cada entidad departamento o empleado participa *a lo sumo en una* instancia de relación. Por tanto, el valor del atributo FechaInicio se puede determinar por separado, bien mediante la entidad departamento participante, bien mediante la entidad empleado (director) participante.

En el caso de un tipo de relación 1:N, un atributo de relación *sólo* se puede migrar al tipo de entidad que se encuentra en el lado N de la relación. En la Figura 3.9, por ejemplo, si la relación TRABAJA_PARA también tiene un atributo FechaInicio que indica la fecha en que un empleado empezó a trabajar para un departamento, este atributo se puede incluir como un atributo de EMPLEADO. Esto se debe a que cada empleado trabaja sólo para un departamento y, por tanto, participa en un máximo de una instancia de relación en TRABAJA_PARA. En los tipos de relación 1:1 y 1:N, la decisión sobre dónde debe colocarse un atributo de relación (como un atributo de tipo de relación o como un atributo de un tipo de entidad participante) la determina subjetivamente el diseñador del esquema.

Para los tipos de relación M:N, algunos atributos pueden determinarse mediante la *combinación de entidades participantes* en una instancia de relación, no mediante una sola relación. Dichos atributos *deben especificarse como atributos de relación*. Un ejemplo de esto es el atributo Horas de la relación M:N TRABAJA_EN (véase la Figura 3.13); el número de horas que un empleado trabaja en un proyecto viene determinado por una combinación empleado-proyecto, y no separadamente por cualquiera de estas entidades.

## 3.5 Tipos de entidades débiles

Los tipos de entidad que no tienen atributos clave propios se denominan **tipos de entidad débiles**. En contraposición, los **tipos de entidad regulares** que tienen un atributo clave (que incluye todos los ejemplos que hemos explicado hasta ahora) se denominan **tipos de entidad fuertes**. Las entidades que pertenecen a un tipo de entidad débil se identifican como relacionadas con entidades específicas de otro tipo de entidad en combinación con uno de sus valores de atributo. Podemos llamar a este otro tipo de entidad **tipo de entidad identificado** o **propietario**[10], y al tipo de relación que relaciona un tipo de entidad débil con su propietario lo podemos llamar **relación identificativa** del tipo de entidad débil.[11] Un tipo de entidad débil siempre tiene una *restricción de participación total* (dependencia de existencia) respecto a su relación identificativa, porque una entidad débil no puede identificarse sin una entidad propietaria. No obstante, no toda dependencia de existencia produce un tipo de entidad débil. Por ejemplo, la entidad PERMISO_CONDUCIR no puede existir a menos que esté relacionada con una entidad PERSONA, aunque tiene su propia clave (NumPermiso) y, por tanto, no es una entidad débil.

Considere el tipo de entidad SUBORDINADO, relacionada con EMPLEADO, que sirve para llevar el control de las personas a cargo de cada empleado a través de una relación 1:N (*véase* la Figura 3.2). Los atributos de SUBORDINADO son Nombre (el nombre de pila del dependiente), FechaNac, Sexo y Relación (con el empleador). Dos personas a cargo de *dos empleados diferentes* pueden, por casualidad, tener los mismos valores para Nombre, FechaNac, Sexo y Relación, pero todavía seguirán siendo entidades distintas. Se identifican como entidades diferentes únicamente después de determinar la *entidad de empleado en particular* con la que

[10] El tipo de entidad identificativa también se denomina a veces **tipo de entidad padre** o **tipo de entidad dominante**.

[11] El tipo de entidad débil también recibe a veces el nombre de **tipo de entidad hija** o **tipo de entidad subordinada**.

cada persona a cargo está relacionada. Cada entidad empleado posee entidades dependientes que están relacionadas con ella.

Un tipo de entidad débil normalmente tiene una **clave parcial**, que es el conjunto de atributos que pueden identificar sin lugar a dudas las entidades débiles que están *relacionadas con la misma entidad propietaria*.[12] En nuestro ejemplo, si asumimos que no puede haber dos dependientes del mismo empleado con el mismo nombre, el atributo Nombre de SUBORDINADO es la clave parcial. En el peor de los casos, la clave parcial será un atributo compuesto por todos los *atributos de la entidad débil*.

En los diagramas ER, tanto el tipo de la entidad débil como la relación identificativa, se distinguen rodeando sus cuadros y rombos mediante unas líneas dobles (véase la Figura 3.2). El atributo de clave parcial aparece subrayado con una línea discontinua o punteada.

Los tipos de entidades débiles se puede representar a veces como atributos complejos (compuestos, multivalor). En el ejemplo anterior, podríamos especificar un atributo multivalor Subordinados para EMPLEADO, que es un atributo compuesto por los atributos simples Nombre, FechaNac, Sexo y Relación. El diseñador de la base de datos toma la decisión del tipo de representación que hay que usar. Uno de los criterios que puede utilizar es elegir la representación del tipo de entidad débil si hay muchos atributos. Si la entidad débil participa independientemente en los tipos de relación de otra forma que su tipo de relación identificativa, entonces no debe modelarse como un atributo complejo.

En general, se puede definir cualquier cantidad de niveles de tipos de entidad débil; un tipo de entidad propietaria puede ser ella misma un tipo de entidad débil. Además, un tipo de entidad débil puede tener más de un tipo de entidad identificativa y un tipo de relación identificativa de grado superior a dos, como se ilustra en la Sección 3.9.

## 3.6 Perfeccionamiento del diseño ER para la base de datos EMPRESA

Ahora podemos refinar el diseño de la base de datos EMPRESA de la Figura 3.8 convirtiendo los atributos que representan relaciones en tipos de relaciones. La razón de cardinalidad y la restricción de participación de cada tipo de relación vienen determinadas por los requisitos enumerados en la Sección 3.2. Si no es posible determinar alguna razón de cardinalidad o dependencia a partir de los requisitos, habrá que consultar con los usuarios para determinar esas restricciones estructurales.

En nuestro ejemplo, especificaremos los siguientes tipos de relaciones:

- ADMINISTRA, un tipo de relación 1:1 entre EMPLEADO y DEPARTAMENTO. La participación de EMPLEADO es parcial, pero la de DEPARTAMENTO no queda clara a partir de los requisitos. Consultamos con los usuarios, que nos dicen que un departamento siempre debe tener un gerente, lo que implica una participación total.[13] El atributo FechaInicio se asigna a este tipo de relación.
- TRABAJA_PARA, un tipo de relación 1:N entre DEPARTAMENTO y EMPLEADO. Ambas participaciones son totales.
- CONTROLA, un tipo de relación 1:N entre DEPARTAMENTO y PROYECTO. La participación de PROYECTO es total, mientras que la de DEPARTAMENTO se ha determinado como parcial, después de haber consultado con los usuarios, que indicaron que es posible que algunos departamentos no controlen proyecto alguno.

[12] La clave parcial a veces recibe el nombre de **discriminador**.

[13] Las reglas del minimundo que determinan las restricciones se denominan a veces reglas empresariales, pues están determinadas por la *empresa* u organización que utilizará la base de datos.

- CONTROL, un tipo de relación 1:N entre EMPLEADO (en el papel de supervisor) y EMPLEADO (en el papel de supervisado). Como los usuarios han indicado que no todo empleado es un supervisor y no todo empleado tiene un supervisor, se determina que las dos participaciones son parciales.
- TRABAJA_EN que, después de que los usuarios hayan indicado que en un proyecto pueden trabajar varios empleados, se determina que es un tipo de relación M:N con el atributo Horas. Se determina que ambas participaciones son totales
- SUBORDINADOS_DE, un tipo de relación 1:N entre EMPLEADO y SUBORDINADO, que también es la relación identificativa del tipo de entidad débil SUBORDINADO. La participación de EMPLEADO es parcial, en tanto que la de SUBORDINADO es total.

Después de especificar los seis tipos de relación anteriores, eliminamos de los tipos de entidad de la Figura 3.8 todos los atributos que se han convertido en relaciones, entre los que se encuentran Director y FechaIngresoDirector de DEPARTAMENTO; DepartamentoControl de PROYECTO; Departamento, Supervisor y Trabaja_en de EMPLEADO; y Empleado de SUBORDINADO. Es importante tener el mínimo de redundancia posible cuando diseñemos el esquema conceptual de una base de datos. Si es deseable algo de redundancia a nivel de almacenamiento o a nivel de la vista de usuario, se puede introducir más tarde, como explicamos en la Sección 1.6.1.

# 3.7 Diagramas ER, convenciones de denominación y problemas de diseño

## 3.7.1 Resumen de la notación para los diagramas ER

Las Figuras 3.9 a 3.13 ilustran ejemplos de la participación de los tipos de entidad en los tipos de relación mediante sus extensiones: las instancias de entidad individuales y las instancias de relación en los conjuntos de entidades y los conjuntos de relaciones. En los diagramas ER se hace hincapié en la representación de los esquemas, más que de las instancias. Esto es más útil en el diseño de bases de datos porque el esquema de una base de datos rara vez cambia, mientras que el contenido de los conjuntos de entidades cambia con frecuencia. Además, normalmente es más fácil visualizar el esquema que la extensión de una base de datos, porque es mucho más pequeño.

La Figura 3.2 muestra el **esquema ER de la base de datos** EMPRESA como un **diagrama ER**. Vamos a repasar la notación completa de un diagrama ER. Los tipos de entidad como EMPLEADO, DEPARTAMENTO y PROYECTO aparecen en rectángulos. Los tipos de entidad como TRABAJA_PARA, ADMINISTRA, CONTROLA y TRABAJA_EN se muestran en rombos conectados a los tipos de entidades participantes mediante líneas rectas.

Los atributos se muestran en óvalos, y cada uno está conectado a su tipo de entidad o tipo de relación mediante una línea recta. Los atributos que componen un atributo compuesto se conectan con el óvalo que representa el atributo compuesto, como se muestra con el atributo Nombre de EMPLEADO. Los atributos multivalor se muestran con óvalos dobles, como el atributo Ubicaciones de DEPARTAMENTO. Los nombres de los atributos clave aparecen subrayados. Los atributos derivados se muestran con óvalos de línea punteada, como el atributo NumEmpleados de DEPARTAMENTO.

Los tipos de entidades débiles se distinguen porque se colocan en rectángulos de borde doble y porque las relaciones que los identifican aparecen en rombos dobles, como se ilustra con la entidad SUBORDINADO y el tipo de relación identificativa SUBORDINADOS_DE. La clave parcial del tipo de entidad débil se subraya con una línea punteada.

En la Figura 3.2 la razón de cardinalidad de cada tipo de relación *binaria* se especifica adjuntando 1, M o N a cada borde participante. La razón de cardinalidad de DEPARTAMENTO:EMPLEADO en ADMINISTRA es

1:1, mientras que es 1:N para DEPARTAMENTO:EMPLEADO en TRABAJA_PARA, y M:N para TRABAJA_EN. La restricción de participación se especifica mediante una línea sencilla para la participación parcial y con líneas dobles para la participación total (dependencia de existencia).

En la Figura 3.2 mostramos los nombres del papel para el tipo de relación CONTROL porque el tipo de entidad EMPLEADO puede desempeñar los dos papeles en esa relación. La cardinalidad es 1:N de supervisor a supervisado porque cada empleado en el papel de supervisado tiene como máximo un supervisor directo, mientras que un empleado en el papel de supervisor puede supervisar a ninguno o más empleados.

La Figura 3.14 resume las convenciones de los diagramas ER.

### 3.7.2 Asignación correcta de nombre a las construcciones del esquema

Al diseñar el esquema de una base de datos, la elección de nombre para los tipos de entidad, atributos, tipos de relaciones y (especialmente) los papeles desempeñados no siempre es directo. Hay que elegir nombres que transmitan, lo mejor posible, los significados de las distintas estructuras del esquema. Hemos optado por *nombres en singular* para los tipos de entidad, en lugar de nombres plurales, porque el nombre de un tipo de entidad se aplica a cada entidad individual que pertenece a ese tipo de entidad. En nuestros diagramas ER utilizaremos la convención de que los nombres de los tipos de entidades y de los tipos de relación se escriben en mayúsculas, los nombres de los atributos se escriben con la primera letra en mayúscula, y los nombres de los papeles en minúsculas. En la Figura 3.2 hemos utilizado esta convención.

Como práctica general, dada una descripción narrativa de los requisitos de la base de datos, los *nombres* que aparecen en la narrativa tienden a ser nombres de tipos de entidades, y los *verbos* tienden a indicar nombres de tipos de relación. Los nombres de los atributos normalmente proceden de los nombres originales que describen los nombres correspondientes a los tipos de entidades.

Otra consideración en cuanto a la denominación implica la elección de nombres de relación binaria para que el diagrama ER del esquema se pueda leer de izquierda a derecha y de arriba abajo. Por regla general, hemos seguido esta norma en la Figura 3.2. Para completar la explicación de esta convención, tenemos que hacer una excepción a lo mostrado en la Figura 3.2: el tipo de relación SUBORDINADOS_DE se lee de abajo hacia arriba. Al describir esta relación, podemos decir que las entidades SUBORDINADO (tipo de entidad inferior) son SUBORDINADOS_DE (nombre de relación) un EMPLEADO (tipo de entidad superior). Para que esto se pueda leer de arriba hacia abajo, tenemos que renombrar el tipo de relación a TIENE_SUBORDINADOS, que se puede leer del siguiente modo: una entidad EMPLEADO (tipo de entidad superior) TIENE_SUBORDINADOS (nombre de relación) del tipo SUBORDINADO (tipo de entidad inferior). Este asunto surge porque cada relación binaria puede describirse como que empieza en cualquiera de los dos tipos de entidad participantes, como se explica al principio de la Sección 3.4.

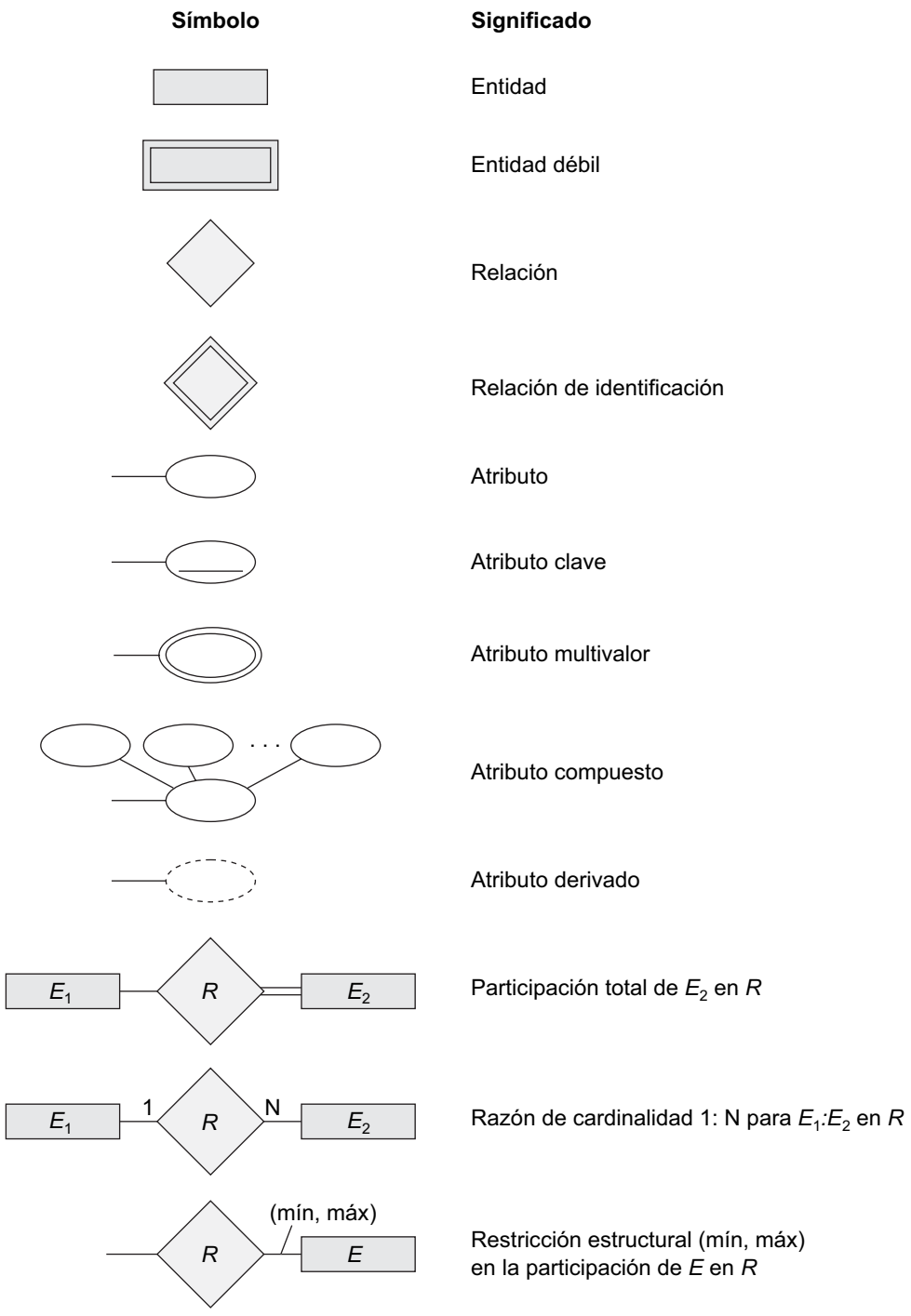
### 3.7.3 Opciones de diseño para el diseño conceptual ER

A veces puede resultar complejo decidir si un concepto en particular del minimundo debe modelarse como un tipo de entidad, un atributo o un tipo de relación. En esta sección ofrecemos algunos consejos breves sobre la construcción que debe elegirse en situaciones particulares.

En general, el proceso de diseño del esquema debe considerarse como un proceso de refinamiento iterativo: primero se crea un diseño inicial y después se va refinando paulatinamente hasta alcanzar el diseño más adecuado. Algunos de los refinamientos que a menudo se utilizan son los siguientes:

- Un concepto se puede interpretar primero como un atributo y, después, acabar como una relación porque se haya determinado que el atributo es una referencia a otro tipo de entidad. A menudo sucede que un par de estos atributos son inversos entre sí y se refinan como una relación binaria. Explicamos en profundidad este tipo de refinamiento en la Sección 3.6.

**Figura 3.14.** Resumen de la notación para los diagramas ER.




- De forma parecida, un atributo que existe en varios tipos de entidad puede elevarse o promocionarse a un tipo de entidad independiente. Por ejemplo, suponga que varios tipos de entidades de la base de datos UNIVERSIDAD, como ESTUDIANTE, PROFESOR y CURSO, disponen en el diseño inicial de

un atributo Departamento; el diseñador puede optar entonces por crear un tipo de entidad DEPARTAMENTO con un solo atributo NombreDpto y relacionarlo con los tres tipos de entidad (ESTUDIANTE, PROFESOR y CURSO) a través de las relaciones apropiadas. Más tarde pueden descubrirse otros atributos/relaciones de DEPARTAMENTO.

- Es posible aplicar un refinamiento inverso al caso anterior: por ejemplo, si en el diseño inicial existe un tipo de entidad DEPARTAMENTO con un solo atributo NombreDpto que está únicamente relacionado con otro tipo de entidad, ESTUDIANTE. En este caso, DEPARTAMENTO se puede reducir o degradar a un atributo de ESTUDIANTE.
- La Sección 3.9 explica las opciones concernientes al grado de una relación. En el Capítulo 4 explicamos otros refinamientos relacionados con la especialización/generalización. El Capítulo 12 explica los refinamientos *top-down* (descendente) y *bottom-up* (ascendente) adicionales que son comunes en el diseño del esquema conceptual.

### 3.7.4 Notaciones alternativas para los diagramas ER

Hay muchas notaciones diagramáticas alternativas para la visualización de los diagramas ER. El Apéndice A ofrece algunas de las notaciones más populares. En la Sección 3.8 introducimos la notación UML (Lenguaje de modelado universal, *Universal Modeling Language*) para los diagramas de clase, que se han propuesto como estándar para el modelado conceptual de objetos.

En esta sección describimos una notación ER alternativa para especificar las restricciones estructurales en las relaciones. Esta notación implica asociar un par de números enteros (mín, máx) a cada participación de un tipo de entidad $E$ en un tipo de relación $R$, donde $0 \leqslant$ mín $\leqslant$ máx y máx $\geqslant 1$. Los números significan que para cada entidad $e$ de $E$, $e$ debe participar en al menos mín y a lo sumo en máx instancias de relación de $R$ *en cualquier momento*. En este método, mín = 0 significa una participación parcial, mientras que mín > 0 implica una participación total.

La Figura 3.15 muestra el esquema de la base de datos EMPRESA utilizando la notación (mín, máx).[14] Normalmente, se utiliza la notación de cardinalidad razón/línea-sencilla/línea-doble o la notación (mín, máx). Esta última es más precisa, y la podemos utilizar para especificar las restricciones estructurales de los tipos de relación de *cualquier grado*. Sin embargo, no es suficiente para especificar algunas restricciones clave o relaciones de grado superior, como explicamos en la Sección 3.9.

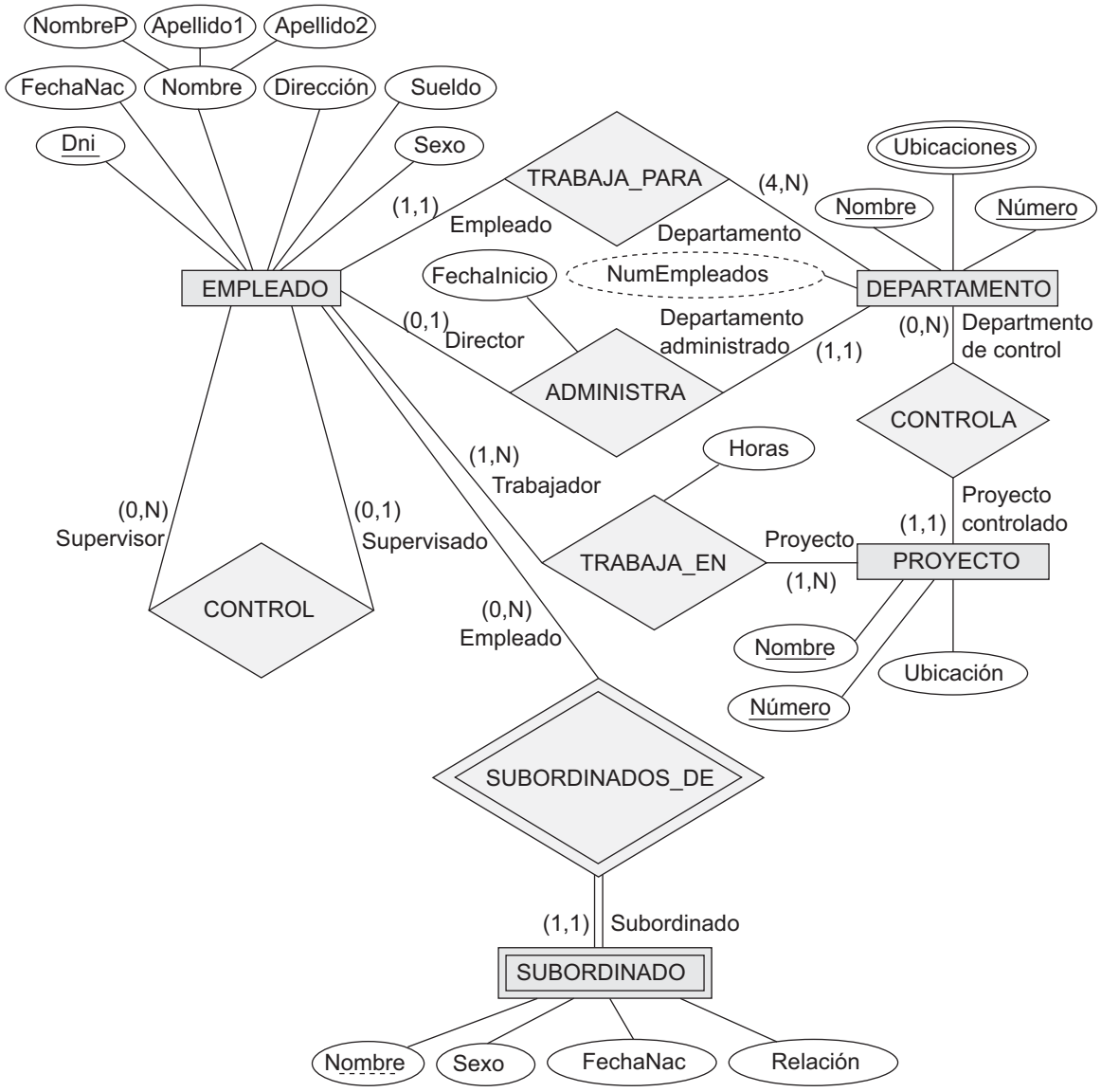
La Figura 3.15 también muestra los nombres de todos los papeles para el esquema de la base de datos EMPRESA.

## 3.8 Ejemplo de otra notación: diagramas de clase UML

La metodología UML se está utilizando extensamente en el diseño de software y tiene muchos tipos de diagramas para los distintos fines del diseño de software. Sólo explicaremos brevemente los fundamentos de los **diagramas de clase UML** y los compararemos con los diagramas ER. En algunos casos, los diagramas de clase se pueden considerar como una notación alternativa a los diagramas ER. La notación UML adicional y sus conceptos se presentan en la Sección 4.6 y el Capítulo 12. La Figura 3.16 muestra cómo el esquema de la base de datos EMPRESA (véase la Figura 3.15) se puede visualizar utilizando la notación de los diagramas de clase UML. Los *tipos de entidades* de la Figura 3.15 se modelan como *clases* en la Figura 3.16. Una *entidad* en ER se corresponde con un *objeto* en UML.

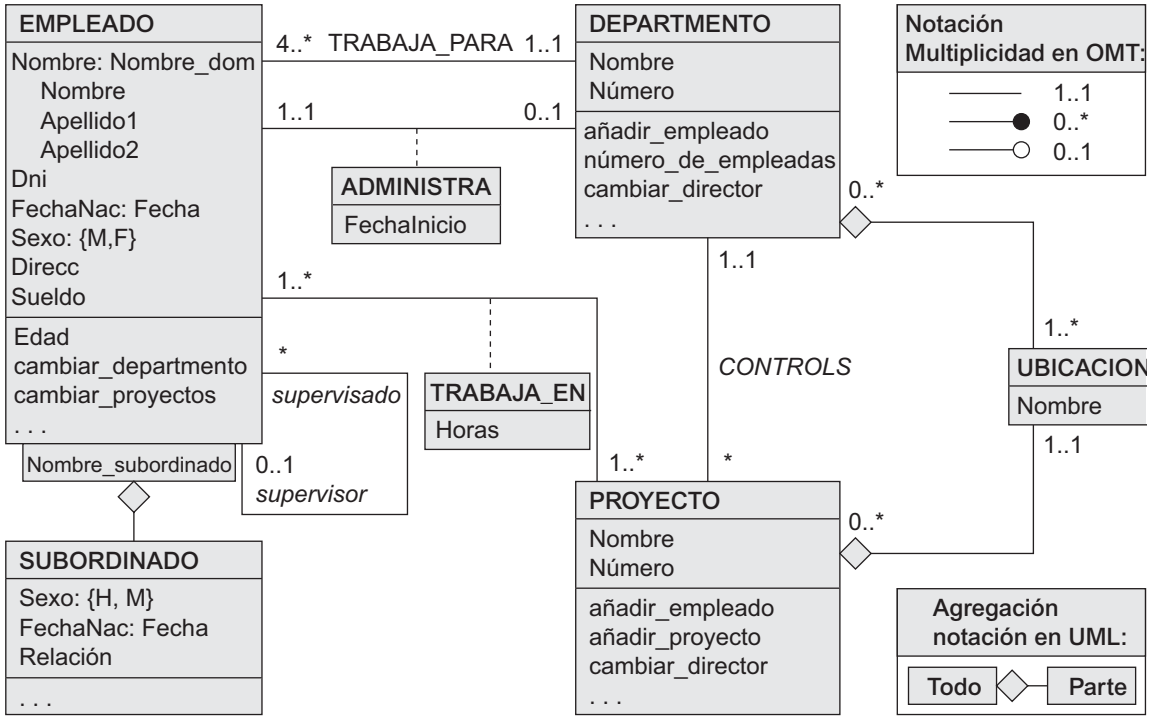
[14] En algunas notaciones, en concreto las que se utilizan en las metodologías de modelado de objetos, como UML, (mín, máx) se coloca en los lados opuestos a los que mostramos. Por ejemplo, para la relación TRABAJA_PARA de la Figura 3.15, el (1,1) estaría en el lado DEPARTAMENTO, y (4,N) estaría en el lado EMPLEADO. Aquí hemos utilizado la notación original de Abrial (1974).

**Figura 3.15.** Los diagramas ER para el esquema de la empresa. Hemos utilizado la notación (mín, máx) y los nombres de papel para especificar las restricciones estructurales.




En los diagramas de clase UML, una **clase** (equivalente a un tipo de entidad en ER) se muestra como un cuadro (véase la Figura 3.16) que incluye tres secciones: la sección superior ofrece el **nombre de la clase**; la sección intermedia incluye los **atributos** de los objetos individuales de la clase; y la última sección incluye las **operaciones** que se pueden aplicar a esos objetos. En los diagramas ER *no* se especifican las operaciones. Tomemos como ejemplo la clase EMPLEADO de la Figura 3.16. Sus atributos son Nombre, Dni, FechaNac, Sexo, Dirección y Sueldo. El diseñador puede especificar opcionalmente el **dominio** de un atributo, si lo desea, colocando el símbolo de dos puntos (:) seguido por el nombre de dominio o descripción, como se ilustra para los atributos Nombre, Sexo y FechaNac de EMPLEADO en la Figura 3.16. Un atributo compuesto se modela como un **dominio estructurado**, como en el caso del atributo Nombre de EMPLEADO. Un atributo multivalor generalmente se modelará como una clase separada, como UBICACIÓN en la Figura 3.16.

**Figura 3.16.** Esquema conceptual de EMPRESA en la notación de diagrama de clase UML.




En la tecnología UML, los tipos de relación se denominan **asociaciones** y las instancias de relación, **vínculos**. Una **asociación binaria** (tipo de relación binaria) se representa como una línea que conecta las clases participantes (tipos de entidad) y, opcionalmente, puede tener un nombre. Un atributo de relación, denominado **atributo de vínculo**, se coloca en un recuadro conectado con la línea de la asociación mediante una línea discontinua. La notación (mín, máx) descrita en la Sección 3.7.4 se utiliza para especificar las restricciones de relación, que en terminología UML se denominan **multiplicidades**. Las multiplicidades se especifican como *mín..máx*, y un asterisco (∗) indica que no hay un límite máximo en la participación. No obstante, las multiplicidades se colocan en los *extremos opuestos de la relación* en comparación con la notación explicada en la Sección 3.7.4 (compare las Figuras 3.15 y 3.16). En UML, un asterisco indica una multiplicidad de 0..*, y un 1 indica una multiplicidad de 1..1. Una relación recursiva (consulte la Sección 3.4.2) se denomina **asociación reflexiva** en UML, y los nombres de papeles (como las multiplicidades) se colocan en los extremos opuestos de una asociación en comparación con la colocación de los nombres de papel en la Figura 3.15. En UML, hay dos tipos de relaciones: asociación y agregación. La **agregación** está pensada para representar una relación entre un objeto completo y sus partes constitutivas, y tiene una notación diagramática distinta. En la Figura 3.16 modelamos las ubicaciones de un departamento y la ubicación sencilla de un proyecto como agregaciones. No obstante la agregación y la asociación no tienen propiedades estructurales diferentes y la elección del tipo de relación que hay que utilizar es algo subjetivo. En el modelo ER, las dos se representan como relaciones.

UML también distingue entre asociaciones (o agregaciones) **unidireccionales** y **bidireccionales**. En el caso unidireccional, la línea que conecta las clases se muestra con una flecha para indicar que sólo se necesita una dirección para acceder a los objetos relacionados. Si no aparece una flecha, se asume la cualidad bidireccional, que es lo predeterminado. Por ejemplo, si siempre esperamos acceder al director de un departamento a partir de un objeto DEPARTAMENTO, dibujaremos la línea de asociación que representa la asociación ADMI-

NISTRA con una flecha desde DEPARTAMENTO hasta EMPLEADO. Además, es posible especificar que las instancias de relación se **ordenen**. Por ejemplo, podríamos especificar que los objetos de empleado relacionados con cada departamento a través de la asociación (relación) TRABAJA_PARA se ordenen por el valor de su atributo FechaNac. Los nombres de asociación (relación) son *opcionales* en UML, y los atributos de relación se muestran en un cuadro conectado con una línea discontinua a la línea que representa la asociación/agregación (consulte FechaInicio y Horas en la Figura 3.16).

Las operaciones dadas en cada clase se derivan de los requisitos funcionales de la aplicación, como se explicó en la Sección 3.1. Normalmente, es suficiente especificar los nombres de operación al principio para las operaciones lógicas que se espera aplicar a los objetos individuales de una clase (véase la Figura 3.16). Al refinar un diseño, se añaden más detalles, como los tipos de argumento exactos (parámetros) para cada operación, más una descripción funcional de cada operación. UML tiene *descripciones de función* y *diagramas de secuencia* para especificar parte de los detalles de operación, pero esto queda fuera del ámbito de nuestra explicación. El Capítulo 12 introducirá algunos de estos diagramas.

Las entidades débiles se pueden modelar utilizando la construcción denominada **asociación cualificada** (o **agregación cualificada**) en UML; esto puede representar tanto la relación identificativa como la clave parcial, que se coloca en un cuadro conectado con la clase propietaria. Es lo que se ilustra en la Figura 3.16 con la clase SUBORDINADO y su agregación cualificada a EMPLEADO. La clave parcial NombreSubordinado se denomina **discriminador** en terminología UML, puesto que su valor distingue los objetos asociados con (relacionados con) el mismo EMPLEADO. Las asociaciones cualificadas no están restringidas al modelado de entidades débiles, y se pueden utilizar para modelar otras situaciones en UML.

# 3.9 Tipos de relación con grado mayor que dos

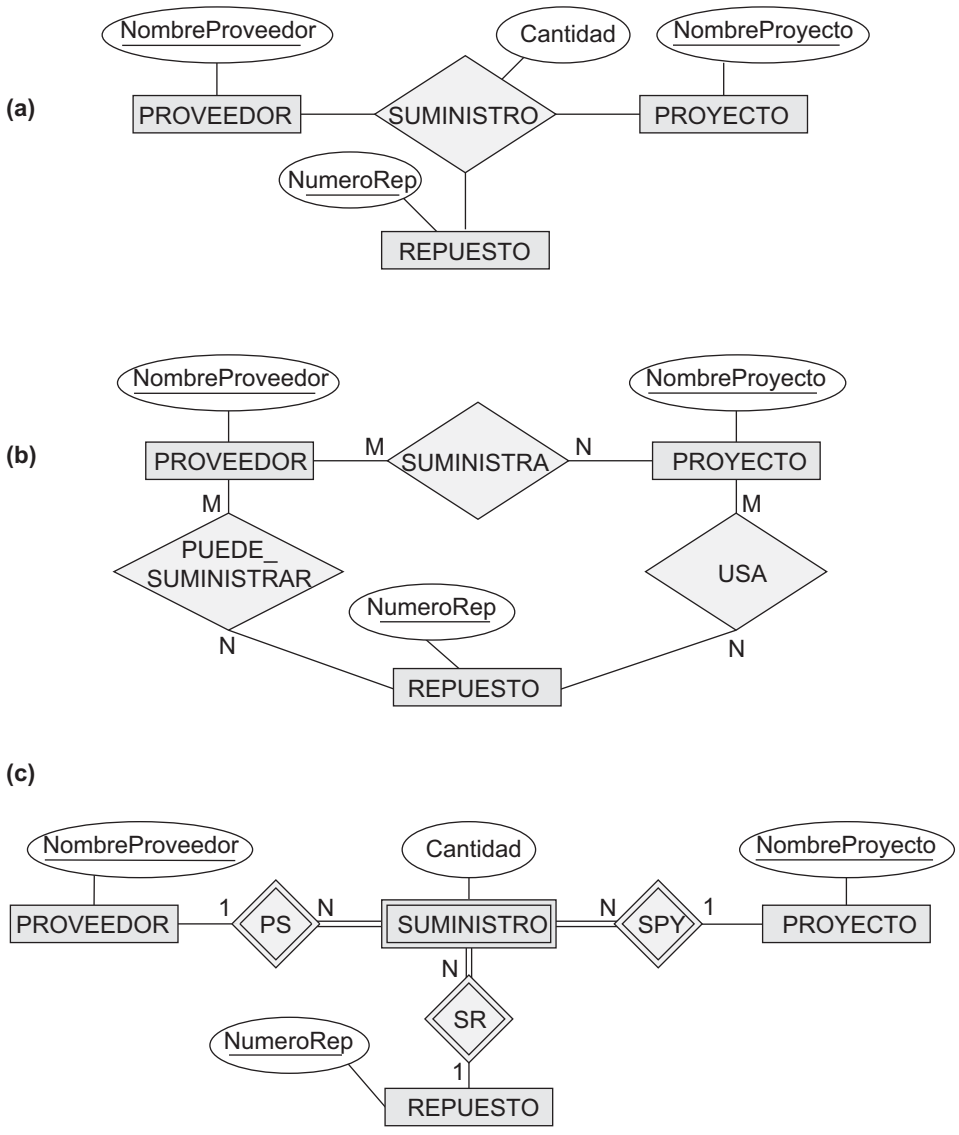
En la Sección 3.4.2 definimos el **grado** de un tipo de relación como el número de tipos de entidades participantes. Un tipo de relación de grado dos es *binario* y un tipo de relación de grado tres, *ternario*. En esta sección explicamos más en detalle las diferencias entre las relaciones binarias y de grado superior, cuándo elegir relaciones de grado superior o binarias, y las restricciones en las relaciones de grado superior.

## 3.9.1 Elección entre relaciones binarias y ternarias (o de grado superior)

En la Figura 3.17(a) se muestra la notación de diagrama ER para un tipo de relación ternario: el esquema para el tipo de relación SUMINISTRO que se mostraba a nivel de instancia en la Figura 3.10. Recuerde que el conjunto de relación de SUMINISTRO es un conjunto de instancias de relación ($s$, $j$, $p$), donde $s$ es un PROVEEDOR que actualmente está suministrando un REPUESTO $p$ a un PROYECTO $j$. En general, un tipo de relación $R$ de grado $n$ tendrá $n$ bordes en un diagrama ER, uno conectando $R$ con cada tipo de entidad participante.

La Figura 3.17(b) muestra un diagrama ER para los tres tipos de relación binaria PUEDE_SUMINISTRAR, USA y SUMINISTRA. En general, un tipo de relación ternaria representa información diferente que tres tipos de relación binaria. Considere los tres tipos de relación binaria PUEDE_SUMINISTRAR, USA y SUMINISTRA. Suponga que PUEDE_SUMINISTRAR, entre PROVEEDOR y REPUESTO, incluye una instancia ($s$, $p$) cuando un proveedor *s puede suministrar* el repuesto $p$ (a cualquier proyecto); USA, entre PROYECTO y REPUESTO, incluye una instancia ($j$, $p$) cuando el proyecto $j$ utiliza el repuesto $p$; y SUMINISTRA, entre PROVEEDOR y PROYECTO, incluye una instancia ($s$, $j$) cuando el proveedor $s$ suministra algún repuesto al proyecto $j$. La existencia de tres instancias de relación ($s$, $p$), ( $j$, $p$) y ($s$, $j$) en PUEDE_SUMINISTRAR, USA y SUMINISTRA, respectivamente, no implica necesariamente que exista una instancia ($s$, $j$, $p$) en la relación ternaria SUMINISTRO, porque el *significado es diferente.* A menudo es complejo decidir si una relación en particular debe representarse como un tipo de relación de grado $n$ o si debe dividirse en varios tipos de relación de

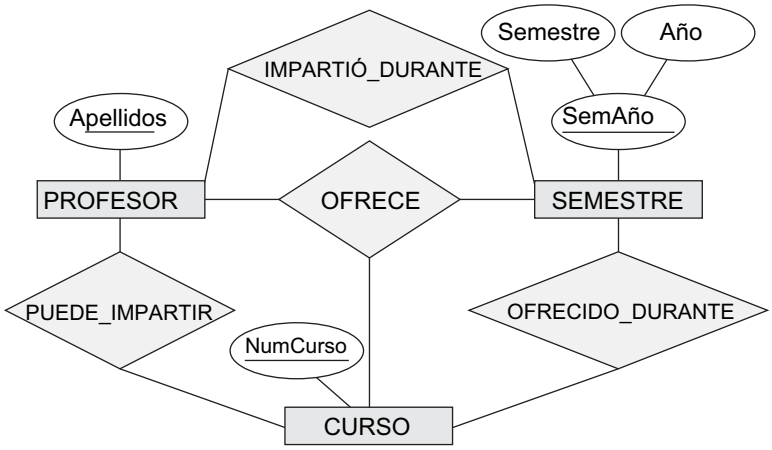
**Figura 3.17.** Tipos de relaciones ternarias . (a) La relación SUMINISTRO. (b) Tres relaciones binarias no son equivalentes a SUMINISTRO. (c) SUMINISTRO representada como un tipo de entidad débil.




menor grado. El diseñador debe basar esta decisión en la semántica o significado de la situación particular que se está representando. La solución típica es incluir la relación ternaria *más* una o más relaciones binarias, si representan varios significados y todas son necesarias para la aplicación.

Algunas herramientas de diseño de bases de datos están basadas en variaciones del modelo ER que sólo permiten las relaciones binarias. En este caso, una relación ternaria como SUMINISTRO debe representarse como un tipo de entidad débil, sin clave parcial y con tres relaciones identificativas. Los tres tipos de entidad participantes, PROVEEDOR, REPUESTO y PROYECTO, son conjuntamente los tipos de entidad propietaria (véase la Figura 3.17[c]). Por tanto, una entidad en el tipo de entidad débil SUMINISTRO de la Figura 3.17(c) queda identificada por la combinación de sus tres entidades propietarias de PROVEEDOR, REPUESTO y PROYECTO.

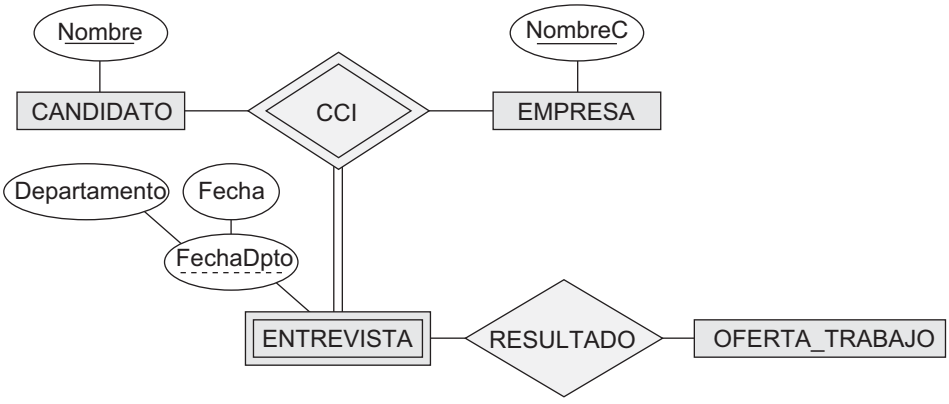
**Figura 3.18.** Otro ejemplo de relación ternaria frente a relación binaria.




También es posible representar la relación ternaria como un tipo de entidad regular introduciendo una clave artificial o sustituta. En este ejemplo, podría utilizarse un atributo clave IdSuministro para el tipo de entidad suministro, convirtiéndolo en un tipo de entidad regular. Tres relaciones 1:N binarias relacionan SUMINISTRO con los tres tipos de entidad participantes.

En la Figura 3.18 se muestra otro ejemplo. El tipo de relación ternaria OFRECE representa información de los profesores que ofrecen cursos durante los semestres; por tanto, incluye una instancia de relación ($p$, $s$, $c$) siempre que el PROFESOR $p$ ofrece un CURSO $c$ durante el SEMESTRE $s$. Los tres tipos de relación binaria de la Figura 4.12 significan lo siguiente: PUEDE_IMPARTIR relaciona un curso con los profesores que pueden *impartirlo*, IMPARTIÓ_DURANTE relaciona un semestre con los profesores que *impartieron algún curso* durante ese semestre, y OFRECIDO_DURANTE relaciona un semestre con los cursos ofrecidos durante ese semestre *por cualquier profesor*. Estas relaciones ternarias y binarias representan diferente información, pero debe haber algunas restricciones entre las relaciones. Por ejemplo, en OFRECE no debería existir una instancia de relación ($p$, $s$, $c$) *a menos* que exista una instancia ($p$, $s$) en IMPARTIÓ_DURANTE, que exista una instancia ($s$, $c$) en OFRECIDO_DURANTE, y que exista una instancia ($p$, $c$) en PUEDE_IMPARTIR. No obstante, lo opuesto no siempre se cumple; podemos tener instancias ($p$, $s$), ($s$, $c$) y ($p$, $c$) en las tres relaciones binarias sin que haya una instancia ($p$, $s$, $c$) en OFRECE. En este ejemplo, y basándose en los significados de las relaciones, podemos inferir las instancias de IMPARTIÓ_DURANTE y OFRECIDO_DURANTE de las instancias en

**Figura 3.19.** Un tipo de entidad débil ENTREVISTA con un tipo de relación identificativa ternaria.



OFRECE, pero no podemos inferir las instancias de PUEDE_IMPARTIR; por consiguiente, IMPARTIÓ_DURANTE y OFRECIDO_DURANTE son redundantes y se pueden omitir.

Aunque por lo general tres relaciones binarias *no pueden* reemplazar a una relación ternaria, esto puede ser válido bajo ciertas *restricciones adicionales*. En nuestro ejemplo, si la relación PUEDE_IMPARTIR es 1:1 (un profesor puede impartir un curso, y un curso puede ser impartido por un solo profesor), entonces la relación ternaria OFRECE se puede omitir porque puede inferirse de las tres relaciones binarias PUEDE_IMPARTIR, IMPARTIÓ_DURANTE y OFRECIDO_DURANTE. El diseñador del esquema debe analizar el significado de cada situación específica para decidir los tipos de relaciones binarias y ternarias que son necesarias.

Es posible tener un tipo de entidad débil con un tipo de relación identificativa ternaria (o *n*-ary). En este caso, el tipo de entidad débil puede tener *varios* tipos de entidad propietarias. En la Figura 3.19 se ofrece un ejemplo.

### 3.9.2 Restricciones en las relaciones ternarias (o de grado superior)

Hay dos notaciones para especificar las restricciones estructurales en las relaciones *n*-ary. *Deben utilizarse ambos* si es importante especificar completamente las restricciones estructurales de una relación ternaria o de grado superior. La primera notación está basada en la notación de la razón de cardinalidad de las relaciones binarias de la Figura 3.2, donde se utiliza 1, M o N en cada arco de participación (los símbolos M y N significan *muchos* o *cualquier cantidad*).[15] Permítanos ilustrar esta restricción con la relación SUMINISTRO de la Figura 3.17.

Recuerde que el conjunto de relación de SUMINISTRO es un conjunto de instancias de relación ($s$, $j$, $p$), donde $s$ es un PROVEEDOR, $j$ es un PROYECTO y $p$ es un REPUESTO. Suponga que existe una restricción según la cual sólo se puede utilizar un proveedor para una combinación proyecto-repuesto particular (sólo un proveedor suministra un repuesto particular a un proyecto concreto). En este caso, colocamos un 1 en la participación PROVEEDOR, y M, N en las participaciones PROYECTO, REPUESTO de la Figura 3.17. Esto especifica la restricción de que una combinación ($j$, $p$) en particular puede aparecer a lo sumo una vez en el conjunto de relación porque cada combinación (PROYECTO, REPUESTO) determina sin lugar a dudas un único proveedor. Por tanto, cualquier instancia de relación ($s$, $j$, $p$) es identificada excepcionalmente en el conjunto de relación por su combinación ($j$, $p$), lo que convierte a ($j$, $p$) en una clave para el conjunto de relación. En esta notación, no es necesario que las participaciones que tienen una especificada sean parte de la clave de identificación del conjunto de relación.[16]

La segunda notación está basada en la notación (mín, máx) de la Figura 3.15 para las notaciones binarias. Una pareja (mín, máx) en una participación especifica aquí que cada entidad está relacionada con al menos *mín* relaciones y con a lo sumo *máx* instancias de relación en el conjunto de relación. Estas restricciones no conllevan determinar la clave de una relación *n*-ary, donde $n > 2$,[17] pero especifica un tipo de restricción diferente que restringe la cantidad de instancias de relación en las que cada entidad puede participar.

## 3.10 Resumen

En este capítulo hemos presentado los conceptos de modelado de un modelo de datos conceptual de nivel alto, el modelo Entidad-Relación (ER). Hemos empezado explicando el papel que un modelo de este tipo juega en el proceso de diseño de una base de datos, y luego presentamos un conjunto de requisitos para la base de datos EMPRESA, que es uno de los ejemplos que utilizamos a lo largo del libro. Definimos los conceptos básicos

[15] Esta notación permite determinar la clave de la relación, como se explica en el Capítulo 7.

[16] Esto también es cierto para las razones de cardinalidad de las relaciones binarias.

[17] Las restricciones (mín, máx) pueden determinar las claves para las relaciones binarias.

de entidad y atributo del modelo ER. Después hablamos de los valores NULL y presentamos los distintos tipos de atributos, que se pueden anidar arbitrariamente para producir atributos complejos:

- Simple o atómico.
- Compuesto.
- Multivalor.

También explicamos brevemente los atributos almacenados frente a los derivados, para después adentrarnos en los conceptos del modelo ER relativos al nivel de esquema o "intención":

- Tipos de entidad y sus conjuntos de entidades correspondientes.
- Atributos clave de los tipos de entidad.
- Conjuntos de valores (dominios) de atributos.
- Tipos de relaciones y sus conjuntos de relaciones correspondientes.
- Participaciones de los tipos de entidades en los tipos de relaciones.

Hemos presentado dos métodos para especificar las restricciones estructurales en los tipos de relaciones. El primer método distingue dos tipos de restricciones estructurales:

- Razones de cardinalidad (1:1, 1:N, M:N en las relaciones binarias).
- Restricciones de participación (total, parcial).

Otro método alternativo para especificar las restricciones estructurales consiste en especificar una cantidad mínima y máxima (mín, máx) de participación de cada tipo de entidad en un tipo de relación. Asimismo, explicamos los tipos de entidad débiles y los conceptos relacionados de tipos de entidad propietarios, identificación de tipos de relación y atributos de clave parcial.

Los esquemas de Entidad-Relación se pueden representar diagramáticamente como diagramas ER. Hemos visto cómo diseñar un esquema ER para la base de datos EMPRESA, definiendo en primer lugar los tipos de entidades y sus atributos, para después refinar el diseño a fin de incluir los tipos de relaciones. Mostramos el diagrama ER correspondiente al esquema de la base de datos EMPRESA y explicamos algunos de los conceptos básicos de los diagramas de clase UML y de cómo se relacionan con los conceptos del modelo ER. También hemos descrito más en detalle los tipos de relación ternaria de alto nivel, así como las circunstancias que los diferencian de las relaciones binarias.

Los conceptos de modelado ER que hemos presentado hasta ahora (tipos de entidades, tipos de relaciones, atributos, claves y restricciones estructurales) pueden modelar las típicas aplicaciones de bases de datos de procesamiento de datos empresariales. No obstante, las aplicaciones más modernas y complejas (por ejemplo, diseño en ingeniería, sistemas de información médica o telecomunicaciones) requieren conceptos adicionales si queremos modelarlas con mayor precisión. En el Capítulo 4 explicamos algunos conceptos avanzados sobre modelado, mientras que en el Capítulo 24 volveremos a ver las técnicas de modelado de datos.

## Preguntas de repaso

**3.1.** Explique el papel de un modelo de datos de alto nivel en el proceso de diseño de una base de datos.

**3.2.** Enumere los distintos casos donde podría resultar apropiado utilizar un valor NULL.

**3.3.** Defina los siguientes términos: *entidad, atributo, valor de atributo, instancia de relación, atributo compuesto, atributo multivalor, atributo derivado, atributo complejo, atributo clave* y *conjunto de valores* (*dominio*).

**3.4.** ¿Qué es un tipo de entidad? ¿Qué es un conjunto de entidades? Explique las diferencias entre una entidad, un tipo de entidad y un conjunto de entidades.

**3.5.** Explique la diferencia entre un atributo y un conjunto de valores.

**3.6.** ¿Qué es un tipo de relación? Explique las diferencias entre una instancia de relación, un tipo de relación y un conjunto de relaciones.

**3.7.** ¿Qué es un rol de participación? ¿Cuándo es necesario utilizar nombres de rol en la descripción de los tipos de relaciones?

**3.8.** Describa las dos alternativas que hay para especificar las restricciones estructurales en los tipos de relaciones. ¿Cuáles son las ventajas y los inconvenientes de cada una?

**3.9.** ¿Bajo qué condiciones un atributo de un tipo de relación binaria puede migrarse para convertirse en un atributo de uno de los tipos de entidad participantes?

**3.10.** Cuando pensamos en las relaciones como en atributos, ¿cuáles son los conjuntos de valores de esos atributos? ¿Qué clase de modelos de datos está basada en este concepto?

**3.11.** ¿Qué se entiende por tipo de relación recursivo? Ponga algunos ejemplos.

**3.12.** ¿Cuándo se utiliza el concepto de tipo de entidad débil en el modelado de datos? Defina los términos *tipo de entidad propietaria, tipo de entidad débil, identificación del tipo de entidad* y *clave parcial*.

**3.13.** ¿Una relación de identificación de un tipo de entidad débil puede ser de un grado mayor que dos? Ofrezca algunos ejemplos para ilustrar su respuesta.

**3.14.** Explique las convenciones para visualizar un esquema ER como un diagrama ER.

**3.15.** Explique las convenciones de denominación que se utilizan para los diagramas de esquema ER.

## Ejercicios

**3.16.** Considere el siguiente conjunto de requisitos para una base de datos UNIVERSIDAD que se utiliza para hacer un seguimiento del certificado de estudios de los estudiantes. Es parecido pero no idéntico a la base de datos de la Figura 1.2:

**a.** La universidad registra el nombre, el número de estudiante, el dni, la dirección y el teléfono actuales, la dirección y el teléfono permanentes, la fecha de nacimiento, el sexo, la clase (estudiante de primer año, de segundo año,..., diplomado), departamento principal, departamento menor (si lo hay) y programa de grado (B.A., B.S., . . ., Ph.D.). Algunas aplicaciones de usuario necesitan referirse a la ciudad, la provincia y el código postal de la dirección permanente del estudiante, así como a los apellidos. Tanto el DNI como el número de estudiante tienen valores únicos para cada estudiante.

**b.** Cada departamento está descrito por un nombre, un código de departamento, un número de oficina, un teléfono de la oficina y la universidad. El nombre y el código tienen valores únicos para cada departamento.

**c.** Cada curso tiene un nombre de curso, una descripción, un número de curso, un número de horas por semestre, un nivel y el departamento que lo ofrece. El valor del número de curso es único para cada curso.

**d.** Cada sección tiene un profesor, un semestre, un año, un curso y un número de sección. Este último distingue las secciones del mismo curso que se imparten durante el mismo semestre/año; sus valores son 1, 2, 3, . . ., hasta el número de secciones impartidas durante cada semestre.

**e.** Un informe de calificaciones consta del estudiante, la sección, la letra de la calificación y un grado numérico (0, 1, 2, 3, o 4).

Diseñe un esquema ER para esta aplicación y dibuje un diagrama ER para el esquema. Especifique los atributos clave de cada tipo de entidad y las restricciones estructurales de cada tipo de relación.

Anote cualquier requisito no especificado y haga las suposiciones adecuadas para realizar una especificación completa.

**3.17.** Los atributos compuestos y multivalor se pueden anidar hasta cualquier nivel. Suponga que queremos diseñar un atributo para un tipo de entidad ESTUDIANTE para hacer un seguimiento de su formación universitaria. Un atributo así tendrá una entrada por cada universidad en la que haya estudiado, y cada una de estas entradas estará compuesta por el nombre de la universidad, las fechas de inicio y fin, las calificaciones (los grados otorgados por esa universidad, si los hubiera) y las entradas con los certificados de estudios (cursos completados en esa universidad, si fuera aplicable). Cada entrada de grado contiene el nombre del grado y el mes y el año en que se consiguió ese grado, y cada entrada de certificado de estudios contiene un nombre del curso, el semestre, el año y la calificación. Diseñe un atributo para almacenar esta información. Utilice las convenciones de la Figura 3.5.

**3.18.** Muestre un diseño alternativo para el atributo descrito en el Ejercicio 3.17 que utilice únicamente tipos de entidad (incluyendo tipos de entidad débiles, si es necesario) y tipos de relación.

**3.19.** Considere el diagrama ER de la Figura 3.20, que muestra un esquema simplificado para un sistema de reservas en aerolíneas. Extraiga del diagrama ER los requisitos y las restricciones que produjeron este esquema. Intente ser tan preciso como sea posible en su especificación de requisitos y restricciones.

**3.20.** En los Capítulos 1 y 2 explicamos el entorno de una base de datos y los usuarios de las bases de datos. Podemos considerar muchos tipos de entidad para describir un entorno semejante, como un DMBS, una base de datos almacenada, un DBA y un diccionario catálogo/datos. Intente especificar todos los tipos de entidad que pueden describir completamente un sistema de bases de datos y su entorno; después, especifique los tipos de relación entre ellos y dibuje un diagrama ER para describir un entorno de bases de datos general semejante.

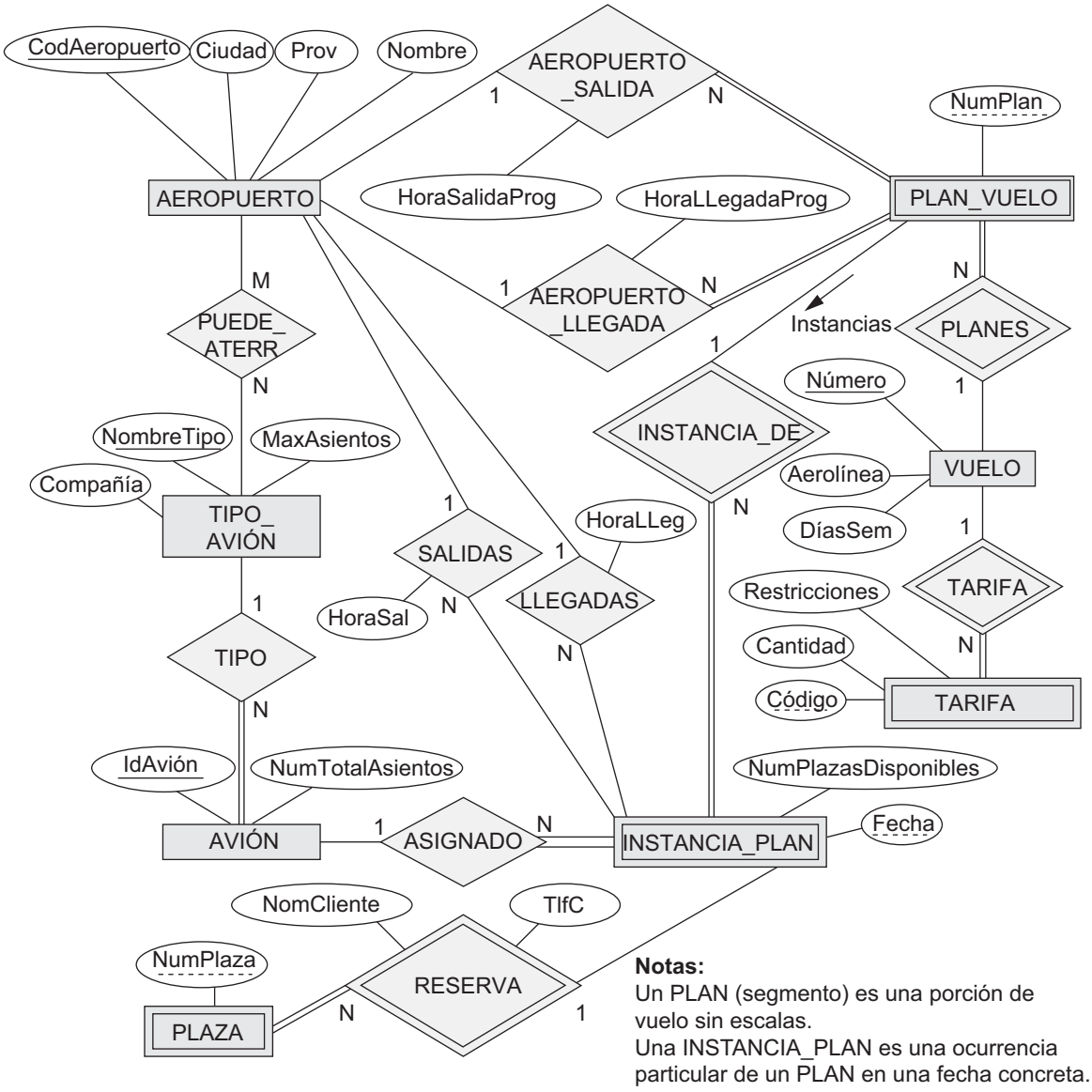
**3.21.** Diseñe un esquema ER para seguir la información sobre las votaciones llevadas a cabo en la Cámara de Diputados de Estados Unidos durante la sesión congresional actual de dos años. La base de datos tiene que registrar el nombre de todos los estados (por ejemplo, 'Texas', 'Nueva York', 'California') e incluir la región del estado (cuyo dominio es {'Noreste', 'Medio oeste', 'Sureste', 'Suroeste', 'Oeste'}). Cada PERSONA_CONGRESO de la Cámara de Diputados aparece descrita por su Nombre más el Distrito representado, la FechaInicio de cuando ese diputado fue elegido por primera vez, y el Partido político al que esa persona pertenece (cuyo dominio es {'Republicano','Demócrata', 'Independiente', 'Otro'}). La base de datos hace un seguimiento de cada PROYECTOLEY (por ejemplo, propuesta de ley), incluyendo el NombrePropuesta, la FechaDeVoto de la propuesta, si la propuesta fue o no aprobada (AprobadaONo) (cuyo dominio es {'Sí', 'No'}), y el Promotor (el o los diputados que promovieron la propuesta de ley). La base de datos también registra la votación de los diputados (el dominio del atributo Voto es {'Sí', 'No', 'Abstención', 'Ausencia'}). Dibuje el diagrama del esquema ER para esta aplicación. Declare explícitamente todas las suposiciones que haga.

**3.22.** Se está construyendo una base de datos para hacer un seguimiento de los equipos y los partidos de una liga deportiva. Un equipo tiene un determinado número de jugadores, y no todos juegan en cada partido. Es deseable hacer un seguimiento de los jugadores que disputan cada partido y por cada equipo, las posiciones en las que jugaron en ese partido y el resultado del mismo. Diseñe un diagrama de esquema ER para esta aplicación, describiendo las suposiciones que haga. Elija su deporte favorito (por ejemplo, fútbol, baloncesto, béisbol).

**3.23.** Considere el diagrama ER de la Figura 3.21 para parte de la base de datos de un BANCO. Cada banco puede tener varias sucursales, y cada sucursal puede tener varias cuentas y préstamos.

**a.** Liste los tipos de entidad (no débiles) del diagrama ER.

**Figura 3.20.** Diagrama ER para el esquema de una base de datos AEROLÍNEA.




**b.** ¿Hay algún tipo de entidad débil? En ese caso, proporcione su nombre, la clave parcial y la relación de identificación.

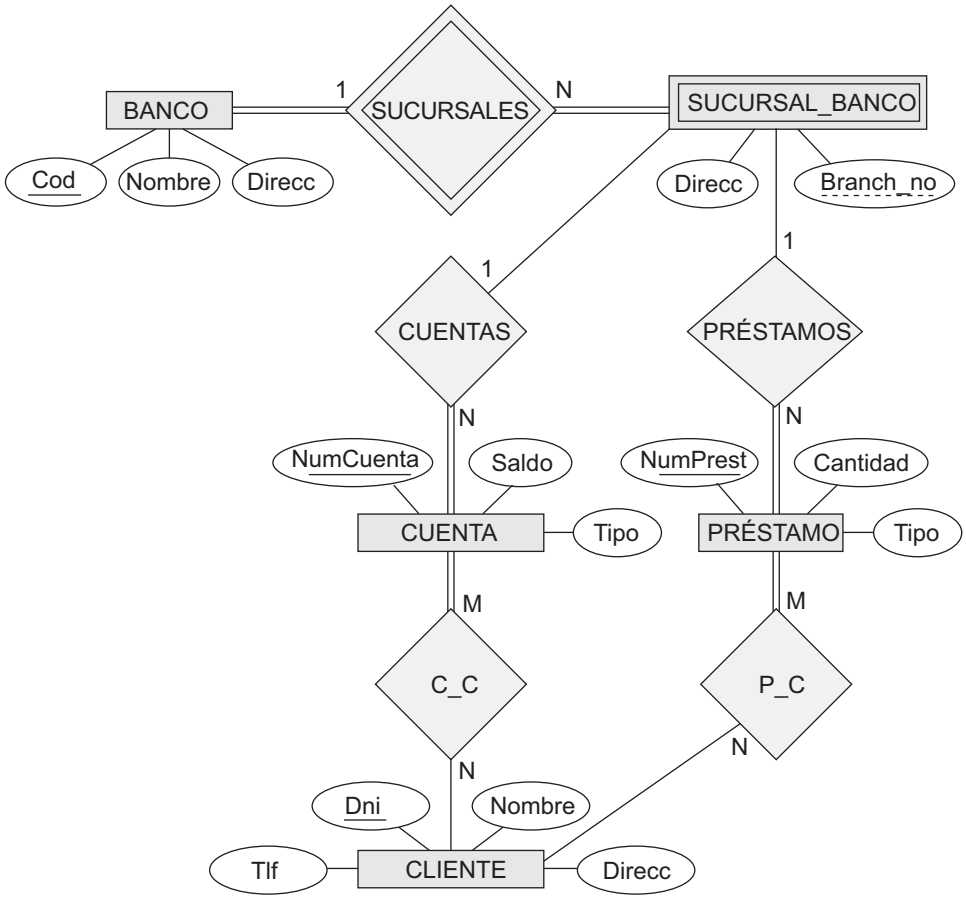
**c.** ¿Qué restricciones especifican en este diagrama la clave parcial y la relación de identificación del tipo de entidad débil?

**d.** Liste los nombres de todos los tipos de relación y especifique la restricción (mín, máx) de cada participación de un tipo de entidad en un tipo de relación. Justifique sus opciones.

**e.** Enumere brevemente los requisitos de usuario que conducen a este diseño de esquema ER.

**f.** Suponga que cada cliente debe tener al menos una cuenta pero está restringido a tener un máximo de dos préstamos simultáneos, y que una sucursal de un banco no puede tener más de 1.000 préstamos. ¿Cómo se muestra esto en las restricciones (mín, máx)?

**Figura 3.21.** Diagrama ER para el esquema de la base de datos de un BANCO.

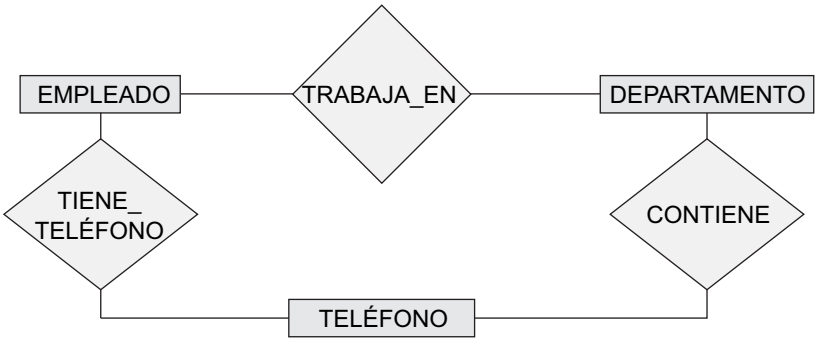



**3.24.** Considere el diagrama ER de la Figura 3.22. Asuma que un empleado puede trabajar en hasta dos departamentos o que puede no ser asignado a cualquier departamento. Suponga que cada departamento debe tener un número de teléfono y que puede tener hasta tres. Proporcione las restricciones (mín, máx) en este diagrama. *Explique claramente las suposiciones adicionales que haga*. En este ejemplo, ¿bajo qué condiciones sería redundante la relación TIENE_TELÉFONO?

**3.25.** Considere el diagrama ER de la Figura 3.23. Un curso puede o no utilizar un libro de texto, pero un texto, por definición, es un libro que se utiliza en algún curso. Un curso no puede utilizar más de cinco libros. Los profesores imparten de dos a cuatro cursos. Proporcione las restricciones (mín, máx) de este diagrama. *Explique claramente las suposiciones adicionales que haga*. Si añadimos la relación ADOPTA entre PROFESOR y TEXTO, ¿qué restricciones (mín, máx) especificaría? ¿Por qué?

**3.26.** Considere un tipo de entidad SECCIÓN en la base de datos UNIVERSIDAD que describe la sección que ofrece los cursos. Los atributos de SECCIÓN son NumSección, Semestre, Año, NumCurso, Profesor, NumSala (donde se imparte la sección), Edificio (donde se imparte la sección), DíasSemana (dominio de las posibles combinaciones de días laborables en las que puede ofrecerse una sección, {'LXV', 'XV', 'MJ', etcétera}) y Horas (dominio de todos los periodos de tiempo posibles durante los que se ofrecen las secciones {'9–9:50 A.M.', '10–10:50 A.M.', . . ., '3:30–4:50 P.M.', '5:30–6:20 P.M.', etcétera}). Suponga que NumSección es un valor único por cada curso

**Figura 3.22.** Parte de un diagrama ER para una base de datos EMPRESA.




dentro de una combinación semestre/año particular (es decir, si un curso se ofrece varias veces durante un semestre en particular, sus ofertas de sección se numeran como 1, 2, 3, etcétera). Hay varias claves compuestas por sección, y algunos atributos están compuestos por más de una clave. Identifique tres claves compuestas y muestre cómo se pueden representar en un diagrama de esquema ER.

**3.27.** Las razones de cardinalidad a menudo dictaminan el diseño detallado de una base de datos. La razón de cardinalidad depende del significado real de los tipos de entidad implicados y queda definida por la aplicación específica. Para las siguientes relaciones binarias, sugiera las razones de cardinalidad basándose en el significado de sentido común de los tipos de entidad. Explique claramente las suposiciones que haga.

| | **Entidad 1** | **Razón de cardinalidad** | **Entidad 2** |
|---|---|---|---|
| 1. | ESTUDIANTE | _____ | DNI |
| 2. | ESTUDIANTE | _____ | PROFESOR |
| 3. | AULA | _____ | PARED |
| 4. | PAÍS | _____ | PRESIDENTE_ACTUAL |
| 5. | CURSO | _____ | LIBROTEXTO |
| 6. | ELEMENTO (que se puede encontrar en un pedido) | _____ | PEDIDO |
| 7. | ESTUDIANTE | _____ | CLASE |
| 8. | CLASE | _____ | PROFESOR |
| 9. | PROFESOR | _____ | OFICINA |
| 10. | ARTÍCULO_SUBASTA_EBAY | _____ | OFERTA_EBAY |

**3.28.** Considere el esquema ER para la base de datos PELÍCULAS de la Figura 3.24.

Asuma que PELÍCULAS es una base de datos rellena. Actor se utiliza como término genérico e incluye actrices. Dadas las restricciones mostradas en el esquema ER, responda a las siguientes afirmaciones con Verdadero, Falso o Quizás. Asigne esta última respuesta a las afirmaciones que, aun no pudiendo mostrarse explícitamente como Verdaderas, tampoco se puede probar que sean Falsas basándose en el esquema mostrado. Justifique sus respuestas.

**Figura 3.23.** Parte de un diagrama ER para una base de datos CURSOS.

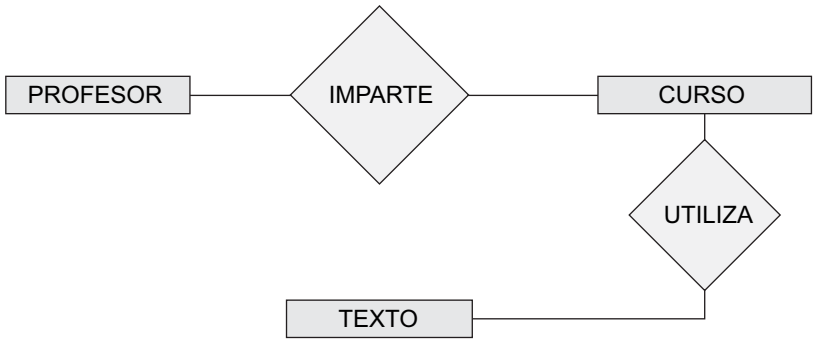
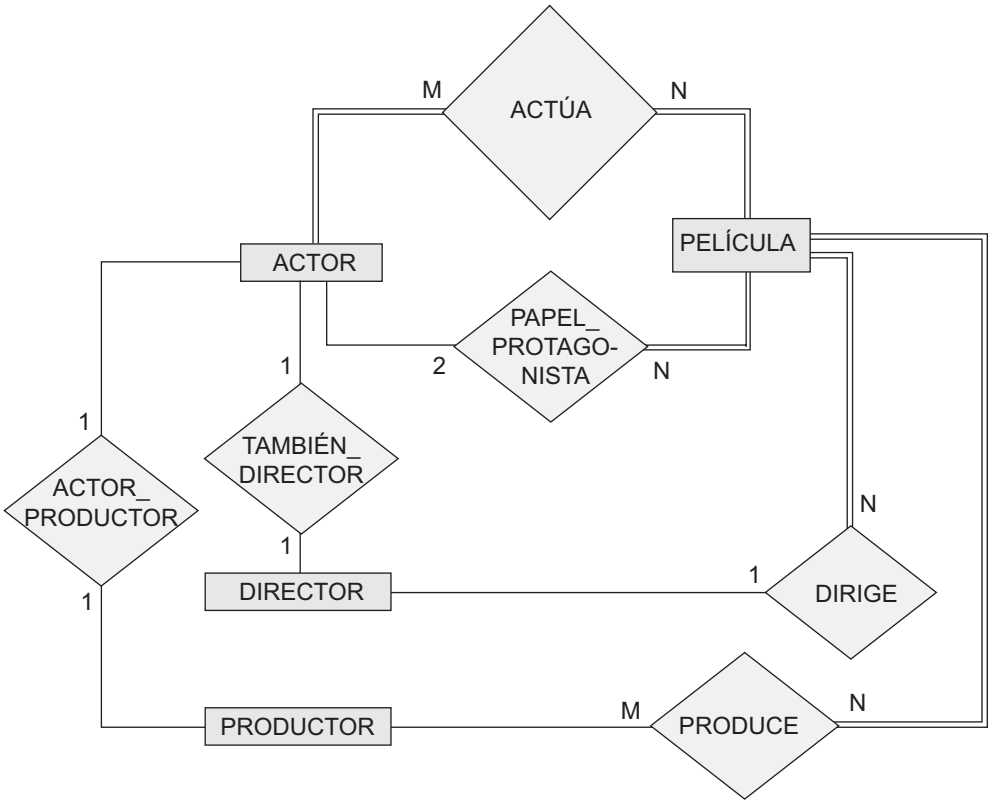



**Figura 3.24.** Diagrama ER para el esquema de una base de datos PELÍCULAS.




**a.** En esta base de datos no hay ningún actor que no haya actuado en ninguna película.
**b.** Hay algunos actores que han actuado en más de diez películas.
**c.** Algunos actores han sido protagonistas en varias películas.
**d.** Una película sólo puede tener un máximo de dos protagonistas.
**e.** Cada director ha sido actor en alguna película.
**f.** Ningún productor ha sido actor alguna vez.
**g.** Un productor no puede ser actor en alguna otra película.

**h.** Hay películas con más de una docena de actores.

**i.** Algunos productores también han sido directores.

**j.** La mayoría de las películas tienen un director y un productor.

**k.** Algunas películas tienen un director pero varios productores.

**l.** Hay algunos actores que han interpretado el papel de protagonista, dirigido una película y producido alguna película.

**m.** Ninguna película tiene un director que también haya actuado en ella.

**3.29.** Dado el esquema ER para la base de datos PELÍCULAS de la Figura 3.24, dibuje un diagrama de instancia utilizando tres películas que se hayan proyectado recientemente. Dibuje las instancias de cada tipo de entidad: PELÍCULAS, ACTORES, PRODUCTORES, DIRECTORES implicados; cree las instancias de las relaciones para estas películas tal y como existen en la realidad.

**3.30.** Ilustre el diagrama UML para el Ejercicio 3.16. Su diseño UML debe observar los siguientes requisitos:

**a.** Un estudiante debe tener la posibilidad de calcular su nota media y añadir o descartar especializaciones principales y secundarias.

**b.** Cada departamento debe ser capaz de añadir o eliminar cursos, y de contratar o despedir profesores.

**c.** Cada profesor debe ser capaz de asignar o cambiar la nota de un estudiante en un curso.

*Nota:* Algunas de estas funciones pueden extenderse por varias clases.

## Ejercicios de práctica

**3.31.** Considere la base de datos UNIVERSIDAD descrita en el Ejercicio 3.16. Cree un esquema ER para ella utilizando una herramienta de modelado de datos como, por ejemplo, ERWin o Rational Rose.

**3.32.** Considere una base de datos PEDIDOS_CORREO en la que los empleados registran los pedidos de piezas por parte de los clientes. Los requisitos en cuanto a datos son los siguientes:

- La empresa de venta por correo tiene empleados identificados por un número de empleado único, además del nombre, los apellidos y el código postal.
- Cada cliente de la empresa está identificado mediante un número de cliente único, el nombre, los apellidos y el código postal.
- Cada pieza o repuesto vendido por la empresa está identificado por un número de repuesto único, un nombre, un precio y la cantidad en stock.
- Cada pedido efectuado por un cliente es registrado por un empleado y se le asigna un número de pedido que es único. Cada pedido contiene la cantidad especificada de uno o más repuestos, la fecha de recibo y la fecha de envío estimada. También se registra la fecha de envío real.

Diseñe un diagrama Entidad-Relación para esta base de datos y construya un diseño utilizando una herramienta de modelado de datos como ERWin o Rational Rose.

**3.33.** Considere una base de datos CINE en la que se registra información relativa a la industria cinematográfica. Los requisitos de datos se resumen a continuación:

- Cada película está identificada por su título y año de proyección. Además, la película tiene una duración en minutos, una productora y está clasificada según uno o más géneros (terror, acción, drama, etcétera). Cada película tiene uno o más directores, y uno o más actores, además de un resumen de la trama. Por último, cada película consta de ninguna o más citas reseñables por parte de los actores que aparecen en ella.
- Los actores están identificados por su nombre y fecha de nacimiento, y aparecen en una o más películas. Cada actor tiene un papel en la película.

- Los directores también están identificados por su nombre y fecha de nacimiento, y dirigen una o más películas. Es posible que un director también actúe en alguna película (incluyendo alguna que él o ella también haya dirigido).
- Las productoras están identificadas por su nombre y una dirección. Una productora produce una o más películas.

Diseñe un diagrama Entidad-Relación para esta base de datos e introduzca el diseño utilizando una herramienta de modelado de datos como ERWin o Rational Rose.

**3.34.** Considere el diagrama ER de la base de datos AEROLÍNEA de la Figura 3.20. Cree este diseño mediante una herramienta de modelado de datos como ERWin o Rational Rose.

## Bibliografía seleccionada

El modelo Entidad-Relación fue introducido por Chen (1976), y en Schmidt and Swenson (1975), Wiederhold and Elmasri (1979) y Senko (1975) aparecen trabajos relacionados. Desde entonces, se han sugerido numerosas modificaciones para el modelo ER. En nuestra presentación hemos incorporado algunas de ellas. Las restricciones estructurales en las relaciones se explican en Abrial (1974), Elmasri y Wiederhold (1980) y Lenzerini and Santucci (1983). Los atributos multivalor y compuestos se incorporan al modelo ER en Elmasri y otros (1985). Aunque no explicamos lenguajes para el modelo ER y sus extensiones, ha habido varias propuestas para dichos lenguajes. Elmasri and Wiederhold (1981) propuso el lenguaje de consulta más amplio para el modelo ER. Markowitz y Raz (1983) propuso otro lenguaje de consulta ER. Senko (1980) presentó un lenguaje de consulta para el modelo DIAM de Senko. Parent y Spaccapietra (1985) presentó un conjunto formal de operaciones denominado álgebra de ER. Gogolla y Hohenstein (1991) presentó otro lenguaje formal para el modelo ER. En Campbell y otros (1985) se presentó un conjunto de operaciones ER y se mostró que estaban relacionalmente completas. Desde 1979 viene celebrándose regularmente una conferencia para la difusión de los resultados de la investigación sobre el modelo ER. La conferencia, ahora conocida como International Conference on Conceptual Modeling, se ha celebrado en Los Ángeles (ER 1979, ER 1983, ER 1997), Washington, D.C. (ER 1981), Chicago (ER 1985), Dijon, Francia (ER 1986), Nueva York (ER 1987), Roma (ER 1988), Toronto (ER 1989), Lausanne, Suiza (ER 1990), San Mateo, California (ER 1991), Karlsruhe, Alemania (ER 1992), Arlington, Texas (ER 1993), Manchester, Inglaterra (ER 1994), Brisbane, Australia (ER 1995), Cottbus, Alemania (ER 1996), Singapur (ER 1998), Salt Lake City, Utah (ER 1999), Yokohama, Japón (ER 2001), Tampere, Finlandia (ER 2002), Chicago, Illinois (ER 2003), Shanghai, China (ER 2004) y Klagenfurt, Austria (ER 2005). La conferencia de 2006 tendrá lugar en Tuscon, Arizona.

CAPÍTULO 4

# El modelo Entidad-Relación mejorado (EER)

Los conceptos de modelado ER tratados en el Capítulo 3 son suficientes para representar muchos de los esquemas de bases de datos de las aplicaciones *tradicionales*. Sin embargo, desde finales de los años 70, los diseñadores de aplicaciones de base de datos han intentado crear esquemas que reflejen de un modo más preciso las propiedades y restricciones de los datos. Esto fue algo especialmente importante en las nuevas aplicaciones de tecnología de bases de datos, como las orientadas al diseño de la ingeniería y la fabricación (CAD/CAM)[1], las telecomunicaciones, los sistemas complejos de software y los GIS (Sistemas de información geográfica, *Geographic Information Systems*). Este tipo de bases de datos tienen unos requisitos más complejos que los necesarios en las aplicaciones tradicionales, lo que llevó al desarrollo de nuevos conceptos en la *semántica de modelado de datos* que se incorporaron en modelos de datos conceptuales como el ER. Se han propuesto muchos modelos de semántica de datos. Muchos de estos conceptos fueron desarrollados de forma independiente en otras áreas de la computación, como la **representación del conocimiento** en la inteligencia artificial y el **modelado de objetos** en la ingeniería de software.

En este capítulo se describen algunos de los aspectos que se han propuesto para la semántica de los modelos de datos, y muestra la manera de mejorar el modelo ER para incluir esos conceptos y obtener el modelo **EER (ER mejorado, *Enhanced ER*)**.[2] Empezaremos en la Sección 4.1 incorporando los conceptos de relación clase/subclase y de tipo de herencia en el modelo ER.

A continuación, en la Sección 4.2, se incorporarán los conceptos de especialización y generalización.

La Sección 4.3 trata los distintos tipos de restricciones en la especialización/generalización, mientras que la Sección 4.4 muestra la forma de modificar la construcción UNION para incluir el concepto de categoría en el modelo EER. La Sección 4.5 muestra un ejemplo del esquema de bases de datos UNIVERSIDAD en el modelo EER y resume los conceptos de este modelo a través de definiciones formales.

En la Sección 4.6 presentamos la notación del diagrama de la clase UML para la representación de la especialización y la generalización, comparándose brevemente con la notación EER y sus conceptos. Todo esto sirve como ejemplo de notación alternativa, y representa una continuación de la Sección 3.8, en la que se presentó la notación del diagrama de la clase UML básica. En la Sección 4.7 explicamos las abstracciones

[1] CAD/CAM son las siglas de Diseño asistido por computador/Fabricación asistida por computador (*Computer-Aided Design/Computer-Aided Manufacturing*).

[2] EER también se ha utilizado como siglas para el modelo ER extendido, *Extended ER*.

fundamentales que se emplean como base de muchos de los modelos semánticos de datos. Por último, la Sección 4.8 resume todo el capítulo.

Para entrar en más detalle en el modelado conceptual, el Capítulo 4 debe considerarse como una continuación del 3. Sin embargo, si sólo necesita una breve introducción al modelado ER, puede saltarse este capítulo, aunque también puede optar por omitir algunas secciones del mismo (de la 4.4 a la 4.8).

# 4.1 Subclases, superclases y herencia

EER contiene todos los conceptos de modelado del modelo ER mostrados en el Capítulo 3, además de incluir la definición de **subclase** y **superclase** y los términos **especialización** y **generalización** (consulte las Secciones 4.2 y 4.3). Otro concepto incluido en el modelo EER es el de **categoría** o **tipo unión** (consulte la Sección 4.4), que se emplea para representar una colección de objetos que es la unión de los objetos de distintos tipos de entidades. Junto a todos estos términos se encuentra el importante mecanismo de **atributo y relación de herencia**. Por desgracia, no existe una terminología estándar para estos conceptos, por lo que utilizaremos la más común (las notas al pie contienen definiciones alternativas). También se describe una técnica de diagramación para mostrar todos estos conceptos cuando confluyen en un esquema EER. Estos esquemas reciben el nombre de **diagramas EER** o **ER mejorado**.

El primer concepto EER que vamos a tratar es el de una **subclase** de un tipo de entidad. Como ya se comentó en el Capítulo 3, un tipo de entidad se emplea para representar tanto a un *tipo de entidad* como al *conjunto de entidades o colección de entidades de ese tipo* que existen en la base de datos. Por ejemplo, el tipo de entidad EMPLEADO describe el tipo (es decir, los atributos y las relaciones) de cada empleado, y hace también referencia al conjunto actual de entidades EMPLEADO de la base de datos EMPRESA. En muchos casos, un tipo de entidad cuenta con varios subgrupos de entidades que son significativos y deben ser representados de forma explícita debido a su importancia en la aplicación de base de datos. Por ejemplo, las entidades que forman parte de EMPLEADO pueden agruparse en SECRETARIA, INGENIERO, DIRECTIVO, TÉCNICO, PERSONAL_FIJO, PERSONAL_TEMPORAL, etc. El conjunto de entidades de cada uno de estos grupos es un subconjunto que pertenece a EMPLEADO, lo que implica que cada una de estas entidades es también un empleado. Podemos decir que cada una de estas agrupaciones es una subclase de la entidad EMPLEADO, mientras que ésta última es la superclase de todas las demás. La Figura 4.1 muestra el modo de representar estos conceptos en forma de diagramas EER (el significado del círculo se explicará en la Sección 4.2).

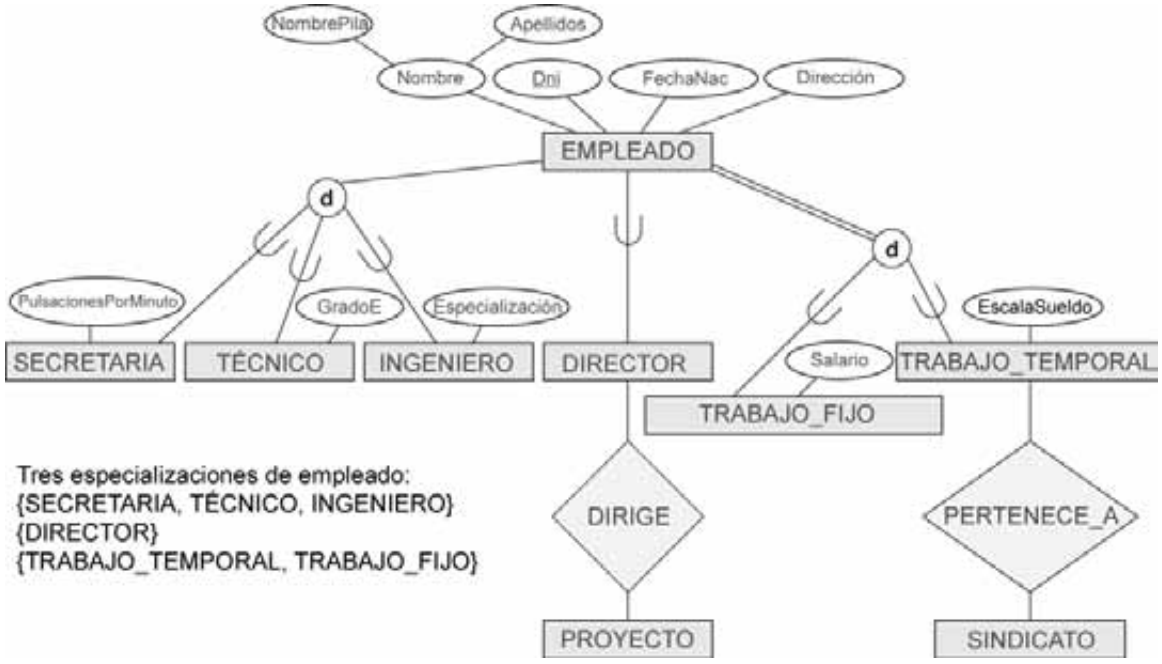
La relación entre una superclase y una de sus subclases recibe el nombre de **superclase/subclase**, o simplemente **relación clase/subclase**.[3] Volviendo a nuestro anterior ejemplo, EMPLEADO/SECRETARIA y EMPLEADO/TÉCNICO son dos relaciones clase/subclase. Observe que una entidad miembro de la subclase representa a la *misma entidad del mundo real* en la superclase; por ejemplo, la entidad SECRETARIA 'Laura Logano' es también el EMPLEADO 'Laura Logano'. Por tanto, el miembro de la subclase es el mismo que la entidad en la superclase, pero en un *papel específico* distinto. Sin embargo, cuando se implementa una relación superclase/subclase en el sistema de bases de datos, podemos representar a un miembro de la subclase como un objeto de base de datos distinto; digamos, un registro diferente que está relacionado a través del atributo clave con su entidad superclase. En la Sección 7.2 se ofrecen varias opciones para representar la relación superclase/subclase en bases de datos relaciones.

Una entidad no puede existir en una base de datos siendo sólo miembro de una subclase; también debe pertenecer a una superclase. Del mismo modo, una entidad puede estar incluida opcionalmente en varias subclases. Por ejemplo, un empleado fijo que también es un ingeniero pertenecerá a las subclases INGENIERO y PERSONAL_FIJO de la entidad EMPLEADO. Sin embargo, no es necesario que cada entidad de una superclase sea miembro de alguna subclase.

---

[3] Una relación clase/subclase suele recibir el nombre de **relación ES-UN/ES-UNA** debido al modo de hacer referencia al concepto. Decimos que una SECRETARIA *es un* EMPLEADO, un TÉCNICO *es un* EMPLEADO, etc.

**Figura 4.1.** Diagrama EER que representa las subclases y la especialización.




Un concepto importante asociado a las subclases es el de **tipo de herencia**. Recuerde que el *tipo* de una entidad está definido por los atributos que posee y los tipos de relación en los que participa. Ya que una entidad en una subclase representa también a la misma persona en la superclase, debe disponer de valores para sus atributos específicos, *así como* otros como miembro de la superclase. Decimos, por tanto, que una entidad que es miembro de una subclase **hereda** todos los atributos de la entidad como miembro de la superclase y las relaciones en las que ésta participa. Observe que una subclase, con sus atributos y relaciones propias (o locales) junto con los que hereda de la superclase, puede ser considerada como un *tipo de entidad* por derecho propio.[4]

# 4.2 Especialización y generalización

## 4.2.1 Especialización

La **especialización** es el proceso de definir un *conjunto de subclases* de un tipo de entidad, la cual recibe el nombre de **superclase** de la especialización. El conjunto de subclases que forman una especialización se define basándose en algunas características distintivas de las entidades en la superclase. Por ejemplo, las subclases {SECRETARIA, INGENIERO, TÉCNICO} son una especialización de la superclase EMPLEADO que distingue a los trabajadores en función al *tipo de trabajo* que desempeñan. Podemos tener varias especializaciones del mismo tipo de entidad en función de varias características distintivas. Por ejemplo, otra especialización de EMPLEADO podría ser el conjunto de subclases {PERSONAL_FIJO, PERSONAL_TEMPORAL}, las cuales distinguen a los trabajadores por el *tipo de contrato que tienen.*

[4] En algunos lenguajes de programación orientados a objetos, existe una restricción común que dice que una entidad (u objeto) *sólo tiene un tipo*. En general, este planteamiento es demasiado restrictivo para cualquier modelo de base de datos.

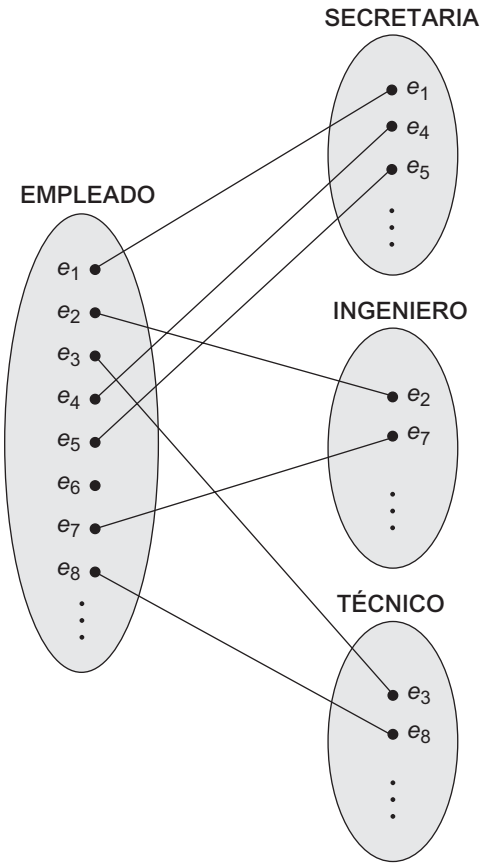
La Figura 4.1 muestra la forma de representar una especialización en un diagrama EER. Las subclases que la definen están unidas por líneas a un círculo que representa la especialización, la cual está a su vez unida a la superclase.

El *símbolo de subconjunto* de cada línea que conecta una subclase al círculo indica la dirección de la relación superclase/subclase.[5] Los atributos que se aplican únicamente a las entidades de una subclase particular (como las PulsacionesPorMinuto de una SECRETARIA) están unidas al rectángulo que la representa, y reciben el nombre de **atributos específicos** (o **atributos locales**) de la subclase. Además, una subclase puede participar de **tipos de relación específicos**, como ocurre en la Figura 4.1 con la subclase PERSONAL_TEMPORAL que participa de la relación PERTENECE_A. El significado del símbolo **d** que aparece en los círculos de dicha figura, y en otros diagramas EER, se explicará a continuación.

La Figura 4.2 muestra algunas instancias que pertenecen a las subclases de la especialización {SECRETARIA, INGENIERO, TÉCNICO}. Observe de nuevo que una entidad que pertenece a una subclase representa a la misma persona que la entidad conectada a ella en la superclase EMPLEADO, aun cuando se muestre dos veces; por ejemplo, *e1* aparece en la Figura 4.2 como EMPLEADO y SECRETARIA.

Tal y como sugiere la figura, una relación superclase/subclase como EMPLEADO/SECRETARIA se asemeja en cierto modo a otra del tipo 1:1 a *nivel de instancia* (véase la Figura 3.12). La diferencia principal es que

**Figura 4.2.** Instancias de una especialización.




[5] Existen notaciones alternativas para las especializaciones; mostraremos la UML en la Sección 4.6 y otros tipos en el Apéndice A.

en una relación 1:1, dos *entidades distintas* están relacionadas, mientras que en el caso de una superclase/subclase, la entidad de la subclase es la misma que la de la superclase pero desempeñando un *papel concreto* (por ejemplo, un EMPLEADO especializado en el papel de SECRETARIA o de TÉCNICO).

Existen dos razones principales para incluir relaciones clase/subclase y especializaciones en un modelo de datos. La primera es que pueden existir ciertos atributos que sólo deban aplicarse a algunas entidades, pero no a toda la superclase. Por tanto, la subclase se define para agrupar a todas esas entidades. Los integrantes de la subclase pueden seguir compartiendo la mayor parte de sus atributos con los otros miembros de la superclase. Por ejemplo, en la Figura 4.1, la subclase SECRETARIA cuenta con el atributo específico PulsacionesPorMinuto, mientras que INGENIERO dispone de otro llamado Especialización, aunque ambos comparten los atributos heredados de la entidad EMPLEADO.

El segundo motivo para usar subclases es que algunos tipos de relaciones sólo pueden establecerse entre miembros de esa subclase. Por ejemplo, si sólo el PERSONAL_TEMPORAL puede estar afiliado a un sindicato, podemos representar esta circunstancia creando la subclase PERSONAL_TEMPORAL de EMPLEADO y relacionarla con una entidad SINDICATO a través de la relación PERTENECE_A, tal y como puede verse en la Figura 4.1.

En resumen, el proceso de especialización no permite hacer lo siguiente:

- Definir un conjunto de subclases de un tipo de entidad.
- Establecer atributos específicos adicionales en cada subclase.
- Establecer relaciones específicas adicionales entre cada subclase y otras entidades u otras subclases.

### 4.2.2 Generalización

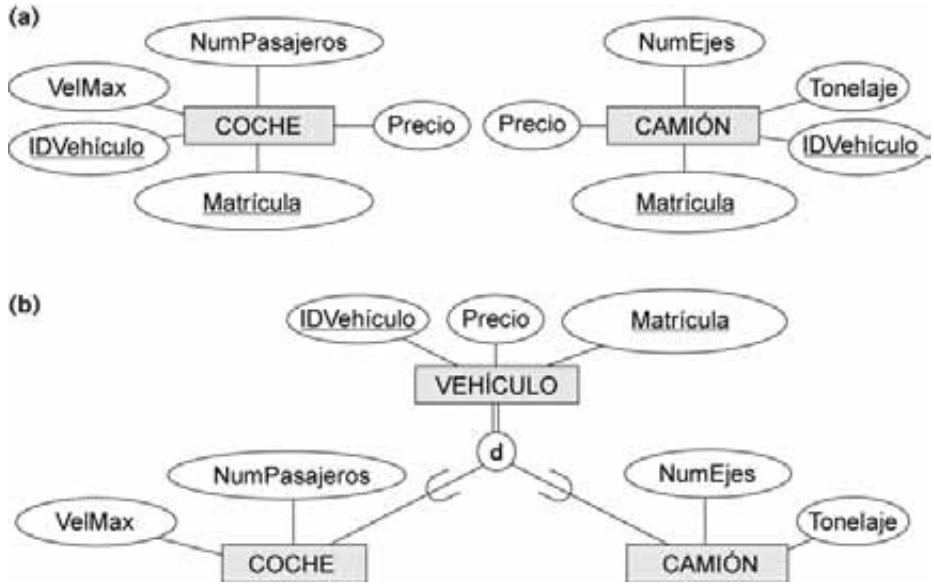
Podemos pensar en un *proceso inverso* de abstracción en el que eliminemos las diferencias existentes entre distintas entidades, identifiquemos las características comunes y las **generalicemos** en una única **superclase** de la que las entidades originales sean **subclases** especiales. Por ejemplo, consideremos las entidades COCHE y CAMIÓN mostradas en la Figura 4.3(a). Ya que ambas cuentan con características comunes, podrían generalizarse en la entidad VEHÍCULO (véase la Figura 4.3[b]). Tanto COCHE como CAMIÓN son ahora subclases de la **superclase generalizada** VEHÍCULO. Usamos el término **generalización** para referirnos al proceso por el cual se define una entidad generalizada a partir de entidades individuales.

Observe que la generalización puede considerarse como el proceso inverso de la especialización desde un punto de vista funcional. Por tanto, en la Figura 4.3 podemos decir que {COCHE, CAMIÓN} son una especialización VEHÍCULO, en lugar de considerar a VEHÍCULO como una generalización de COCHE y CAMIÓN. De forma análoga, en la Figura 4.1 podemos ver a EMPLEADO como una generalización de SECRETARIA, TÉCNICO e INGENIERO. En algunas metodologías de diagramación existen distintos elementos para distinguir una generalización de una especialización. Una flecha que apunta a la superclase generalizada representa una generalización, mientras que cuando lo hace hacia las subclases especializadas indica una especialización.

No utilizaremos esta notación ya que la decisión sobre qué proceso es más apropiado en una situación suele ser algo bastante subjetivo. El Apéndice A muestra algunas alternativas para la representación de diagramas de esquema y de diagramas de clase.

Hasta ahora hemos visto los conceptos de subclase y de relación superclase/subclase, así como los procesos de especialización y generalización. En general, una superclase o una subclase representan una colección de entidades del mismo tipo que, por consiguiente, también describe un *tipo de entidad*; ésta es la razón por la que ambos elementos aparecen como rectángulos en los diagramas EER. A continuación, vamos a tratar con más detalle las propiedades de las especializaciones y las generalizaciones.

**Figura 4.3.** Generalización. (a) Dos entidades, COCHE y CAMIÓN. (b) Generalizando COCHE y CAMIÓN en la superclase VEHÍCULO.




# 4.3 Restricciones y características de las jerarquías de especialización y generalización

En primer lugar, trataremos las restricciones que se aplican a una única especialización o generalización. Por brevedad, la explicación sólo hará referencia a la *especialización* aun cuando ésta se aplique a ambos términos. A continuación, nos centraremos en las diferencias existentes entre los *entramados* (*lattices*) (*herencia múltiple*) y las *jerarquías* (*herencia sencilla*) de especialización/generalización, y elaboraremos las principales diferencias existentes entre ambos procesos durante el diseño de un esquema de base de datos conceptual.

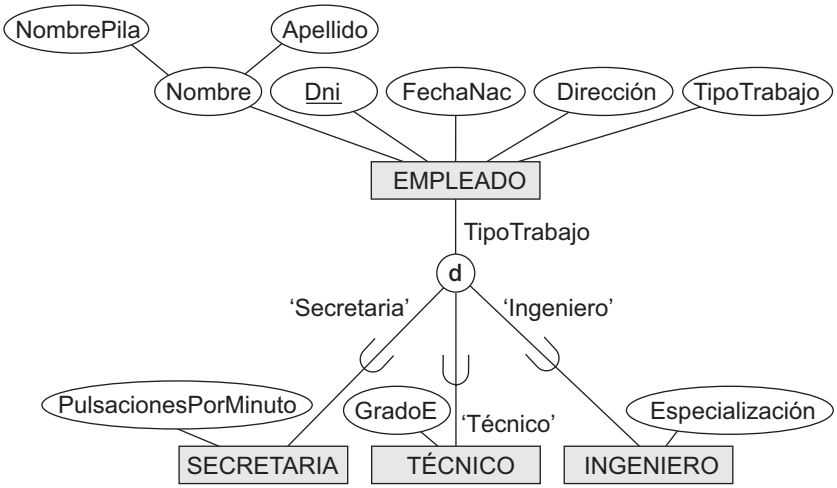
## 4.3.1 Restricciones en la especialización y la generalización

En general, podemos contar con varias especializaciones definidas en la misma entidad (o superclase), como puede verse en la Figura 4.1. En este caso, las entidades pueden pertenecer a las subclases de cada una de las especializaciones. Sin embargo, una especialización también puede contar con una *única* subclase, como ocurre en el caso de {DIRECTOR} de la Figura 4.1; en esta situación, no utilizamos el círculo correspondiente.

En algunas especializaciones, podemos determinar con exactitud las entidades que se convertirán en miembros de cada subclase situando una condición en el valor de algunos atributos de la superclase. Estas subclases reciben el nombre de **subclases** de **predicado definido** (o de **condición definida**). Por ejemplo, si la entidad EMPLEADO cuenta con un atributo TipoTrabajo, tal y como puede verse en la Figura 4.4, podemos especificar la pertenencia a la subclase SECRETARIA mediante la condición (TipoTrabajo = 'Secretaria'), es decir, **definiendo el predicado** de la subclase. Esta condición es una restricción específica que nos permite decir que aquellas entidades de EMPLEADO cuyo valor para el atributo TipoTrabajo sea 'Secretaria' pertenecen a esa subclase. Identificamos una subclase de predicado definido escribiendo la condición a continuación de la línea que conecta la subclase al círculo de especialización.

Si *todas* las subclases de una especialización tienen su condición de pertenencia en el *mismo* atributo de la superclase, la propia especialización recibe el nombre de **especialización de atributo definido**, y el atributo

**Figura 4.4.** Indicación en un diagrama EER de una especialización de atributo definido para TipoTrabajo.




recibe el nombre de **atributo definitorio** de la especialización.[6] Identificamos una especialización de atributo definido colocando el nombre del mismo al lado del arco que va desde el círculo a la superclase, tal y como puede verse en la Figura 4.4.

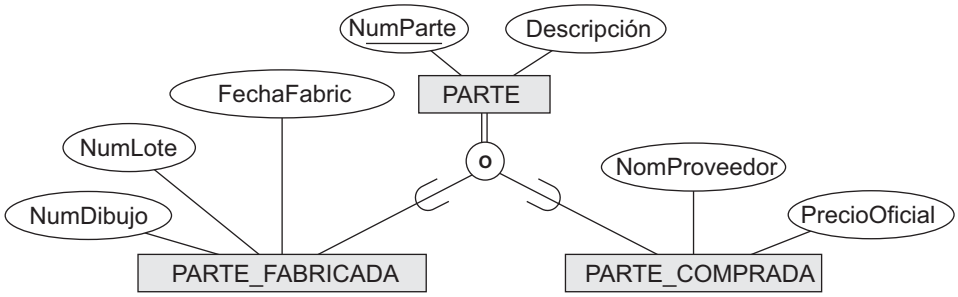
Cuando no tenemos una condición para determinar los miembros de una subclase, se dice que es de tipo **definido por usuario**. Los miembros de este tipo de subclase son determinados por los usuarios de la base de datos cuando aplican la operación para añadir una entidad a la subclase; así pues, los miembros son *especificados individualmente por el usuario para cada entidad*, y no por una condición que pueda evaluarse automáticamente.

Existen otras dos restricciones que pueden aplicarse a una especialización. La primera es la **restricción de disyunción** (***disjointness***), la cual especifica que las subclases de la especialización deben estar separadas. Esto significa que una entidad puede ser, como máximo, miembro de una de las subclases de la especialización. Una especialización de tipo atributo-definido implica la restricción de disyunción en el caso de que el atributo utilizado para definir el predicado de agrupación sea de un solo valor, o monovalor. La Figura 4.4 ilustra este caso, donde la **d** incluida en el círculo simboliza la *separación*. También se utiliza este símbolo para especificar que las subclases de una especialización definidas por el usuario deben ser del mismo tipo, tal y como puede verse en el ejemplo {PERSONAL_FIJO, PERSONAL_TEMPORAL} de la Figura 4.1. Si las subclases no están obligadas a estar separadas, su conjunto de entidades pueden **solaparse**, es decir, la misma entidad podría ser miembro de más de una subclase de la especialización. Este caso, que es el que se produce por defecto, se especifica colocando una **o** en el círculo, tal y como puede verse en la Figura 4.5.

La segunda restricción de una especialización se conoce como **restricción de integridad**, la cual puede ser total o parcial. Una **especialización total** especifica que *cada* entidad en la superclase debe ser miembro de, al menos, una subclase en la especialización. Por ejemplo, si cada EMPLEADO debe ser PERSONAL_FIJO o PERSONAL_TEMPORAL, entonces la especialización {PERSONAL_FIJO, PERSONAL_TEMPORAL} de la Figura 4.1 es una especialización total de EMPLEADO. Esto se muestra en los diagramas EER usando una línea doble que conecta la superclase al círculo. Para mostrar una **especialización parcial** se emplea una línea sencilla, lo que permite que una entidad no pertenezca a ninguna de las subclases. Por

[6] En terminología UML, este tipo de atributo se conoce como *discriminador*.

**Figura 4.5.** Indicación de una especialización de solapamiento (*nondisjoint*) en un diagrama EER.




ejemplo, si alguna entidad EMPLEADO no está incluida en ninguna de las subclases {SECRETARIA, INGENIERO, TÉCNICO} de las Figuras 4.1 y 4.4, entonces esa especialización es parcial.[7]

Observe que las restricciones de disyunción y de integridad son *independientes.* Por consiguiente, son posibles las cuatro siguientes restricciones en una especialización:

- Disyunción, total.
- Disyunción, parcial.
- Solapamiento, total.
- Solapamiento, parcial.

Desde luego, la restricción correcta viene determinada por la aplicación real que se le quiera dar a cada especialización. Por lo general, una superclase identificada a través del proceso de *generalización* es **total**, ya que está *derivada a partir* de las subclases y, por consiguiente, sólo contiene las entidades incluidas en ellas.

Existen ciertas reglas de inserción y borrado que se aplican a una especialización (y una generalización) como consecuencia de las restricciones indicadas anteriormente. Éstas son algunas de esas reglas:

- El borrado de una entidad de una superclase implica su eliminación automática de todas las subclases a las que pertenece.
- La inserción de una entidad en una superclase supone que la misma debe insertarse en todas las subclases de *predicado definido* (o *atributo definido*) en las que esa entidad cumpla la regla.
- La inserción de una entidad en una superclase de *especialización total* conlleva que dicha entidad sea incluida obligatoriamente en, al menos, una de las subclases de la especialización.

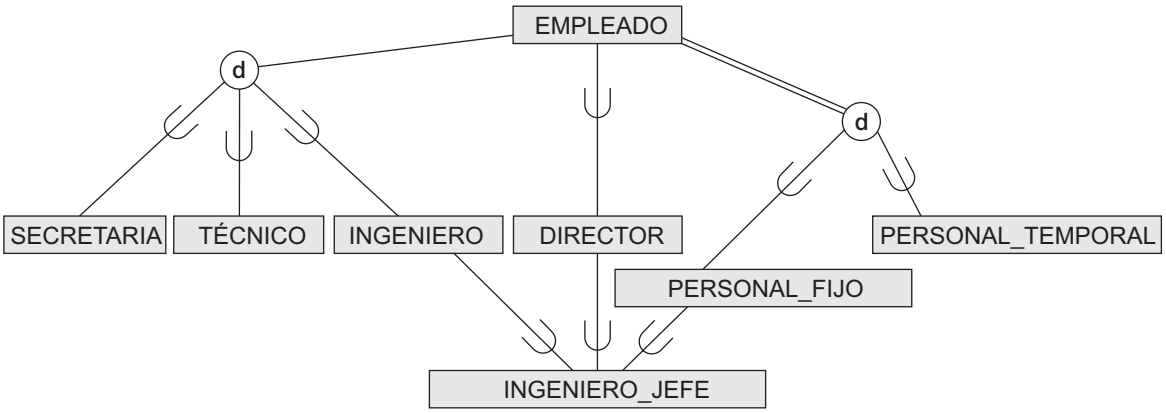
Se exhorta al lector a realizar una lista completa de reglas de inserción y borrado para los distintos tipos de especialización.

### 4.3.2 Jerarquías y entramados de especialización y generalización

Una subclase, por si misma, puede tener más subclases definidas en ella formando una jerarquía (o entramado) de especializaciones. Por ejemplo, en la Figura 4.6, INGENIERO es una subclase de EMPLEADO y, a su vez, una superclase de INGENIERO_JEFE; esto representa una restricción real que dice que todo ingeniero jefe debe ser ingeniero. Una **especialización jerárquica** tiene una restricción que dice que cada subclase participa como tal en *una única* relación clase/subclase, es decir, cada subclase sólo tiene un padre, lo que deriva en la formación de una estructura en árbol. En contraposición, en una **especialización entramada**, una subclase puede serlo en *más de una* relación clase/subclase. Por tanto, la Figura 4.6 es un entramado.

[7] El uso de líneas dobles o sencillas es similar a lo que sucede con la participación parcial o total de un tipo de entidad en un tipo de relación, tal y como se describió en el Capítulo 3.

**Figura 4.6.** Un entramado de especialización con una subclase INGENIERO_JEFE compartida.




La Figura 4.7 muestra otro entramado de especialización de más de un nivel. Esto puede formar parte del esquema conceptual de una base de datos UNIVERSIDAD. Observe que esta disposición podría ser también una jerarquía excepto por la subclase ADJUNTO_ENSEÑANZA, la cual forma parte de dos relaciones clase/subclase diferentes.

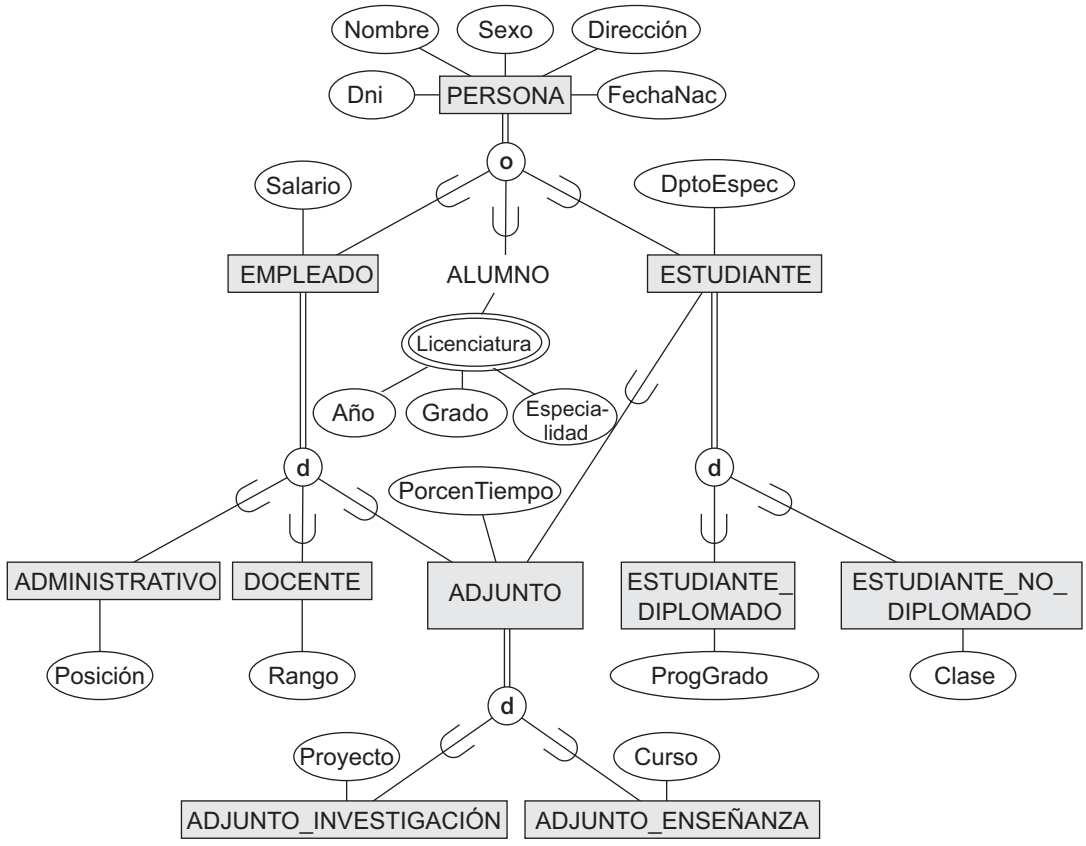
Los requisitos de la base de datos UNIVERSIDAD de la Figura 4.7 son los siguientes:

1. La base de datos mantiene tres tipos de personas: empleados, ex alumnos y estudiantes. Una persona puede pertenecer a uno, dos o los tres tipos, y dispone de un nombre, un DNI, su sexo, su dirección y fecha de nacimiento.
2. Cada empleado dispone de un salario, y se agrupan en tres categorías distintas: personal docente, administrativos y adjuntos. Cada trabajador pertenece a uno de los tres grupos. Para los antiguos alumnos, se mantiene un registro con la titulación máxima conseguida, incluyendo el nombre de dicha titulación, el año de obtención y la especialidad.
3. Cada profesor tiene un rango, mientras que el personal administrativo cuenta con una posición. Los adjuntos están clasificados como asistentes de investigación o de enseñanza, registrándose en la base de datos el porcentaje de tiempo que trabajan, así como sus proyectos de investigación (los primeros) o el curso que imparten (los segundos).
4. Los estudiantes están clasificados como diplomados o como estudiantes propiamente dichos, con los atributos específicos del programa de grado (M.S., Ph.D., M.B.A., etc.) o clase (estudiante de primer año, estudiante de segundo año, etc.), respectivamente.

En la Figura 4.7, todas las personas representadas en la base de datos son miembros de la entidad PERSONA, la cual está especializada en las subclases {EMPLEADO, EX_ALUMNO, ESTUDIANTE}. Esta especialización está solapada; por ejemplo, un ex alumno puede ser también un empleado o un estudiante cursando un nivel más avanzado. La subclase ESTUDIANTE es la superclase de la especialización {ESTUDIANTE_DIPLOMADO, ESTUDIANTE_NO_DIPLOMADO}, mientras que EMPLEADO lo es de {ADJUNTO, DOCENTE, ADMINISTRATIVO}. Observe que ADJUNTO es también una subclase de ESTUDIANTE. Por último, ADJUNTO es la superclase de la especialización {ADJUNTO_INVESTIGACIÓN, ADJUNTO_ENSEÑANZA}.

Independientemente de si se trata de una jerarquía o un entramado de especialización, una subclase hereda los atributos no sólo de su superclase directa, sino también de todas sus superclases predecesoras hasta el tope de la jerarquía o el entramado. Por ejemplo, una entidad en ESTUDIANTE_DIPLOMADO hereda todos los atributos de ESTUDIANTE y PERSONA. Observe que una entidad puede existir en varios *nodos hoja* de la jerarquía, donde un **nodo hoja** es una clase que *no tiene subclases.* Por ejemplo, un miembro de ESTUDIANTE_DIPLOMADO puede serlo también de ADJUNTO_INVESTIGACIÓN.

**Figura 4.7.** Un entramado de especialización con herencia múltiple para una base de datos UNIVERSIDAD.




Una subclase que cuente con *más de una* superclase recibe el nombre de **subclase compartida**, como ocurre con INGENIERO_JEFE en la Figura 4.6. Esto nos lleva al concepto conocido como **herencia múltiple**, en donde la subclase compartida INGENIERO_JEFE hereda directamente atributos y relaciones de varias clases. Observe que la existencia de al menos una subclase compartida nos lleva a un entramado (y, por consiguiente, a una *herencia múltiple*); si no existieran subclases compartidas, tendríamos una jerarquía en lugar de un entramado. La subclase compartida ADJUNTO_ENSEÑANZA de la Figura 4.7, que hereda atributos tanto de EMPLEADO como de ESTUDIANTE, puede ilustrar una regla importante relacionada con la herencia múltiple. En este caso, EMPLEADO y ESTUDIANTE heredan los *mismos atributos* de PERSONA. La regla establece que si un atributo (o relación) que se origina en la *misma superclase* (PERSONA) es heredado más de una vez a través de caminos diferentes (EMPLEADO y ESTUDIANTE) en el entramado, sólo podrá incluirse una vez en la subclase compartida (ADJUNTO_ENSEÑANZA). Por tanto, los atributos de PERSONA sólo se heredan una vez en la subclase ADJUNTO_ENSEÑANZA de la Figura 4.7.

Es importante indicar que algunos modelos y lenguajes *no permiten* la herencia múltiple (subclases compartidas). En un modelo de este tipo es necesario crear subclases adicionales que cubran todas las posibles combinaciones de clases en las que una entidad pertenezca simultáneamente a todas ellas. Por tanto, cualquier especialización de solapamiento precisará de múltiples subclases adicionales. Por ejemplo, el solapamiento de PERSONA en {EMPLEADO, EX_ALUMNO, ESTUDIANTE} (o {E, A, S} para abreviar, según el inglés), precisaría de la creación de siete subclases de PERSONA que cubrieran todos los posibles tipos de entidades: E, A, S, E_A, E_S, A_S, y E_A_S. Obviamente, esto conlleva una complejidad añadida.

Es importante indicar también que algunos mecanismos de herencia que permiten la herencia múltiple no dejan que una entidad tenga varios tipos, lo que implica que sólo puede ser miembro de *una única clase*.[8] En modelos de este tipo, es necesario crear también subclases compartidas adicionales como nodos hoja que abarquen todas las posibles combinaciones de clases.

Aunque hemos empleado la especialización para explicar el tema, se pueden aplicar los mismos conceptos en la generalización, por lo que podríamos hablar de **jerarquía de generalización** y de **entramado de generalización**.

### 4.3.3 Utilización de la especialización y la generalización en el refinamiento de los esquemas conceptuales

Ahora profundizaremos en las diferencias entre los procesos de especialización y de generalización, y en cómo deben usarse para refinar esquemas conceptuales durante el diseño de bases de datos conceptuales. En el proceso de especialización, partimos de una entidad para definir a continuación subclases de la misma a través de especializaciones sucesivas, esto es, definimos de forma repetida agrupaciones más específicas de la entidad principal. Por ejemplo, cuando se diseña el entramado de especialización de la Figura 4.7, podríamos empezar especificando una entidad PERSONA en una base de datos universitaria. A continuación, descubrimos que existirán tres tipos de personas representados en dicha base de datos: trabajadores, ex alumnos y estudiantes. Para ello, creamos la especialización {EMPLEADO, EX_ALUMNO, ESTUDIANTE} y elegimos la restricción de solapamiento porque una persona puede pertenecer a más de una de estas subclases. Más adelante, especializamos EMPLEADO en {ADMINISTRATIVO, DOCENTE, ADJUNTO} y ESTUDIANTE en {ESTUDIANTE_DIPLOMADO, ESTUDIANTE_NO_DIPLOMADO}. Por último, ADJUNTO se especializa en {ADJUNTO_INVESTIGACIÓN, ADJUNTO_ENSEÑANZA}. Esta diversificación progresiva se corresponde con un **proceso de refinamiento conceptual de arriba abajo**.

Hasta aquí, tenemos una jerarquía; ahora nos damos cuenta de que ADJUNTO_ENSEÑANZA es una subclase compartida, ya que también es una subclase de ESTUDIANTE, lo que hace que tengamos un entramado.

Es posible llegar a la misma jerarquía o entramado desde otra dirección. En un caso como éste, el proceso implica llevar a cabo una generalización en lugar de una especialización y se corresponde con una **síntesis conceptual de abajo arriba**. Esta situación obliga, en primer lugar, a que los diseñadores descubran entidades del tipo ADMINISTRATIVO, DOCENTE, EX_ALUMNO, ESTUDIANTE_DIPLOMADO, ESTUDIANTE_NO_DIPLOMADO, ADJUNTO_INVESTIGACIÓN, ADJUNTO_ENSEÑANZA, etc.; a continuación, deben generalizar {DIPLOMADO, ESTUDIANTE_NO_DIPLOMADO} a ESTUDIANTE; {ADJUNTO_INVESTIGACIÓN, ADJUNTO_ENSEÑANZA} a ADJUNTO; {ADMINISTRATIVO, DOCENTE, ADJUNTO} a EMPLEADO; y, por último, {EMPLEADO, EX_ALUMNO, ESTUDIANTE} a PERSONA.

En términos estructurales, las jerarquías o entramados resultantes de este proceso pueden ser idénticas; la única diferencia radica en la manera, o el orden, en el que se especifica el esquema de superclases y subclases. En la práctica, es muy probable que no siga de manera estricta ni el proceso de generalización ni el de especialización, sino que se emplee una combinación de ambos. En este caso, existe una continua incorporación de nuevas clases a la jerarquía o el entramado a medida que el proceso se hace más claro a los usuarios y los diseñadores. Tenga en cuenta que la noción de representación de datos y conocimiento usando jerarquías o entramados de superclase/subclase es muy común en sistemas del conocimiento y sistemas expertos, los cuales combinan tecnología de bases de datos con técnicas de inteligencia artificial. Por ejemplo, los esquemas de representación del conocimiento basados en *frames* tienen un gran parecido con las jerarquías de clase. La especialización es también muy común en las metodologías de diseño de ingeniería de software basadas en el paradigma de la orientación a objetos.

---

[8] En algunos modelos, la clase está obligada a ser un *nodo hoja* en la jerarquía o el entramado.

## 4.4 Modelado de tipos UNION usando categorías

Todas las relaciones superclase/subclase que hemos visto hasta ahora tenían una *sola superclase.* Una subclase compartida como INGENIERO_JEFE en el entramado de la Figura 4.6 es la subclase en tres relaciones superclase/subclase *distintas*, donde cada una de estas tres relaciones cuenta con una *única* superclase. Sin embargo, no resulta extraño que la necesidad obligue a modelar una única relación superclase/subclase con *más de una* superclase, donde esas superclases representen diferentes tipos de entidades. En este caso, la subclase representará una colección de objetos que es un subconjunto de la UNION de distintos tipos de entidades; llamaremos a esta *subclase* un **tipo unión** o una **categoría**.[9]

Por ejemplo, supongamos que tenemos tres entidades: PERSONA, BANCO y EMPRESA. En una base de datos de registro de vehículos, el propietario de uno de ellos puede ser una persona física, un banco (que lo haya adquirido mediante un embargo) o una empresa. Para desempeñar el papel de *propietario de un vehículo*, necesitamos crear una clase (colección de entidades) que incluya estos tres tipos. Para ello, se crea una categoría PROPIETARIO que sea una *subclase de la UNION* de los tres conjuntos de entidades. En la Figura 4.8 puede verse el modo de mostrar las categorías en un diagrama EER. Las superclases EMPRESA, BANCO y PERSONA se conectan al círculo mediante un punto (.). Un arco con el símbolo de subconjunto conecta el círculo a la categoría (subclase) PROPIETARIO.

En caso de necesitarse un predicado, éste aparece a continuación de la línea que sale de la superclase a la cual debe aplicarse. En la Figura 4.8 tenemos dos categorías: PROPIETARIO, que es una subclase de la unión de PERSONA, BANCO y EMPRESA, y VEHÍCULO_REGISTRADO, que es una subclase de la unión de COCHE y CAMIÓN.

Una categoría tiene dos o más superclases que pueden representar *distintos tipos de entidades,* mientras que las otras relaciones superclase/subclase siempre tienen una sola superclase. Podemos comparar una categoría, como PROPIETARIO en la Figura 4.8, con la subclase compartida INGENIERO_JEFE de la Figura 4.6. El resultado es una subclase de *cada una* de las tres superclases INGENIERO, JEFE y PERSONAL_FIJO, por lo que una entidad que fuera miembro de INGENIERO_JEFE debería existir en las *tres.* Esto supone la aplicación de la restricción de que un ingeniero jefe debe ser un INGENIERO, un JEFE y PERSONAL_FIJO, es decir, INGENIERO_JEFE es un subconjunto de la *intersección* de las tres subclases (conjuntos de entidades). En el otro extremo, una categoría es un subconjunto de la *unión* de sus superclases. Por tanto, una entidad miembro de PROPIETARIO debe aparecer *sólo* en una de las superclases. Esto supone la aplicación de la restricción de que un PROPIETARIO puede ser una EMPRESA, un BANCO o una PERSONA en la Figura 4.8.

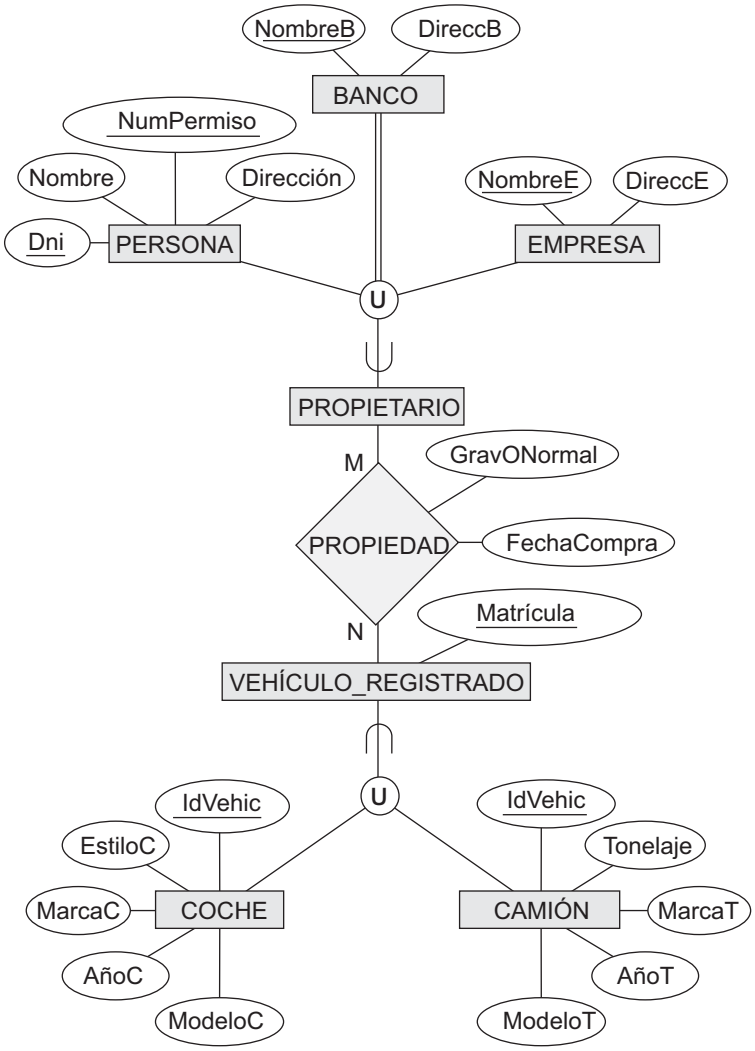
La herencia de atributo funciona de manera más selectiva en el caso de las categorías. Por ejemplo, en la Figura 4.8, cada entidad PROPIETARIO hereda los atributos de una EMPRESA, una PERSONA o un BANCO dependiendo de la superclase a la que pertenezca esa entidad. Por otro lado, una subclase compartida como INGENIERO_JEFE (Figura 4.6) hereda *todos* los atributos de sus superclases PERSONAL_FIJO, INGENIERO y JEFE.

Es interesante observar la diferencia existente entre la categoría VEHÍCULO_REGISTRADO (Figura 4.8) y la superclase generalizada VEHÍCULO (Figura 4.3[b]). En dicha figura, cada coche y cada camión es un VEHÍCULO; sin embargo, en la Figura 4.8, la categoría VEHÍCULO_REGISTRADO sólo incluye algunos coches y algunos camiones, pero no necesariamente todos ellos (por ejemplo, algunos de estos vehículos pueden no estar registrados). En general, una especialización o una generalización como la de la Figura 4.3(b), si fuera *parcial*, podrían no evitar que VEHÍCULO contuviera otros tipos de entidades, como motocicletas. Sin embargo, una categoría como VEHÍCULO_REGISTRADO de la Figura 4.8 implica que sólo coches y camiones, pero no otro tipo de entidad, pueden ser miembros de ella.

---

[9] Nuestro uso del término *categoría* está basado en el modelo ECR (Relación Entidad-Categoría, *Entity-Category-Relationship*) (Elmasri y otros, 1985).

**Figura 4.8.** Dos categorías (tipo unión): PROPIETARIO y VEHÍCULO_REGISTRADO.




Una categoría puede ser **total** o **parcial**. Una categoría total contiene la *unión* de todas las entidades de sus superclases, mientras que una parcial puede almacenar un *subconjunto de la unión*. Una categoría total está representada por una línea doble que la conecta con el círculo, en tanto que una parcial está indicada por una línea sencilla.

Las superclases de una categoría pueden tener diferentes atributos clave, como ya ha quedado demostrado en la categoría PROPIETARIO de la Figura 4.8, o tener el mismo, como ocurre con VEHÍCULO_REGISTRADO. Observe que si una categoría es total (no parcial), puede estar representada alternativamente como una especialización total (o una generalización total). En este caso, la elección del tipo de representación a usar es algo subjetivo. Si las dos clases representan al mismo tipo de entidades y comparten numerosos atributos, incluyendo los clave, es preferible la especialización/generalización; en cualquier otro caso, es más apropiado decantarse por la categorización (tipo unión).

Es interesante hacer notar que en algunos modelos (consulte el Capítulo 20), todos los objetos son especializaciones de una única clase raíz. En un modelo de este tipo, es posible modelar los tipos de unión usando la especialización apropiada.

# 4.5 Ejemplo EER de un esquema UNIVERSIDAD, diseños y definiciones formales

En esta sección empezaremos por dar un ejemplo de un esquema de base de datos en el modelo EER para ilustrar el uso de los distintos conceptos mostrados en este capítulo y en el anterior. A continuación, trataremos el tema del diseño para los esquemas conceptuales y terminaremos resumiendo los conceptos del modelo EER y los definiremos formalmente de la misma manera que hicimos con los del modelo ER en el Capítulo 3.

## 4.5.1 La base de datos UNIVERSIDAD

Para nuestro ejemplo, consideremos la base de datos UNIVERSIDAD que contiene información sobre los estudiantes y sus especialidades, traslados y registros, así como los cursos ofrecidos por la misma. Esta base de datos también almacena los proyectos de investigación patrocinados por la universidad y los estudiantes graduados. Este esquema se muestra en la Figura 4.9.

Para cada persona, la base de datos mantiene su [Nombre], [DNI], [Dirección], [Sexo] y [FechaNac]. Hay definidas dos subclases de la entidad PERSONA: PROFESOR y ESTUDIANTE. Los atributos específicos de PROFESOR son [Rango] (asistente, asociado, adjunto, investigador, etc.), [Oficina], [TlfOficina] y [Salario]. Todos los profesores están relacionados con el/los departamento/s a los que pertenecen: [PERTENECE] (un profesor puede estar asociado a varios de ellos, por lo que la relación es M:N). Un atributo específico de ESTUDIANTE es [Clase] (estudiante de primer año=1, estudiante de segundo año=2, . . . , graduado=5). Cada ESTUDIANTE está también relacionado con su especialidad y su formación secundaria, ([PRINCIPAL] y [SECUNDARIA]), de los niveles del curso en el que actualmente están matriculados [REGISTRADO], y de los cursos completados, [CERTIFICADO]. Cada instancia CERTIFICADO incluye la [Nota] que el estudiante recibe al completar ese nivel.

ESTUDIANTE_GRADUADO es una subclase de ESTUDIANTE, con el predicado definido Clase = 5. Por cada estudiante graduado se mantiene una lista de las calificaciones anteriores en un atributo multivalor compuesto llamado [Licenciatura]. También relacionamos al estudiante graduado con un [TUTOR] y su [TRIBUNAL], en caso de existir.

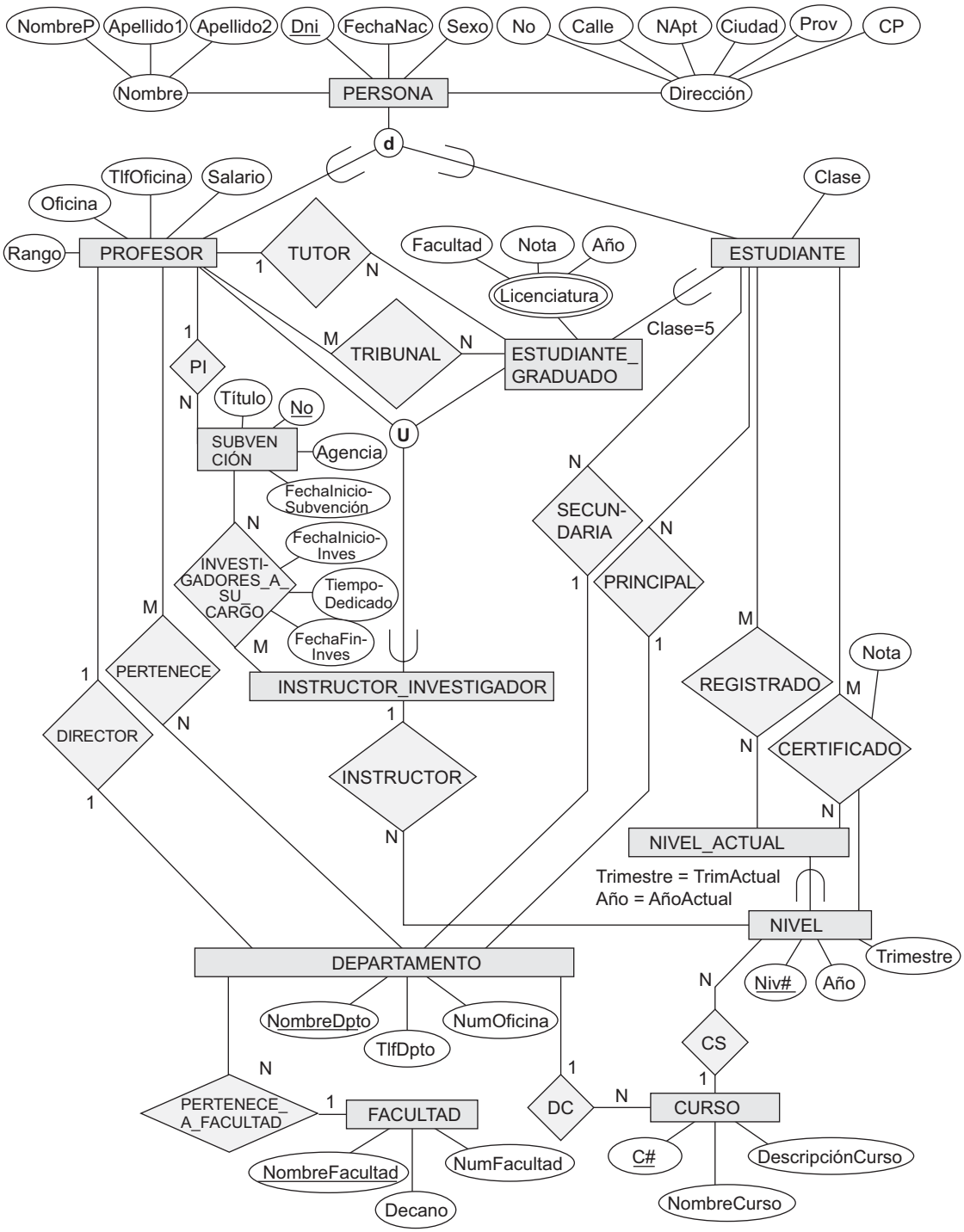
Un departamento académico cuenta con los atributos nombre [NombreDpto], teléfono [TlfDpto] y número de oficina [NumOficina] y está relacionado con la persona que lo dirige [DIRECTOR] y la facultad a la que pertenece [PERTENECE_A_FACULTAD]. Los atributos de cada facultad son su nombre [NombreFacultad], su número de oficina [NumFacultad] y su [Decano].

Cada curso cuenta con los atributos número de curso [C#], el nombre [NombreCurso] y la descripción del mismo [DescripcionCurso]. Cada curso ofrece varios niveles, por lo que cada uno de estos niveles cuenta con un número de nivel [Niv#] y el año y el trimestre en el que se ofrece ([Año] y [Trimestre]).[10] Los números de nivel los identifican de forma única, y los que están siendo ofrecidos en el trimestre actual se encuentran en la subclase NIVEL_ACTUAL de NIVEL, la cual cuenta con un predicado del tipo Trimestre = TrimActual y Año = AñoActual. Cada nivel está relacionado con el profesor que lo imparte ([INSTRUCTOR]) en caso de que se encuentre en la base de datos.

La categoría INSTRUCTOR_INVESTIGADOR es un subconjunto de la unión de PROFESOR y ESTUDIANTE_GRADUADO que incluye a todos los profesores, así como a los estudiantes graduados que están dedicados a la investigación o la enseñanza. Por último, la entidad SUBVENCIÓN contiene las subvenciones y los contratos adjudicados a la universidad. Cada subvención tiene los atributos [Título], número de subvención [No], la entidad adjudicataria [Agencia] y la fecha de inicio [FechaInicioSubvención], y está relacionado con un investigador principal [PI] y todos los [INVESTIGADORES_A_SU_CARGO]. Cada instancia de esta última tiene como atributos la fecha en la que cada investigador empieza con su tarea [FechaInicioInvestigación], la

[10] Asumimos que en esta diversidad se está usando el sistema *trimestral* en lugar del *semestral*.

**Figura 4.9.** Un esquema EER conceptual para la base de datos UNIVERSIDAD.




de finalización (en caso de conocerse) [FechaFinInvestigación] y el porcentaje de tiempo que dedica al proyecto [TiempoDedicado].

### 4.5.2 Consideraciones de diseño para la especialización/generalización

No siempre resulta sencillo elegir el diseño conceptual más apropiado para una base de datos. En la Sección 3.7.3, mostramos algunos de los problemas típicos a los que se suele enfrentar un diseñador de bases de datos cuando tiene que representar un tipo de entidad, una relación y los atributos en un esquema ER. En esta sección nos centraremos en las guías maestras para el diseño de la especialización/generalización y las categorías (tipos de unión) en un modelo EER.

Como ya se comentó en la Sección 3.7.3, el diseño conceptual de una base de datos debe considerarse como un proceso de refinamiento iterativo hasta llegar a lo que más se ajuste a nuestras necesidades. Las siguientes notas pueden ayudar a alcanzar este propósito:

- En general, se pueden definir muchas especializaciones y subclases para que el modelo conceptual sea fiel. Sin embargo, el inconveniente es que el diseño se vuelve algo confuso.
- Si una subclase tiene algunos atributos específicos (locales) y no cuenta con relaciones concretas, puede incluirse en la superclase. Los atributos específicos pueden contener valores NULL para aquellas entidades que no sean miembros de la subclase. Un atributo *tipo* podría especificar esta circunstancia.
- De forma análoga, si todas las subclases de una especialización/generalización cuentan con atributos específicos pero no con relaciones, pueden incluirse en la superclase y sustituirse con uno o más atributos *tipo* que especifiquen la subclase o subclases a la que cada entidad pertenece.
- Deben evitarse los tipos y las categorías a menos que la situación garantice este tipo de construcción, lo cual ocurre en ciertas situaciones prácticas. En caso de ser posible, intentaremos modelar usando especialización/generalización tal y como se ha comentado al final de la Sección 4.4.
- La elección de una restricción disyunción/solapamiento y total/parcial en una especialización/generalización está condicionada por las reglas en las que se está llevando a cabo el modelado. Si los requisitos no indican ningún tipo de restricción particular, la elección predeterminada debería ser el solapamiento parcial, ya que esto no especifica ninguna restricción en los miembros de la subclase.

Como ejemplo para la aplicación de esta indicación, considere el ejemplo mostrado en la Figura 4.6, donde no se muestran atributos específicos (locales). Podemos fundir todas las subclases en la entidad EMPLEADO y añadirle los siguientes atributos:

- TipoTrabajo, cuyo conjunto de valores {‘Secretaria’, ‘Ingeniero’, ‘Técnico’} indicaría la subclase de la primera especialización a la que cada empleado pertenece.
- TipoContrato, cuyo conjunto de valores {‘Fijo’, ‘Temporal’} indicaría la subclase de la segunda especialización a la que cada empleado pertenece.
- EsJefe, cuyo conjunto de valores {‘Si’, ‘No’} indicaría si un empleado es jefe o no.

### 4.5.3 Definiciones formales para los conceptos del modelo EER*

Vamos a resumir ahora los conceptos del modelo EER y dar definiciones formales. Una **clase**[11] es un conjunto o colección de entidades; aquí se incluye cualquier construcción del esquema EER que agrupe entidades, como tipos de entidad, subclases, superclases y categorías. Una **subclase** *S* es una clase cuyas entidades deben ser siempre un subconjunto de las entidades de otra clase llamada la **superclase** *C* de la **relación superclase/**

[11] Aquí, el uso de la palabra *clase* difiere del uso más habitual que se le da en los lenguajes de programación orientados a objetos como C++. En él, una clase es una definición de tipo estructurada junto con sus funciones (operaciones).

**subclase** (o **IS-A, o ES-UN**). Indicamos una relación de este tipo como *C/S.* Para una relación superclase/subclase, siempre debemos tener:

$$S \subseteq C$$

Una **especialización** $Z = \{S_1, S_2, \ldots, S_n\}$ es un conjunto de subclases que tienen la misma superclase $G$, es decir, $G/Si$ es una relación superclase/subclase para $i = 1, 2, \ldots, n$. $G$ recibe el nombre de **tipo entidad-generalizada** (o la **superclase** de la especialización, o una **generalización** de las subclases $\{S_1, S_2, \ldots, S_n\}$ ). $Z$ se dice que es **total** si siempre (y en cualquier momento) tenemos:

$$\bigcup_{i=1}^{n} S_i = G$$

En cualquier otro caso, $Z$ se dice que es **parcial**. $Z$ es una disyunción si siempre tenemos:

$S_i \cap Sj = \varnothing$ (conjunto vacío) para $i \neq j$

En cualquier otro caso, $Z$ se dice que es **solapada**.

Se dice que una subclase $S$ de $C$ es de **predicado definido** si un predicado $p$ de los atributos de $C$ se utiliza para especificar las entidades de $C$ que son miembros de $S$, esto es, $S = C[p]$, donde $C[p]$ es el conjunto de entidades de $C$ que satisfacen $p$. Una subclase que no está definida por un predicado se dice que es de **usuario definido**.

Una especialización $Z$ (o generalización $G$) se dice que es de **atributo definido** si un predicado $(A = c_i)$, donde $A$ es un atributo de $G$ y $ci$ es una constante del dominio de $A$, se utiliza para especificar los miembros de cada subclase $S_i$ en $Z$. Observe que si $c_i \neq c_j$ para $i \neq j$, y $A$ es un atributo de un solo valor, entonces la especialización será una disyunción.

Una **categoría** $T$ es una clase que es un subconjunto de la unión de $n$ superclases $D_1, D_2, \ldots, D_n$, $n > 1$, y está formalmente especificada como:

$$T \subseteq (D_1 \cup D_2 \ldots \cup D_n)$$

Puede usarse un predicado $p_i$ en los atributos de $D_i$ para especificar los miembros de cada $D_i$ que lo son también de $T$. Si se especifica un predicado en cada $D_i$ tenemos

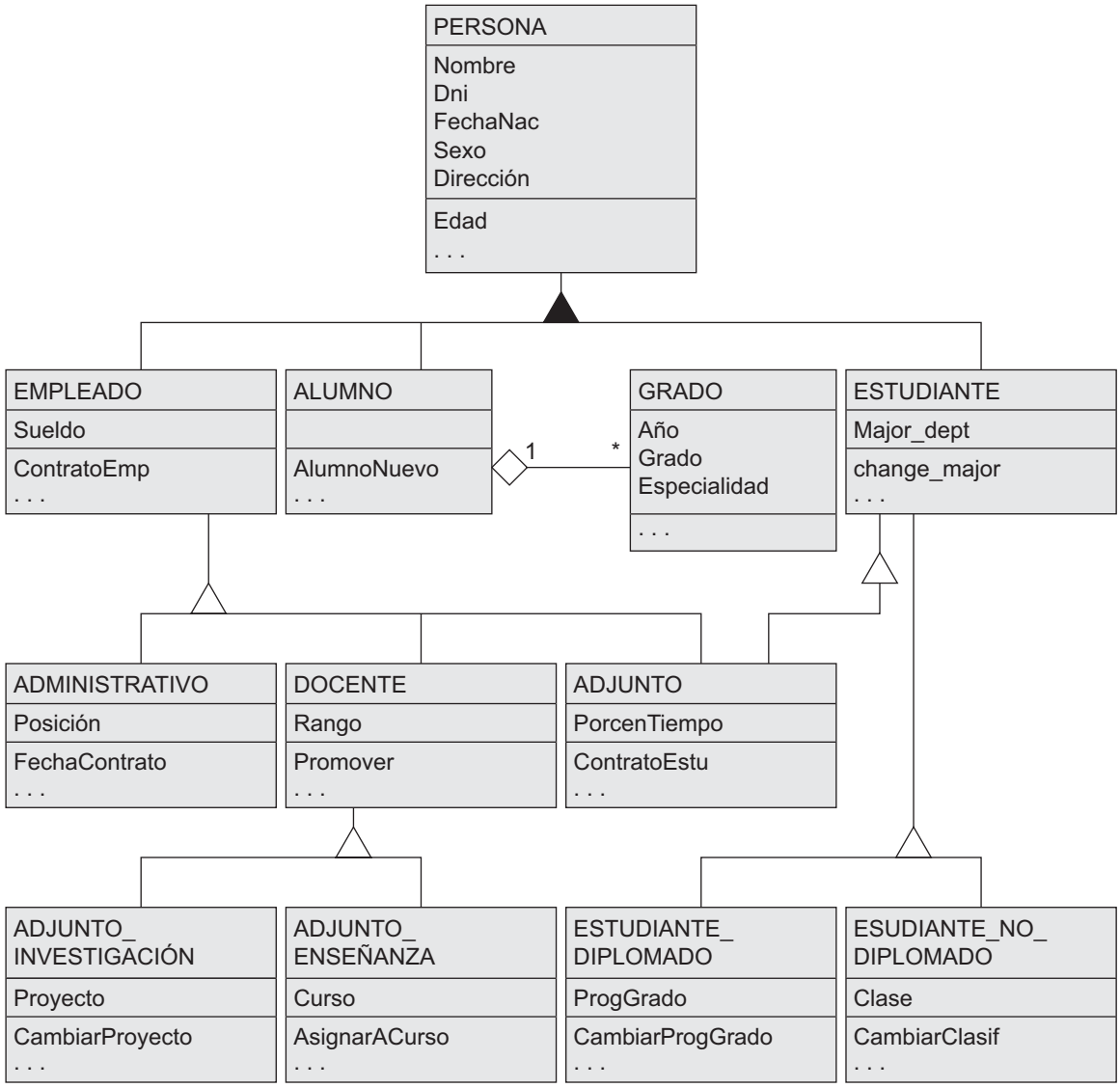
$$T = (D_1[p_1] \cup D_2[p_2] \ldots \cup D_n[p_n])$$

Ahora, es necesario ampliar la definición de **tipo de relación** mostrada en el Capítulo 3 permitiendo que cualquier clase (y no sólo cualquier tipo de entidad) participe en la relación. Por tanto, en esta definición debemos cambiar las palabras *tipo de entidad* por *clase*. La indicación grafica de EER es consecuente con la de ER porque todas las clases están representadas por rectángulos.

## 4.6 Ejemplo de otra notación: representación de la especialización y la generalización en diagramas de clase UML

Vamos a tratar ahora la notación UML para la generalización/especialización y la herencia. En la Sección 3.8 ya se presentó la terminología y la notación básica del diagrama de clase UML. La Figura 4.10 muestra una posible diagramación UML que coincide con el diagrama EER mostrado en la Figura 4.7. La notación básica para la especialización/generalización (véase la Figura 4.10) es conectar las subclases por líneas verticales a otra horizontal, la cual cuenta con un triángulo que conecta la línea horizontal a la superclase a través de otra línea vertical. Un triángulo en blanco indica una especialización/generalización con la restricción de

**Figura 4.10.** Diagrama de clase UML correspondiente al diagrama EER de la Figura 4.7, el cual ilustra la notación UML para una especialización/generalización.




disyunción, mientras que otro relleno especifica otra de tipo *solapamiento*. La superclase raíz recibe el nombre de **clase base**, mientras que los nodos hoja se conocen como **clases hoja**. En ambos casos está permitida la herencia simple o múltiple.

El comentario y el ejemplo anteriores (junto con la Sección 3.8) ofrecen una breve panorámica de los diagramas de clase UML y su terminología. En UML existen muchos otros detalles sobre los que no hemos hablado ya que están fuera del ámbito de este libro, y son poco relevantes para la ingeniería de software. Por ejemplo, las clases pueden ser de varios tipos:

- Las clases abstractas definen atributos y operaciones, pero no cuentan con objetos que se correspondan con esas clases. Se utilizan principalmente para especificar un conjunto de atributos y operaciones que pueden ser heredados.

- Las clases concretas pueden tener objetos (entidades) instanciados que pertenezcan a la clase.
- Las clases plantilla especifican un patrón que puede usarse más adelante para definir otras clases.

En el diseño de bases de datos, nos centramos principalmente en la especificación de clases concretas cuyas colecciones de objetos están permanentemente (o persistentemente) almacenadas en la base de datos. Las notas bibliográficas del final del capítulo muestran algunas referencias a libros que describen con detalle la operativa UML. El Capítulo 12 contiene material adicional sobre UML, y el modelado de objetos en general se trata en el Capítulo 20.

## 4.7 Abstracción de datos, representación del conocimiento y conceptos de ontología

En esta sección comentaremos, en términos abstractos, algunos de los conceptos de modelado que describimos algo más concretamente en nuestra representación de los modelos ER y EER del Capítulo 3 y, anteriormente, en este mismo capítulo. Esta terminología no sólo se emplea en el modelado de datos conceptual, sino también en la literatura sobre inteligencia artificial cuando se trata de la **KR (Representación del conocimiento, *Knowledge Representation***). Esta sección trata sobre las similitudes y las diferencias existentes entre el modelado conceptual y la representación del conocimiento, e introduce algo de terminología alternativa y algunos conceptos adicionales.

El objetivo de las técnicas KR es desarrollar conceptos para el modelado acertado de ciertos **dominios de conocimiento** creando una **ontología**[12] que describe los conceptos del área. Esto se utiliza para almacenar y manipular conocimiento para el dibujo de inferencias, la toma de decisiones o la respuesta a preguntas. Los objetivos de la KR son similares a los de los modelos de datos semánticos, aunque existen importantes similitudes y diferencias entre ambas:

- Las dos disciplinas usan un proceso de abstracción para identificar propiedades comunes y aspectos importantes de los objetos del minimundo (conocido también como dominio de discurso en el KR), a la vez que elimina diferencias insignificantes y detalles sin importancia.
- Ambas disciplinas ofrecen conceptos, restricciones, operaciones y lenguajes para la definición de datos y la representación del conocimiento.
- La KR es, generalmente, más extensa en el ámbito que en los modelos de datos semánticos. Diferentes formas de conocimiento, como reglas (usadas en la inferencia, la deducción y la búsqueda), conocimiento predeterminado e incompleto, y conocimiento espacial y temporal, son representados en esquemas KR. Los modelos de bases de datos están empezando a expandirse para incluir algunos de estos conceptos (consulte el Capítulo 24).
- Los esquemas KR incluyen **mecanismos de razonamiento** que deducen hechos adicionales a partir de otros almacenados en la base de datos. Así pues, mientras la mayor parte de los sistemas de bases de datos están limitados a realizar consultas directas, los sistemas basados en conocimiento que utilizan esquemas KR pueden efectuar preguntas que impliquen **inferencias** sobre los datos almacenados. Las bases de datos actuales están empezando a incluir mecanismos de inferencia (consulte la Sección 24.4).
- Mientras la mayor parte de los modelos de bases de datos se concentran en la representación de sus esquemas, o meta-conocimiento, los esquemas KR suelen mezclar los esquemas con las propias instancias para proporcionar mayor flexibilidad a la hora de representar excepciones. Esto, con frecuencia, suele desembocar en deficiencias cuando estos esquemas KR son implementados, especialmente

[12] Una *ontología* es algo similar a un esquema conceptual, aunque con más conocimiento, reglas y excepciones.

cuando se comparan con bases de datos y cuando es preciso almacenar una gran cantidad de datos (hechos).

En esta sección, trataremos cuatro **conceptos de abstracción** utilizados en modelos de datos semánticos, como el EER y en esquemas KR: (1) clasificación e instanciación, (2) identificación, (3) especialización y generalización y (4) agregación y asociación. Los conceptos pareados de clasificación e instanciación son inversos a los de generalización y especialización. La agregación y la asociación también están relacionadas. Para clarificar el proceso de abstracción de datos y mejorar nuestra comprensión de los procesos relacionados del diseño de esquema conceptual, hablaremos de estos conceptos abstractos y de su relación con las representaciones concretas usadas en el modelo EER. Terminaremos la sección con un breve comentario acerca de la *ontología*, la cual está siendo usada ampliamente en recientes investigaciones de la representación del conocimiento.

## 4.7.1 Clasificación e instanciación

El proceso de **clasificación** supone la asignación sistemática de objetos/entidades similares a objetos de tipo clase/entidad. Ahora podemos describir (hablando de bases de datos) o razonar (en KR) las clases en lugar de los objetos individuales. Las colecciones de objetos comparten los mismos tipos de atributos, relaciones y restricciones, y al clasificar los objetos simplificamos el descubrimiento de sus propiedades. La **instanciación** es la operación inversa a la clasificación y se refiere al proceso de generación y examen de los distintos objetos de una clase. Por tanto, una instancia de un objeto está relacionada con su clase objeto por la relación **ES-UNA-INSTANCIA-DE** o **ES-UN-MIEMBRO-DE**. Aunque los diagramas EER no muestran las instancias, los UML disponen de una forma de instanciación que permite la visualización de objetos individuales. En nuestra introducción a UML no describimos esta característica.

En general, los objetos de una clase deben tener una estructura similar. Sin embargo, algunos objetos pueden mostrar propiedades que difieran en parte de las de otros objetos de la clase; estos **objetos excepción** también tienen que modelarse, y los esquemas KR permiten más excepciones que los de base de datos. Además, ciertas propiedades se aplican a toda la clase y no a objetos individuales; los esquemas KR permiten este tipo de **propiedades de clase**, al igual que los UML.

En el modelo EER, las entidades están clasificadas en tipos de entidad según sus relaciones y atributos básicos. Las entidades, a su vez, están divididas en subclases y categorías en base a las similitudes y diferencias (excepciones) existentes entre ellas. Las instancias relación están clasificadas en tipos de relación. Por tanto, los tipos de entidad y de relación, las subclases y las categorías son diferentes tipos de clases en el modelo EER. Además, no ofrece explícitamente las propiedades de clase, aunque puede desarrollarse para hacerlo. En UML, los objetos están clasificados en clases, y es posible mostrar tanto las propiedades de la clase como la de los objetos individuales.

Los modelos de representación de conocimiento permiten múltiples esquemas de clasificación en los que una clase es una *instancia* de otra clase (llamada **meta-clase**). Tenga en cuenta que esto no puede mostrarse directamente en un modelo EER, ya que sólo disponemos de dos niveles: clases e instancias. Por tanto, la única relación posible entre las clases es la de tipo superclase/subclase, mientras que en algunos esquemas KR es posible representar directamente una relación adicional clase/instancia en una jerarquía de clase. Una instancia puede ser, por sí misma, otra clase, permitiendo esquemas de clasificación de múltiples niveles.

## 4.7.2 Identificación

La **identificación** es el proceso de abstracción por el que las clases y los objetos son identificables de forma única por medio de algún **identificador**. Por ejemplo, un nombre de clase identifica inequívocamente a toda esa clase. Es necesario un mecanismo adicional para mantener separadas distintas instancias de objetos mediante identificadores de objeto. Además, es preciso identificar múltiples manifestaciones del mismo obje-

to del mundo real en la base de datos. Por ejemplo, podemos tener una tupla <'Matías Flis', '610618', '376-9821'> en una relación PERSONA y otra tupla <'301-54-0836', 'CC', 3.8> en ESTUDIANTE que pareciera que representasen a la misma persona. No existe manera de identificar que estos dos objetos de base de datos (tuplas) representan a la misma persona a menos que, durante la *fase de diseño*, preparemos los mecanismos adecuados para establecer esta relación cruzada. Por consiguiente, la identificación es necesaria a dos niveles:

- Para distinguir entre objetos y clases de la base de datos.
- Para identificar objetos de base de datos y relacionarlos con sus homólogos del mundo real.

En el modelo EER, la construcción de un esquema de identificación se basa en un sistema de nombres únicos para los constructores. Por ejemplo, cada clase (ya sea un tipo de entidad, una subclase, una categoría o un tipo de relación) debe contar con un nombre distinto. Los nombres de atributo de una clase específica también deben ser diferentes. También son necesarias reglas para identificar con claridad las referencias a los nombres de atributo en una jerarquía o entramado de especialización o generalización.

A nivel de objeto, se emplean los valores de los atributos clave para distinguir entre entidades de un tipo de entidad particular. Para los tipos débiles, las entidades se identifican por una combinación de sus propios valores clave parciales y las entidades que están relacionadas en el tipo (o tipos) de entidad propietaria. Las instancias de relación están identificadas por alguna combinación de las entidades que relacionan, en base al índice de cardinalidad especificado.

### 4.7.3 Especialización y generalización

La especialización es el proceso para clasificar una clase en subclases más especializadas. La generalización es el proceso inverso de generalizar varias clases en una clase abstracta de nivel superior que incluya los objetos de todas esas clases. La especialización es un refinamiento conceptual, mientras que la generalización es una síntesis conceptual. Las subclases se emplean en el modelo EER para representar la especialización y la generalización. La relación entre una subclase y su superclase recibe el nombre de relación **ES-UNA-SUBCLASE-DE** o, abreviando, una relación **ES-UN/ES-UNA**.

### 4.7.4 Agregación y asociación

La agregación es un concepto abstracto para la construcción de objetos complejos a partir de sus objetos componente. Existen tres situaciones en las que este concepto puede estar relacionado con el modelo EER. El primero se produce cuando añadimos atributos de un objeto para formar el objeto completo. El segundo se da cuando representamos una relación de agregación como una relación común. El tercer caso, para el cual no se proporciona explícitamente el modelo EER, supone la posibilidad de combinar objetos que están relacionados con una instancia relación particular en un *objeto agregado de nivel superior.* Esto suele ser útil a veces cuando este objeto, por sí mismo, está relacionado con otro. Llamamos a la relación existente entre los objetos primitivos y sus objetos agregados **ES-UNA-PARTE-DE**; la situación inversa recibe el nombre de **ES-UN-COMPONENTE-DE**. UML proporciona soporte para los tres tipos de agregación.

La abstracción de **asociación** se utiliza para asociar objetos procedentes de varias *clases independientes.* Por tanto, es muy parecido al segundo uso de la agregación. En el modelo EER está representado por los tipos de relación, mientras que en UML lo está por las asociaciones. Esta relación abstracta se llama **ESTÁ-ASOCIADA-CON**.

Para entender mejor los distintos usos de la agregación, considere el esquema ER de la Figura 4.11(a), el cual almacena información sobre las entrevistas realizadas por los aspirantes a obtener un empleo en distintas empresas. La clase EMPRESA es una agregación de los atributos (u objetos componente) NombreEmpresa (nombre de la empresa) y DirEmpresa (dirección de la empresa), mientras que ASPIRANTE_TRABAJO es una agregación de Dni, Nombre, Dirección y Teléfono. Los atributos de relación NombreContacto y TlfContacto

representan el nombre y el teléfono de la persona de la empresa responsable de la entrevista. Supongamos que alguna de las entrevistas tiene como consecuencia una oferta de trabajo, mientras que otras no. Podríamos querer tratar ENTREVISTA como una clase para asociarla con OFERTA_TRABAJO. El esquema de la Figura 4.11(b) es *incorrecto* porque supone que cada instancia de la relación entrevista tiene una oferta de trabajo. El esquema de la Figura 4.11(c) *no está permitido*, ya que el modelo ER no permite establecer relaciones entre relaciones.

Una forma de representar esta situación es crear una clase agregada de nivel superior compuesta por EMPRESA, ASPIRANTE_TRABAJO y ENTREVISTA, y relacionarla con OFERTA_TRABAJO (véase la Figura 4.11[d]). Aunque el modelo EER, como ya hemos mencionado, no permite esta situación, algunos modelos de datos semánticos sí que lo hacen y nombran al objeto resultante un **compuesto** u **objeto molecular**. Otros modelos tratan a los objetos entidad y relación uniformemente y, por ello, permiten las relaciones entre relaciones (véase la Figura 4.11[c]).

Para representar correctamente esta situación en el modelo ER, necesitamos crear un nuevo tipo de entidad débil ENTREVISTA (véase la Figura 4.11[e]), y relacionarlo con OFERTA_TRABAJO. De este modo, siempre es posible representar correctamente estas situaciones en el modelo ER creando tipos de entidad adicionales aunque, conceptualmente, puede ser más deseable permitir la representación directa de la agregación, como puede verse en la Figura 4.11(d), o consentir las relaciones entre relaciones (véase la Figura 4.11[c]).

La principal distinción estructural entre la agregación y la asociación es que cuando una instancia de asociación se borra, los objetos que participan de ella pueden seguir existiendo. Sin embargo, si abogamos por la noción de objeto agregado (por ejemplo, un COCHE compuesto de objetos MOTOR, CHASIS y RUEDAS), el borrado del objeto COCHE implica la eliminación de todos los demás.

### 4.7.5 Ontologías y la semántica Web

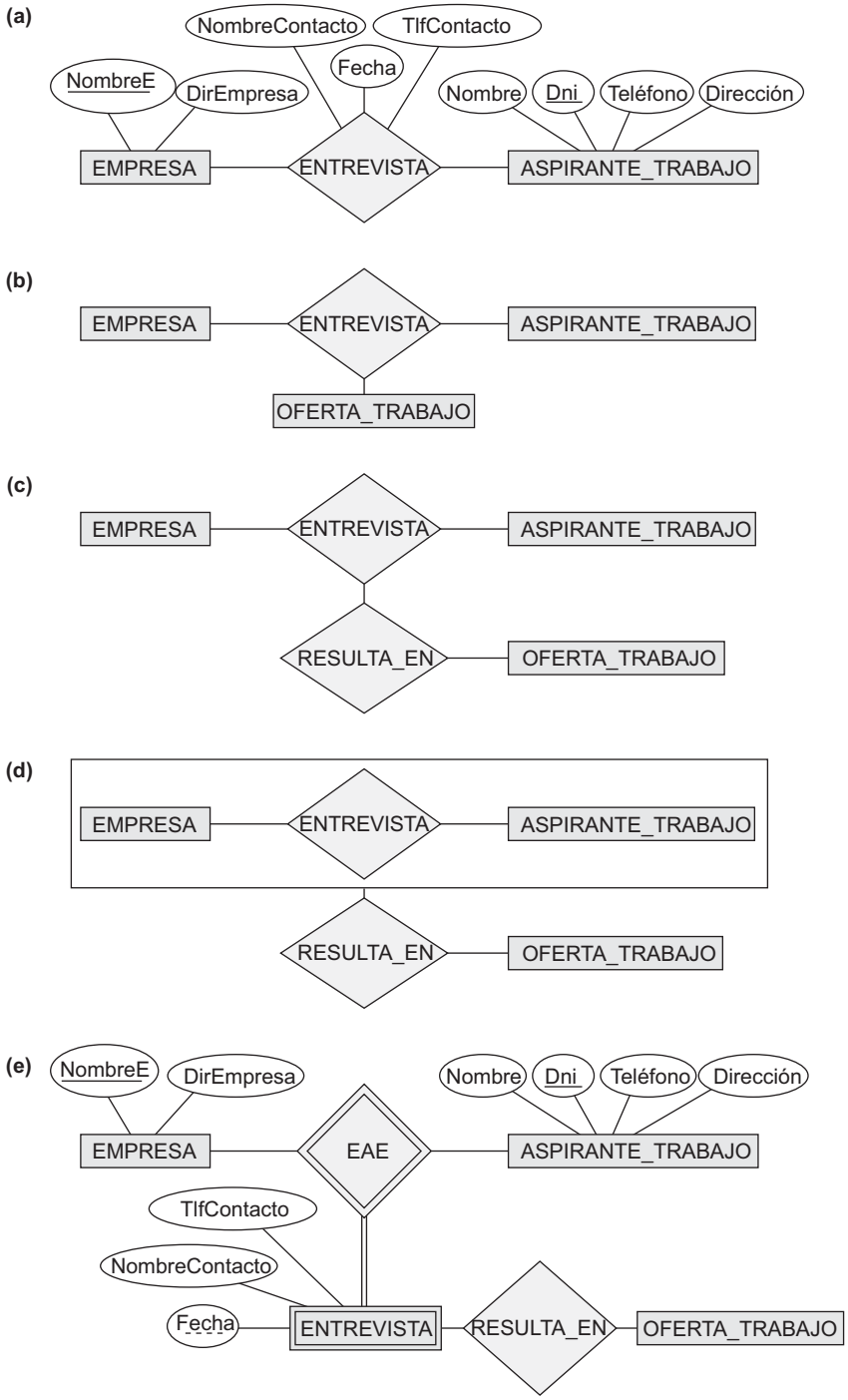
Últimamente, la cantidad de datos informatizados y de información disponible en la Web está fuera de control. Para ello, se utilizan muchos modelos y formatos diferentes. Además de los modelos de bases de datos mostrados en este libro, una gran cantidad de información se almacena en forma de **documentos**, los cuales precisan de una estructura mucho menor de la necesaria en la información de una base de datos. **Semantic Web** es un proyecto de investigación que está intentando permitir el intercambio de información entre computadores de la Web, además de intentar crear modelos de representación de conocimiento que sean lo más generales posible para permitir el intercambio y la búsqueda de información significativa entre máquinas. Se está intentando que la *ontología* sea la piedra angular sobre la que se asiente Semantic Web, y está íntimamente relacionado con la representación del conocimiento. En esta sección, se ofrecerá una breve introducción sobre qué es la ontología y cómo puede usarse para automatizar la comprensión, búsqueda e intercambio de información.

El estudio de las ontologías intenta describir las estructuras y las relaciones que son posibles en la realidad a través de vocabulario común; por consiguiente, puede considerarse como una forma de describir el conocimiento de la realidad de una cierta comunidad. La ontología tuvo su origen en la filosofía y la metafísica. Una definición de **ontología** comúnmente usada es la de la *especificación* de una *conceptualización*.[13]

En esta definición, una **conceptualización** es el conjunto de conceptos usados para representar la parte de realidad o conocimiento que son de interés a una comunidad de usuarios. La **especificación** se refiere al lenguaje y el vocabulario empleados para especificar la conceptualización. La ontología incluye tanto la *especificación* como la *conceptualización.* Por ejemplo, a través de dos ontologías diferentes puede especificarse la misma conceptualización. Aun basándonos en esta definición general, no existe consenso acerca de lo que es exactamente la ontología. Éstas son algunas formas de describirla:

[13] Esta definición la propuso Gruber (1995).

**Figura 4.11.** Agregación. (a) Tipo de relación ENTREVISTA. (b) Incluyendo OFERTA_TRABAJO en un tipo de relación ternario (incorrecto). (c) Con la relación RESULTA_EN participando en otras relaciones (no está permitido en ER). (d) Usando una agregación y un objeto compuesto (molecular) (normalmente, no está permitido en ER, aunque sí en algunas herramientas de modelado). (e) Representación correcta en ER.



- Un **diccionario de sinónimos** (e incluso un **diccionario** o un **glosario** de términos) describen las relaciones existentes entre las palabras (vocabulario) que representan diferentes conceptos.
- Una **taxonomía** describe el modo en que están relacionados los conceptos de un dominio particular de conocimiento usando estructuras similares a las empleadas en una especialización o una generalización.
- Un **esquema de base de datos** detallado está considerado por algunos como una ontología que describe los conceptos (entidades y atributos) y las relaciones de un minimundo real.
- Una **teoría lógica** usa los conceptos de la lógica matemática para intentar definir los conceptos y sus interrelaciones.

Habitualmente, los conceptos utilizados para describir ontologías son muy similares a los que empleamos en el modelado conceptual, como entidades, atributos, relaciones, especializaciones, etc. La diferencia principal entre una ontología y, digamos, un esquema de base de datos es que el esquema suele limitarse a describir un pequeño subconjunto de un minimundo real con el objetivo de almacenar y administrar datos. Una ontología suele considerarse algo más general, ya que intenta describir una parte de la realidad o un área de interés (por ejemplo, términos médicos, aplicaciones de comercio electrónico) de la forma más completa posible.

# 4.8 Resumen

En este capítulo hemos estudiado las extensiones del modelo ER que mejoran sus capacidades de representación. Llamamos al modelo resultante ER mejorado o modelo EER. Presentamos el concepto de una subclase y su superclase y el mecanismo relacionado de herencia atributo/relación. Mostramos cómo, a veces, es necesario crear clases de entidades adicionales, ya fuera debido a atributos específicos adicionales o debido a tipos de relación concretos. Abordamos los dos procesos principales para la definición de jerarquías y entramados superclase/subclase: la especialización y la generalización.

A continuación, mostramos la forma de representar estas nuevas construcciones en un diagrama EER. También debatimos los diferentes tipos de restricciones que pueden aplicarse a la especialización o la generalización: total/parcial y disyunción/solapamiento. Además, puede definirse un predicado para una subclase o un atributo para una especialización. Explicamos las diferencias existentes entre subclases definidas por usuario y de predicado definido y entre especializaciones del mismo tipo. Para terminar, planteamos el concepto de una categoría o tipo unión, la cual se define como un subconjunto de la unión de dos o más clases, y ofrecimos definiciones formales de todos los conceptos presentados.

Mostramos parte de la notación y la terminología UML para representar la especialización y la generalización. En la Sección 4.7 abordamos brevemente la disciplina de la representación del conocimiento y el modo que está relacionado con el modelado de datos semántico. También ofrecimos una panorámica y un resumen de los tipos de conceptos de la representación abstracta de datos: la clasificación y la instanciación, la identificación, la especialización y la generalización, y la agregación y la asociación, sin olvidarnos de la forma en que los conceptos EER y UML están relacionados con todos ellos.

## Preguntas de repaso

**4.1.** ¿Qué es una subclase? ¿Cuándo es necesaria una subclase en el modelado de datos?

**4.2.** Defina los siguientes términos: superclase de una subclase, relación superclase/subclase, relación es-una, especialización, generalización, categoría, atributos específicos (locales) y relaciones específicas.

**4.3.** Aborde el mecanismo de herencia atributo/relación. ¿Por qué es útil?

**4.4.** Comente las subclases definidas por usuario y de predicado definido, e identifique las diferencias existentes entre ellas.

**4.5.** Plantee las especializaciones definidas por usuario y de predicado definido, e identifique las diferencias existentes entre ellas.

**4.6.** Explique los dos tipos principales de restricciones en las especializaciones y las generalizaciones.

**4.7.** ¿Cuál es la diferencia entre una jerarquía y un entramado de especialización?

**4.8.** ¿Cuál es la diferencia entre una especialización y una generalización? ¿Por qué no podemos mostrar esta diferencia en los diagramas de esquema?

**4.9.** ¿En qué difiere una categoría de una subclase compartida corriente? ¿Para qué se usa una categoría? Argumente su respuesta con ejemplos.

**4.10.** Por cada uno de los siguiente términos UML (consulte las Secciones 3.8 y 4.6), indique el correspondiente en el modelo EER, en caso de existir: objeto, clase, asociación, agregación, generalización, multiplicidad, atributos, discriminador, enlace, atributo de enlace, asociación reflexiva y asociación cualificada.

**4.11.** Comente las diferencias principales existentes entre la notación en los diagramas de esquema EER y los de clase UML comparando el modo en que se representan los conceptos comunes.

**4.12.** Enumere los distintos conceptos de abstracción de datos y los conceptos de modelado correspondientes en el modelo EER.

**4.13.** ¿Qué característica de agregación no existe en el modelo EER? ¿Cómo podría mejorarse para soportarla?

**4.14.** ¿Cuáles son las principales similitudes y diferencias existentes entre las técnicas de modelado de bases de datos conceptuales y las de representación del conocimiento?

**4.15.** Comente las similitudes y diferencias existentes entre una ontología y un esquema de base de datos.

## Ejercicios

**4.16.** Diseñe un esquema EER para una aplicación de bases de datos en la que esté interesado. Especifique todas las restricciones que necesite. Asegúrese de que el esquema dispone de, al menos, cinco tipos de entidad, cuatro tipos de relación, un tipo de entidad débil, una relación superclase/subclase, una categoría y un tipo de relación n-cualquiera (n > 2).

**4.17.** Considere el esquema ER BANCO de la Figura 3.21, y que es necesario controlar los diferentes tipos de CUENTA (AHORRO, ARQUEO, etc.) y PRÉSTAMO (COCHE, HIPOTECARIO, etc.). Suponga también que es aconsejable gestionar cada TRANSACCIÓN de una CUENTA (depósitos, retiradas, cheques, etc.) y cada PAGO del PRÉSTAMO; ambas situaciones incluyen la cantidad, la fecha y la hora. Modifique el esquema BANCO usando los conceptos ER y EER de especialización y generalización. Haga constar cualquier supuesto que haga sobre requerimientos adicionales.

**4.18.** La siguiente historia narra una versión simplificada de la organización de las instalaciones para unas Olimpiadas de verano. Dibuje un diagrama EER que muestre los tipos de entidad, los atributos, las relaciones y las especializaciones para esta aplicación. Haga constar cualquier supuesto que haga. Las instalaciones olímpicas están divididas en complejos deportivos, los cuales, a su vez, son de tipo *monodeportivo* y *multideportivo*. Los complejos multideportivos tienen áreas diseñadas para cada una de las especialidades y cuentan con un indicador (por ejemplo, centro, esquina NE, etc.). Un complejo dispone de una localización, su jefe de organización, el área ocupada, etc. Cada complejo alberga una serie de eventos (por ejemplo, en la pista de carreras se pueden disputar varios tipos de ellas). Cada evento está planificado para una fecha, tendrá una duración, un número de participantes y jueces, etc. Será preciso mantener una lista de todos los jueces junto con las pruebas en las que estarán presentes. Para cada prueba será necesario un equipamiento distinto (por ejemplo, las porterías, las pértigas, las barras paralelas) y su mantenimiento. Ambos tipos de complejos (monodeportivo y multideportivo) contarán con distintos tipos de información. Para cada uno, es preciso mantener el número de instalaciones necesarias, junto con un presupuesto aproximado.

**4.19.** Identifique los conceptos más importantes representados en el estudio de la base de datos de una biblioteca mostrado más adelante. En particular, preste atención a las abstracciones de la clasificación (tipos de entidad y de relación), la agregación, la identificación y la especialización/generalización. Especifique las restricciones de cardinalidad (mínima, máxima) siempre que sea posible. Enumere los detalles que afectarán al diseño eventual, pero que no tengan interrelación con el conceptual. Identifique de forma separada las restricciones semánticas. Dibuje un diagrama EER de esta base de datos.

**Caso a estudiar.** La Georgia Tech Library (GTL) cuenta aproximadamente con unos 16.000 miembros, 100.000 títulos y 250.000 volúmenes (una media de 2,5 copias por libro). Alrededor del 10% se encuentra permanentemente fuera en modo de préstamo. Los bibliotecarios aseguran que los libros que se deseen pedir prestados lo estarán en el momento en que los miembros así lo deseen. Además, es preciso que conozcan cuántas copias están prestadas en cada momento. Existe un catálogo de libros *online* disponible que enumera las obras por autor, título y área. Para cada una de ellas, el catálogo mantiene un descriptor de libro que oscila desde una frase a varias páginas. Se quiere que estas referencias estén accesibles para los bibliotecarios cuando un miembro solicita información acerca de un libro. La plantilla de la biblioteca incluye un bibliotecario jefe, los bibliotecarios departamentales asociados, los de referencia, el plantel de verificación y los bibliotecarios asistentes.

Los libros pueden retenerse durante 21 días, y los miembros sólo pueden tener 5 ejemplares a la vez. Por lo general, los usuarios devuelven los libros a las tres o cuatro semanas, y la mayoría sabe que disponen de una semana de gracia antes de que se les notifique esta situación, por lo que todos intentan hacerlo antes de que expire dicho periodo. Es necesario hacer un recordatorio de la devolución a alrededor del 5% de los miembros, y la mayor parte de los libros se devuelven un mes después de vencer la fecha tope. Incluso, existe un 5% de obras que se pierden definitivamente. Los miembros más activos de la biblioteca son aquéllos que solicitan libros, al menos, diez veces al año. El 1% de los miembros más activos realiza el 15% de las peticiones, y el 10% de los miembros más activos el 40%. Cerca del 20% nunca realiza una petición.

Para pertenecer a la biblioteca, los solicitantes rellenan un formulario en el que se incluye su DNI, su dirección postal personal y la de su centro de estudios y los números de teléfono. Los bibliotecarios expiden entonces una tarjeta magnética numerada con su foto y válida para cuatro años. Un mes antes de que expire, se le envía un mensaje de renovación. Los profesores de los centros de enseñanza son considerados miembros de forma automática. Cuando un nuevo profesor entra en un colegio, se obtiene la información necesaria de su registro de empleado y se envía por correo una tarjeta a su dirección profesional. Los profesores pueden revisar libros en intervalos de tres meses, y su periodo de gracia es de dos semanas. Las renovaciones se remiten a su dirección profesional.

La biblioteca no presta ciertos libros, como obras de referencia, volúmenes raros y mapas, por lo que los bibliotecarios deben ser capaces de distinguir qué obras pueden prestar y cuáles no. Además, cuentan con una lista de algunos libros que resultaría interesante adquirir pero que no pueden, como obras raras o descatalogadas y otras que se perdieron o se destruyeron y que no han sido reemplazadas. Algunos libros pueden tener el mismo título; por consiguiente, este dato no puede usarse como campo clave. Cada uno de ellos está identificado por su ISBN (*International Standard Book Number*), un código internacional único asignado a todos los libros. Dos obras con el mismo título pueden tener ISBN diferentes si están escritos en idiomas diferentes o tienen distintas encuadernaciones (tapa dura, tapa blanda). Las ediciones de la misma obra tienen distintos ISBN.

La base de datos propuesta debe diseñarse de forma que controle los miembros, los libros, el catálogo y los préstamos.

**4.20.** Diseñe una base de datos para gestionar la información de un museo de arte. Asumimos que los siguientes datos fueron recopilados:

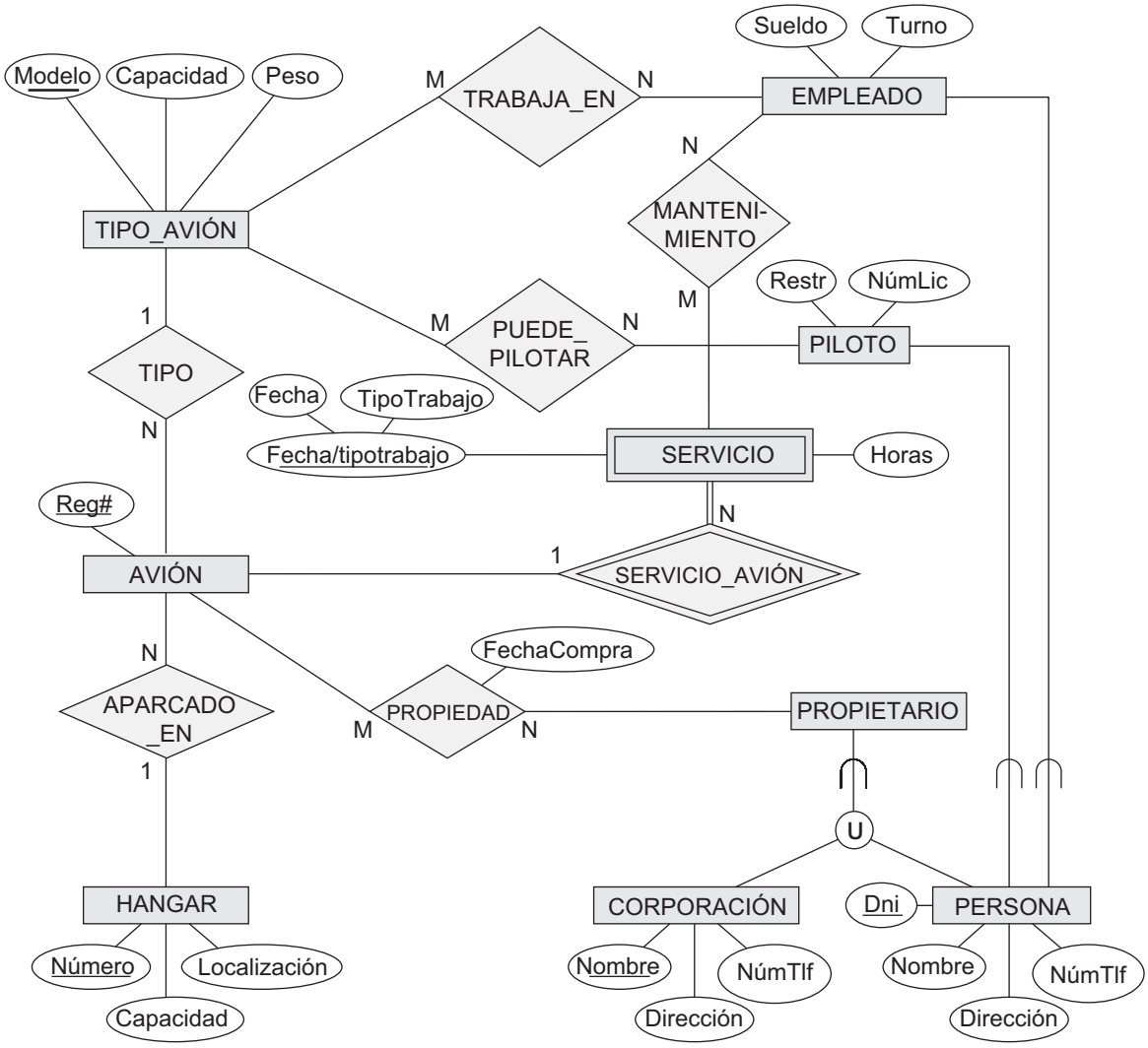
- El museo dispone de una colección de OBJETOS_DE_ARTE. Cada uno de ellos cuenta con un identificador único (Id), un Artista (en caso de conocerse), el Año de creación (si se conoce), un Título y una Descripción. Los objetos están clasificados de varias formas, tal y como se comentará más adelante.
- OBJETOS_DE_ARTE está categorizada en función a sus tipos, de los cuales hay tres principales: PINTURA, ESCULTURA y MONUMENTO más un cuarto llamado OTRO para acomodar a aquéllos que no se ajustan a ninguno de los otros tres.
- Una PINTURA tiene un TipoPintura (aceite, al agua, etc.), el material sobre el que está Dibujado (papel, lienzo, madera, etc.) y un EstiloPintura (moderno, abstracto, etc.).
- Una ESCULTURA o un MONUMENTO tiene el Material sobre el que fue creado (madera, piedra, etc.), una Altura, una Anchura y un EstiloEscultura.
- Un objeto de arte encuadrado en la categoría OTRO tiene un TipoObra (impresión, fotografía, etc.) y un EstiloOtro.
- Los OBJETOS_DE_ARTE están clasificados como una COLECCIÓN_PERMANENTE (aquéllos que son propiedad del mueso) y como PRESTADO. Los datos con los que contamos del primer tipo son la FechaAdqusición, su Estado (en exhibición, en préstamo o almacenada) y su Coste. La información tomada sobre los objetos de tipo PRESTADO incluye la Colección propietaria de la misma, su FechaPréstamo y la FechaDevolución.
- La información acerca del país o la cultura de Origen (Italia, Egipto, América, India, etc.) y su Época (Renacimiento, Moderna, Antigua, etc.) se almacena en cada OBJETO_DE_ARTE.
- El museo conserva información sobre el ARTISTA, en caso de conocerse: Nombre, FechaNac (si se sabe), FechaFallecimiento (si corresponde), PaísOrigen, Época, EstiloPrincipal y Descripción. Se asume que el Nombre es un dato único.
- Se celebran distintas EXHIBICIONES, cada una con su Nombre, su FechaInicio y su FechaFinalización. Las EXHIBICIONES están relacionadas con los objetos de arte que están en estado de exhibición durante la misma.
- También se mantiene información sobre otras COLECCIONES con las que el museo interactúa, incluyendo su Nombre (único), el Tipo (museo, personal, etc.), Descripción, Dirección, Teléfono y PersonaContacto actual.

Diseñe un diagrama EER para esta aplicación. Haga constar cualquier supuesto que haga, y justifíquelo.

**4.21.** La Figura 4.12 muestra un diagrama EER para la base de datos de un pequeño aeropuerto que se utiliza para mantener la información de las aeronaves, sus propietarios, los empleados del aeropuerto y los pilotos. Éstos son los datos recopilados. Cada AVIÓN dispone de un número de registro [Reg#], es de un tipo particular [TIPO] y está aparcado en un hangar concreto [APARCADO_EN].

Cada TIPO_AVIÓN tiene un número de [Modelo], una [Capacidad] y un [Peso]. Cada HANGAR cuenta con un [Número], una [Capacidad] y una [Localización]. La base de datos también controla los propietarios de cada aeronave [PROPIETARIO] y los empleados encargados del [MANTENIMIENTO]. Cada PROPIETARIO está relacionado con un AVIÓN e incluye la fecha de adquisición [FechaCompra]. Cada relación en MANTENIMIENTO asocia a un empelado con un [SERVICIO]. Cada avión se somete a revisión cada cierto tiempo; por tanto, está relacionado con [SERVICIO_AVIÓN] por un número de registro SERVICIO, cada uno de los cuales incluye como atributos la fecha de mantenimiento [Fecha], el número de horas empleadas en el trabajo [Horas] y el tipo de servicio efectuado [TipoTrabajo]. Utilizamos una entidad débil [SERVICIO] para representar el servicio del avión, ya que su número de registro se utiliza para identificar un registro de servicio. Un PROPIETARIO puede ser tanto una persona como una corporación. Por tanto, utilizamos

**Figura 4.12.** Esquema EER para la base de datos PEQUEÑO_AEROPUERTO.




un tipo de unión (categoría) [PROPIETARIO] que es un subconjunto de los tipos de entidad [CORPORACIÓN] y [PERSONA]. Tanto los pilotos [PILOTO] como los empleados [EMPLEADO] son subclases de PERSONA. Cada PILOTO tiene, como atributos específicos, su número de licencia de vuelo [NúmeroLicencia] y sus restricciones [Restricción], mientras que los de cada EMPLEADO son el [Salario] y el [Turno]. Toda entidad PERSONA de la base de datos tiene su número del Documento nacional de identidad [Dni], un [Nombre], una [Dirección] y un [NúmeroTeléfono]. Para cada CORPORACIÓN existe un [Nombre], una [Dirección] y un [NúmeroTeléfono]. La base de datos también mantiene los tipos de aviones que cada piloto está autorizado a pilotar [PUEDE_PILOTAR] y aquéllos en los que cada empleado puede realizar tareas de mantenimiento [TRABAJA_EN]. Observe cómo el esquema EER de PEQUEÑO_ AEROPUERTO de la Figura 4.12 puede representarse en notación UML. (*Nota*. No hemos comentado la forma de representar las categorías [tipos de unión] en UML, por lo que no tiene que indicarlas ni en ésta ni en la siguiente pregunta.)

**4.22.** Desarrolle el modo de representar en notación UML el esquema EER UNIVERSIDAD de la Figura 4.9.

**4.23.** Considere los conjuntos de entidades y atributos mostrados en la siguiente tabla. Coloque una marca en una de las columnas de cada fila para indicar la relación entre las columnas derecha y la situada más a la izquierda.

(a) El lado izquierdo tiene una relación con el derecho.

(b) El lado derecho es un atributo del lado izquierdo.

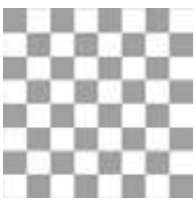
(c) El lado izquierdo es una especialización del derecho.

(d) El lado izquierdo es una generalización del derecho.

| Entidad | (a) Tiene una relación con | (b) Tiene un atributo que es | (c) Es una especialización de | (d) Es una generalización de | Entidad o atributo |
|---|---|---|---|---|---|
| 1. MADRE | | | | | PERSONA |
| 2. HIJA | | | | | MADRE |
| 3. ESTUDIANTE | | | | | PERSONA |
| 4. ESTUDIANTE | | | | | IdEstudiante |
| 5. COLEGIO | | | | | ESTUDIANTE |
| 6. COLEGIO | | | | | AULA |
| 7. ANIMAL | | | | | CABALLO |
| 8. CABALLO | | | | | Raza |
| 9. CABALLO | | | | | Edad |
| 10. EMPLEADO | | | | | DNI |
| 11. MUEBLE | | | | | SILLA |
| 12. SILLA | | | | | Peso |
| 13. HUMANO | | | | | MUJER |
| 14. SOLDADO | | | | | PERSONA |
| 15. SOLDADO_ENEMIGO | | | | | PERSONA |

**4.24.** Dibuje un diagrama UML para almacenar en una base de datos una partida de ajedrez. Puede consultar la dirección http://www.chessgames.com para obtener información acerca de una aplicación similar a la que tiene que desarrollar. Documente claramente cualquier decisión que tome en su diagrama UML. Las siguientes pueden ser algunas de esas decisiones:

1. La partida se realiza entre dos jugadores.
2. Se juega en un tablero de 8×8 similar al mostrado a continuación:

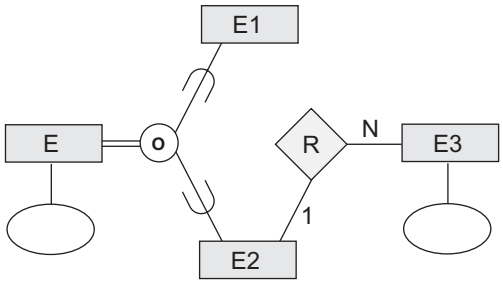


3. Los jugadores asumen un color (negro o blanco) al inicio de la partida.
4. Cada jugador empieza con las siguientes piezas:
   a. 1 rey
   b. 1 reina
   c. 2 torres
   d. 2 alfiles
   e. 2 caballos
   f. 8 peones
5. Cada pieza se encuentra en su posición inicial.
6. Cada pieza cuenta con su propio conjunto de movimientos permitidos en función al estado de la partida. No es necesario preocuparse de qué movimientos son legales y cuáles no, excepto en los siguientes casos:
   a. Una pieza puede moverse a un cuadro vacío o capturar una pieza del contrario.
   b. Si una pieza es capturada, se elimina del tablero.
   c. Si un peón alcanza la última fila es "promocionado", convirtiéndose en otra pieza (reina, torre, alfil o rey).

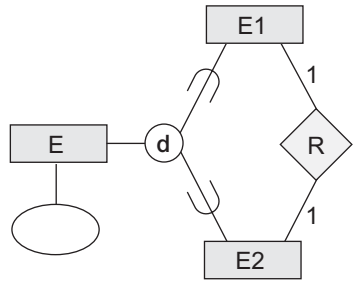
*Nota*: Alguna de estas funciones pueden aplicarse a múltiples clases.

**4.25.** Dibuje un diagrama EER para la partida de ajedrez descrita en el Ejercicio 4.24. Concéntrese en los aspectos de almacenamiento persistente del sistema. Por ejemplo, puede que sea necesario recuperar todos los movimientos de cada partida en orden secuencial.

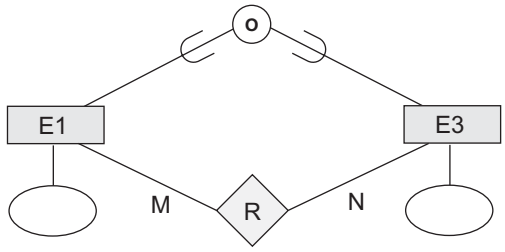
**4.26.** ¿Cuáles de los siguientes diagramas EER son incorrectos y por qué? Documente claramente cualquier decisión que tome.




**(a)**

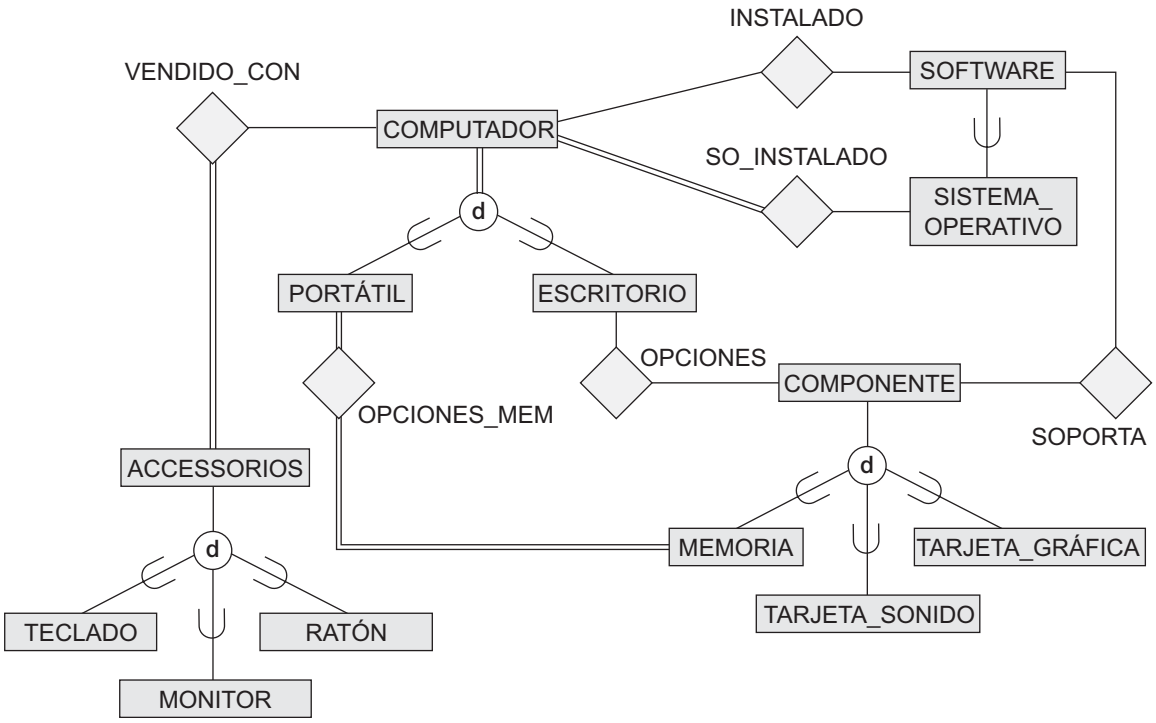



**(b)**




**(c)**

**4.27.** Considere un diagrama EER en el que se describe los sistemas informáticos de una empresa. Proporcione sus propios atributos y claves para cada tipo de entidad. Facilite las restricciones de cardinalidad máximas justificando su elección. Escriba una descripción completa de lo que representa este diagrama EER.




## Ejercicios de práctica

**4.28.** Considere el diagrama EER de la base de datos UNIVERSIDAD mostrada en la Figura 4.9. Realice este diseño usando alguna herramienta de modelado de datos como ERWin o Rational Rose. Elabore una lista de las diferencias de notación existente entre el diagrama en modo texto y el equivalente construido con la herramienta.

**4.29.** Considere el diagrama EER de la base de datos del PEQUEÑO_AEROPUERTO mostrada en la Figura 4.12. Realice este diseño usando alguna herramienta de modelado de datos como ERWin o Rational Rose. Preste especial atención a la forma de modelar la categoría PROPIETARIO en este diagrama. (*Consejo*: Considere usar PROPIEDAD_DE_EMPRESA y una PROPIEDAD_ DE_PERSONA como dos tipos de relación diferentes).

**4.30.** Considere la base de datos UNIVERSIDAD descrita en el Ejercicio 3.16. En la Práctica 3.31 realizó un esquema ER para la misma mediante herramientas de modelado como ERWin o Rational Rose. Modifique este diagrama clasificando un CURSO como CURSO_BÁSICO o CURSO_SUPERIOR y un INSTRUCTOR como PROFESOR_AGREGADO o PROFESOR_TITULAR. Incluya los atributos apropiados para estos nuevos tipos de entidades. A continuación, establezca las relaciones necesarias para que el profesor agregado sea el encargado de impartir los cursos básicos, mientras que los titulares se encarguen de los superiores.

## Bibliografía seleccionada

Son muchos los artículos que han propuesto modelos de datos semánticos o conceptuales. A continuación le ofrecemos una lista representativa de los mismos. Un grupo de ellos, entre los que se incluyen Abrial (1974), el modelo DIAM de Senko (1975), el método NIAM (Verheijen y VanBekkum 1982), y Bracchi y otros (1976), presentan los modelos semánticos que están basados en el concepto de relaciones binarias. Un segundo grupo más moderno aborda los métodos para extender el modelo relacional y mejorar sus posibilidades.

Entre ellos se incluyen los de Schmid y Swenson (1975), Navathe y Schkolnick (1978), el modelo RM/T de Codd (1979), Furtado (1978) y el modelo estructural de Wiederhold y Elmasri (1979).

El modelo ER fue propuesto originalmente por Chen (1976) y formalizado en Ng (1981). Desde entonces se han propuesto numerosas extensiones, como la de Scheuermann y otros (1979), Dos Santos y otros (1979), Teorey y otros (1986), Gogolla y Hohenstein (1991) y el modelo ECR (Entidad-categoría-relación, *Entity-Category-Relationship*) de Elmasri y otros (1985). Smith y Smith (1977) presentan los conceptos de generalización y agregación. El modelo de datos semántico de Hammer y McLeod (1981) presentó los conceptos de entramados clase/subclase, así como conceptos de modelado avanzados.

En Hull y King (1987) aparece un sondeo sobre el modelado de datos semántico. Eick (1991) plantea el diseño y las transformaciones de los esquemas conceptuales. Los análisis de las restricciones para *n* relaciones aparece en Soutou (1998). El UML se muestra con detalle en Booch, Rumbaugh y Jacobson (1999). Fowler y Scott (2000), junto con Stevens y Pooley (2000), ofrecen una introducción concisa a los conceptos UML.

Fensel (2000, 2003) habla sobre la Semantic Web y la aplicación de ontologías. Uschold y Gruninger (1996), junto con Gruber (1995), abordan las ontologías. El número de junio de 2002 de *Communications of the ACM* está dedicado a los conceptos y las aplicaciones de la ontología. Fensel (2003) es un libro que trata las ontologías y el comercio electrónico.

# PARTE 2

## Modelo relacional: conceptos, restricciones, lenguajes, diseño y programación

CAPÍTULO 5

# El modelo de datos relacional y las restricciones de una base de datos relacional

Este capítulo abre la Parte 2, dedicada a las bases de datos relacionales. El modelo relacional fue presentado por primera vez por Ted Codd, de IBM Research, en 1970 en un documento ya clásico (Codd 1970), y atrajo la atención inmediatamente debido a su simplicidad y fundamentación matemática. El modelo utiliza el concepto de una *relación matemática* (algo parecido a una tabla de valores) como su bloque de construcción básico, y tiene su base teórica en la teoría de conjuntos y la lógica del predicado de primer orden. En este capítulo abordaremos las características básicas del módulo y sus restricciones.

Las primeras implementaciones comerciales del modelo relacional, como SQL/DS del sistema operativo MVS de IBM y Oracle DBMS, estuvieron disponibles a principios de los años 80. Desde entonces, ha sido implementado en otros muchos. Los DBMS relacionales más populares en la actualidad (los RDBMS) son DB2 e Informix Dynamic Server (de IBM), Oracle y Rdb (de Oracle) y SQL Server y Access (de Microsoft).

Debido a la importancia del modelo relacional, toda la Parte 2 de este libro está dedicada a él y a sus lenguajes asociados. El Capítulo 6 trata sobre las operaciones del álgebra relacional e introduce la notación de los cálculos relacionales de dos de sus tipos: tupla y dominio. El Capítulo 7 se centra en las estructuras de datos del modelo relacional para la construcción de modelos ER y EER, y muestra algoritmos para el diseño de un esquema de base de datos relacional asociando el esquema conceptual de los modelos ER o EER (consulte los Capítulos 3 y 4) con una representación relacional. Estas asociaciones están incorporadas en muchas herramientas CASE[1] y de diseño de bases de datos.

En el Capítulo 8 se describe el lenguaje de consultas SQL, que es el *estándar* de los DBMS relacionales comerciales. El Capítulo 9 está dedicado a las técnicas de programación empleadas para acceder a los sistemas de bases de datos y la notación para conectar con ellas a través de los protocolos estándar ODBC y JDBC. Los Capítulos 10 y 11, en la Parte 3, presentan otros aspectos del modelo relacional, las restricciones formales de las dependencias funcionales y multivalor; estas dependencias se utilizan para desarrollar una teoría de diseño de base de datos relacional conocida como *normalización.*

Entre los modelos de datos anteriores al relacional podemos citar los jerárquicos y de red. Fueron propuestos a principios de los 60 e implementados posteriormente en las primeras versiones de DBMS a finales de la misma década y principios de la siguiente. Debido a su importancia histórica y al gran número de usuarios de estos DBMS, hemos incluido un resumen de los puntos más importantes de estos modelos en los apéndices,

[1] CASE son las siglas de Ingeniería de software asistida por computador (*Computer-Aided Software Engineering*).

los cuales están disponibles en el sitio web de este libro (www.librosite.net/elmasri). Todos ellos se seguirán utilizando durante muchos años, por lo que ahora se los conoce como *sistemas de bases de datos heredados.*

En este capítulo nos concentraremos en la descripción de los principios básicos del modelo de datos relacional. Empezaremos en la Sección 5.1 definiendo los conceptos de modelado y notación del modelo relacional. La Sección 5.2 está dedicada a las restricciones relacionales, las cuales están consideradas en la actualidad como una parte importante del modelo y están impuestas en la mayoría de DBMS. La Sección 5.3 define las operaciones de actualización del modelo relacional, el modo de manipular las violaciones de las restricciones de integridad e introduce el concepto de una transacción.

# 5.1 Conceptos del modelo relacional

El modelo relacional representa la base de datos como una colección de *relaciones.* Informalmente, cada una de estas relaciones se parece a una tabla de valores o, de forma algo más extensa, a un fichero *plano* de registros. Por ejemplo, la base de datos de ficheros mostrada en la Figura 1.2 es similar a la representación del modelo relacional. Sin embargo, existen importantes diferencias entre las relaciones y los ficheros planos, como veremos muy pronto.

Cuando una relación está pensada como una **tabla** de valores, cada fila representa una colección de valores relacionados. En el Capítulo 3 se presentaron los tipos de entidad y de relación como conceptos para el modelado de datos reales. En el modelo relacional, cada fila de la tabla representa un hecho que, por lo general, se corresponde con una relación o entidad real. El nombre de la tabla y de las columnas se utiliza para ayudar a interpretar el significado de cada uno de los valores de las filas. Por ejemplo, la primera tabla de la Figura 1.2 se llama ESTUDIANTE porque cada fila representa la información de un estudiante particular. Los nombres de columna (Nombre, NumEstudiante, Clase y Especialidad) especifican el modo de interpretar los valores de cada fila en función de la columna en la que se encuentren. Todos los datos de una columna son del mismo tipo de dato.

En la terminología formal del modelo relacional, una fila recibe el nombre de *tupla,* una cabecera de columna es un *atributo* y el nombre de la tabla una *relación.* El tipo de dato que describe los valores que pueden aparecer en cada columna está representado por un *dominio* de posibles valores. Ahora pasaremos a describir con más detalle todos estos términos.

## 5.1.1 Dominios, atributos, tuplas y relaciones

Un **dominio** *D* es un conjunto de valores atómicos. Por **atómico** queremos decir que cada valor de un dominio es indivisible en lo que al modelo relacional se refiere. Una forma habitual de especificar un dominio es indicar un tipo de dato desde el que se dibujan los valores del mismo. También resulta útil darle un nombre que ayude en la interpretación de sus valores. Los siguientes son algunos ejemplos de dominios:

- NumerosTelefonosFijos. El conjunto de los 9 dígitos que componen los números de teléfono en España.
- NumerosTelefonosMoviles. El conjunto de los 9 dígitos que componen los números de teléfono móviles en España.
- DocumentoNacionalIdentidad. El conjunto de documentos nacionales de identidad (DNI) válidos en España.
- Nombres. El conjunto de caracteres que representan el nombre de una persona.
- MediaNotasCurso. Los posibles valores obtenidos al calcular la media de las notas obtenidas por un alumno a lo largo del curso. Debe ser un valor en punto flotante comprendido entre 1 y 10.
- EdadesEmpleado. Las posibles edades de los empleados de una empresa; cada una debe estar comprendida entre 16 y 80.

- NombresDepartamentosAcademicos. El conjunto de nombres de los departamentos académicos de una universidad, como Informática, Económicas o Física.
- CodigosDepartamentosAcademicos. El conjunto de códigos de los departamentos, como 'INF', 'ECON' y 'FIS'.

Lo expuesto anteriormente se conoce como definiciones *lógicas* de dominios. Para cada uno de ellos se especifica también un **tipo de dato** o **formato**. Por ejemplo, el tipo de datos del dominio Numeros-TelefonosFijos puede declararse como una cadena de caracteres de la forma *ddddddddd*, donde cada *d* es un dígito numérico (decimal) y los dos, o tres, primeros especifican la provincia del número. El tipo de datos para EdadesEmpleado es un número entero comprendido entre 16 y 80, mientras que para Nombres-DepartamentosAcademicos, es el conjunto de todas las cadenas de caracteres que representen los nombres de departamento válidos. Un dominio cuenta, por tanto, con un nombre, un tipo de dato y un formato. También puede facilitarse información adicional para la interpretación de sus valores; por ejemplo, un dominio numérico como PesoPersona debería contar con las unidades de medida, como kilogramos o libras.

Un **esquema de relación**[2] $R$, denotado por $R(A_1, A_2, \ldots, A_n)$, está constituido por un nombre de relación $R$ y una lista de atributos $A_1, A_2, \ldots, A_n$. Cada **atributo** $A_i$ es el nombre de un papel jugado por algún dominio $D$ en el esquema de relación $R$. Se dice que $D$ es el **dominio** de $A_i$ y se especifica como **dom**($A_i$). Un esquema de relación se utiliza para *describir* una relación; se dice que $R$ es el **nombre** de la misma. El **grado** (o ***arity***) de una relación es el número de atributos $n$ de la misma.

El siguiente es un ejemplo de esquema de relación de siete niveles (describe los estudiantes de una universidad):

ESTUDIANTE(Nombre, Dni, TlfParticular, Dirección, TlfTrabajo, Edad, Mnc)

Usando los tipos de datos de cada atributo, la definición aparece escrita a veces como:

ESTUDIANTE(Nombre: cadena, Dni: cadena, TlfParticular: cadena, Dirección: cadena,
TlfTrabajo: cadena, Edad: entero, Mnc: real)

En este esquema de relación, ESTUDIANTE es el nombre de la misma y cuenta con siete atributos. En la definición de más arriba mostramos una asignación de tipos genéricos a los atributos como *cadena* o *entero*. Basándonos en los ejemplos de dominios mostrados anteriormente, estos son los que se corresponden con alguno de los atributos de la relación ESTUDIANTE: dom(Nombre) = Nombres; dom(Dni) = Documento-NacionalIdentidad; dom(TlfParticular) = NumerosTelefonosFijos,[3] dom(TlfTrabajo) = NumerosTelefonosFijos y dom(Mnc) = MediaNotasCurso. Es posible referirse también a los atributos de una relación por su posición dentro de la misma; así, el segundo atributo de ESTUDIANTE es Dni, mientras que el cuarto es Dirección.

Una **relación** (o **estado de relación**)[4] $r$ del esquema $R(A_1, A_2, \ldots, A_n)$, también especificado como $r(R)$, es un conjunto de $n$-tuplas $r = \{t_1, t_2, \ldots, t_m\}$. Cada **tupla** $t$ es una lista ordenada de $n$ valores $t = <v_1, v_2, \ldots, v_n>$, donde $v_i$, $1 \leq i \leq n$, es un elemento de dom($A_i$) o un valor especial NULL (los valores NULL se tratan más adelante y en la Sección 5.1.2). El $i$-enésimo valor de la tupla $t$, que se corresponde con el atributo $A_i$, se referencia como $t[A_i]$ (o $t[i]$ si utilizamos una notación posicional). Los términos **intensidad de la relación** para el esquema $R$ y **extensión de relación** del estado $r(R)$ son también muy utilizados.

La Figura 5.1 muestra un ejemplo de una relación ESTUDIANTE que se corresponde con el esquema del mismo nombre recién especificado. Cada tupla de la relación representa a un estudiante en particular.

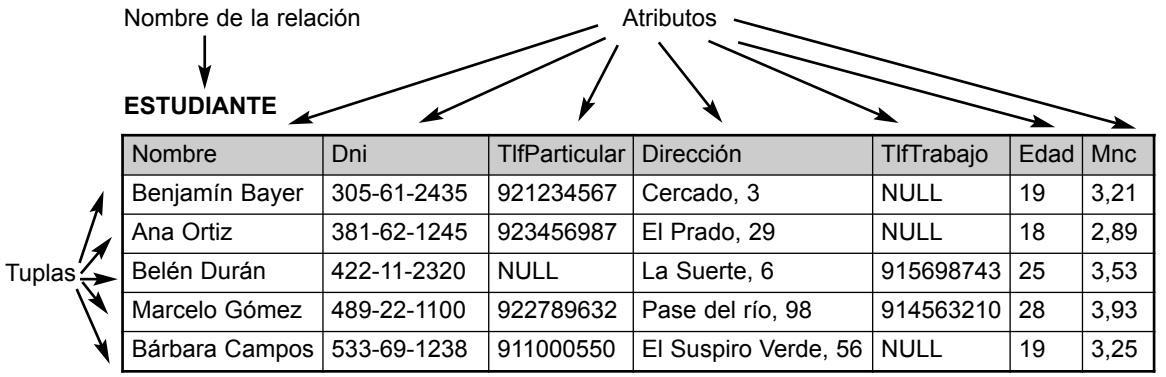
---

[2] Un esquema de relación recibe a veces el nombre de **diseño de relación**.

[3] Debido a que casi todo el mundo dispone de un teléfono móvil, podría usarse también NumerosTelefonosMoviles como dominio.

[4] Este concepto también ha recibido el nombre de **instancia de relación**, aunque no utilizaremos este término porque se emplea también para hacer referencia a una única tupla o fila.

**Figura 5.1.** Los atributos y tuplas de una relación ESTUDIANTE.




| Nombre | Dni | TlfParticular | Dirección | TlfTrabajo | Edad | Mnc |
|---|---|---|---|---|---|---|
| Benjamín Bayer | 305-61-2435 | 921234567 | Cercado, 3 | NULL | 19 | 3,21 |
| Ana Ortiz | 381-62-1245 | 923456987 | El Prado, 29 | NULL | 18 | 2,89 |
| Belén Durán | 422-11-2320 | NULL | La Suerte, 6 | 915698743 | 25 | 3,53 |
| Marcelo Gómez | 489-22-1100 | 922789632 | Pase del río, 98 | 914563210 | 28 | 3,93 |
| Bárbara Campos | 533-69-1238 | 911000550 | El Suspiro Verde, 56 | NULL | 19 | 3,25 |

Mostramos la relación como una tabla en la que cada tupla aparece como una *fila* y cada atributo como un *encabezamiento de columna* que indica la interpretación que habrá que dar a cada uno de los valores de la misma. Los *valores NULL* representan atributos cuyos valores no se conocen, o no existen, para una tupla ESTUDIANTE individual.

La anterior definición de relación puede ser *enunciada* más formalmente del siguiente modo. Una relación (o estado de relación) $r(R)$ es una **relación matemática** de grado $n$ en los dominios $\text{dom}(A_1)$, $\text{dom}(A_2)$, . . . , $\text{dom}(A_n)$ que es un **subconjunto** del **producto cartesiano** de los dominios que definen $R$:

$$r(R) \subseteq (\text{dom}(A_1) \times \text{dom}(A_2) \times \ldots \times \text{dom}(A_n))$$

El producto cartesiano especifica todas las posibles combinaciones de valores de los dominios subyacentes. Por tanto, si especificamos el número total de valores, o **cardinalidad,** de un dominio $D$ como $|D|$ (asumiendo que todos ellos son finitos), el número total de tuplas del producto cartesiano es:

$$|\text{dom}(A_1)| \times |\text{dom}(A_2)| \times \ldots \times |\text{dom}(A_n)|$$

Este producto de cardinalidades de todos los dominios representa el número total de posibles instancias, o tuplas, que pueden existir en la relación $r(R)$. De todas estas posibles combinaciones, un estado de relación en un momento dado (el **estado de relación actual**) sólo refleja las tuplas válidas que representan un estado particular del mundo real. En general, a medida que varía el estado del mundo real lo hace también la relación, convirtiéndose en otro estado de relación diferente. Sin embargo, el esquema $R$ es relativamente estático y *no* cambia salvo en circunstancias excepcionales (por ejemplo, como resultado de añadir un atributo para representar nueva información que no se encontraba originalmente en la relación). Es posible para varios atributos *tener el mismo dominio.* Los atributos indican diferentes **papeles**, o interpretaciones, para el dominio. Por ejemplo, en la relación ESTUDIANTE, NumerosTelefonosFijos juega el papel de TlfParticular y TlfTrabajo.

## 5.1.2 Características de las relaciones

La definición anterior de relación implica ciertas características que la hace diferente de un fichero o una tabla. Vamos a ver a continuación algunas de esas características.

**Ordenación de tuplas en una relación.** Una relación está definida como un *conjunto* de tuplas. Matemáticamente, los elementos de un conjunto *no guardan un orden* entre ellos; por tanto, las tuplas en una relación tampoco la tienen. En otras palabras, una relación no es sensible al ordenamiento de las tuplas. Sin embargo, en un fichero, los registros están almacenados físicamente en el disco (o en memoria), por lo que siempre hay establecido un orden entre ellos. De forma análoga, cuando mostramos una relación como una tabla, las filas aparecen con un cierto orden.

**Figura 5.2.** La relación ESTUDIANTE de la Figura 5.1 con un orden de tuplas distinto.

**ESTUDIANTE**

| Nombre | Dni | TlfParticular | Dirección | TlfTrabajo | Edad | Mnc |
|---|---|---|---|---|---|---|
| Belén Durán | 422-11-2320 | NULL | La Suerte, 6 | 915698743 | 25 | 3,53 |
| Bárbara Campos | 533-69-1238 | 911000550 | El Suspiro Verde, 56 | NULL | 19 | 3,25 |
| Marcelo Gómez | 489-22-1100 | 922789632 | Pase del río, 98 | 914563210 | 28 | 3,93 |
| Ana Ortiz | 381-62-1245 | 923456987 | El Prado, 29 | NULL | 18 | 2,89 |
| Benjamín Bayer | 305-61-2435 | 921234567 | Cercado, 3 | NULL | 19 | 3,21 |

**Figura 5.3.** Dos tuplas idénticas cuando el orden de los atributos y los valores no forma parte de la definición de relación.

*t* = <(Nombre, Belén Durán),(Dni, 422-11-2320),(TlfParticular, NULL),(Dirección, La Suerte, 6),(TlfTrabajo, 915698743),(Edad, 25),(Mnc, 3,53)>

*t* = <(Dirección, La Suerte, 6),(Nombre, Belén Durán),(Dni, 422-11-2320),(Edad, 25),(TlfTrabajo, 915698743),(Mnc, 3,53),(TlfParticular, NULL)>

---

La ordenación de las tuplas no forma parte de la definición de una relación porque ésta intenta representar hechos a un nivel lógico o abstracto. En una relación pueden especificarse muchos órdenes. Por ejemplo, las tuplas en la relación ESTUDIANTE de la Figura 5.1 podrían ordenarse lógicamente por Nombre, Dni, Edad o cualquier otro atributo. La definición de una relación no especifica ningún orden: *no hay una preferencia* por un orden con respecto a otro. Por consiguiente, las relaciones de las Figuras 5.1 y 5.2 se consideran *idénticas*. Cuando se implementa una relación como un fichero, o se muestra como una tabla, es posible especificar un orden en los registros del primero o las filas del segundo.

**Ordenación de los valores dentro de una tupla y definición alternativa de una relación.** Según la definición anterior de relación, una *n*-tupla es una *lista ordenada de n* valores, por lo que el orden de valores dentro de una de ellas (y por consiguiente de los atributos de un esquema de relación) es importante. Sin embargo, a nivel lógico, el orden de los atributos y sus valores *no* es tan importante mientras se mantenga la correspondencia entre ellos.

Puede darse una **definición alternativa** de una relación que haría *innecesaria* la ordenación de los valores de una tupla. En esta definición, un esquema de relación $R = \{A_1, A_2, \ldots, A_n\}$ es un *conjunto* de atributos, y un estado de relación $r(R)$ es un conjunto finito de asignaciones $r = \{t_1, t_2, \ldots, t_m\}$, en el que cada tupla $t$ es una **asociación** desde $R$ hacia $D$, y $D$ es la unión de los dominios de atributos; esto es, $D = \text{dom}(A_1) \cup \text{dom}(A_2) \cup \ldots \cup \text{dom}(A_n)$. En esta definición, $t[A_i]$ debe estar en $\text{dom}(A_i)$ para $1 \leq i \leq n$ para cada asignación $t$ en $r$. La asignación $t_i$ recibe el nombre de tupla.

Según esta definición de tupla como una asignación, una **tupla** puede considerarse como un **conjunto** de parejas (<atributo>, <valor>), donde cada una de ellas proporciona el valor de la asignación de un atributo $A_i$ a un valor $v_i$ de $\text{dom}(A_i)$. La ordenación de atributos *no* es importante ya que el nombre del mismo aparece con su valor. Según esta definición, las dos tuplas de la Figura 5.3 son idénticas. Esto tiene sentido a nivel abstracto o lógico, ya que no hay ningún motivo para preferir que una pareja aparezca antes que otra en una tupla.

Cuando se implementa una relación como un fichero, los atributos están ordenados físicamente como campos dentro de un registro. Por lo general, usaremos la **primera definición** de relación, en la que los atributos y los

valores de dentro de las tuplas *están ordenados,* porque simplifica mucho la notación. Sin embargo, la definición alternativa ofrecida aquí es mucho más general.[5]

**Valores y NULLs en las tuplas.** Cada valor en una tupla es un valor **atómico**, es decir, no es divisible en componentes dentro del esqueleto del modelo relacional básico. Por tanto, no están permitidos los atributos compuestos y multivalor (consulte el Capítulo 3). Este modelo suele recibir a veces el nombre de **modelo relacional plano**. Una gran parte de la teoría que se esconde tras el modelo relacional fue desarrollada con este principio en mente, el cual recibe el nombre de principio de **primera forma normal**.[6] Así pues, los atributos multivalor deben representarse en relaciones separadas, mientras que los compuestos lo están sólo por sus atributos de componente simple en el modelo relacional básico.[7]

Un concepto importante es el de los valores NULL (nulo), que se utilizan para representar los valores de atributos que pueden ser desconocidos o no ser aplicables a una tupla. Para estos casos, existe un valor especial llamado NULL. Por ejemplo, en la Figura 5.1, algunas tuplas ESTUDIANTE tienen NULL en sus números de teléfono del trabajo porque no cuentan con una oficina donde localizarlos. Otros lo tienen en el teléfono particular, presumiblemente porque no cuenten con un terminal en su casa o no lo conozcan (el valor es *desconocido*). En general, los valores NULL pueden tener varios significados, como lo de *valor desconocido*, *valor existente pero no disponible* o *atributo no aplicable a esta tupla*. Un ejemplo de este último caso ocurrirá si añadimos un atributo EstadoVisado a la relación ESTUDIANTE que sólo sea aplicable a los estudiantes extranjeros. Es posible idear diferentes códigos para cada uno de los significados de NULL.

La incorporación de diferentes tipos de valores NULL en las operaciones de un modelo relacional (consulte el Capítulo 6) se ha demostrado que es compleja y está fuera del alcance de nuestra presentación.

Los valores NULL se originaron por las razones comentadas anteriormente (datos indefinidos, desconocidos o no presentes en un determinado momento). El significado exacto de un valor NULL controla la forma en que se comporta durante las operaciones de agregación o comparación aritmética con otros valores. Por ejemplo, una comparación de dos valores NULL provoca ambigüedades: si los clientes A y B tienen direcciones NULL, ¿significa esto que ambos la comparten? Durante el diseño de una base de datos es preferible evitar los valores NULL tanto como sea posible. Los trataremos de nuevo en los Capítulos 6 y 8 en el contexto de operaciones y consultas, y en el 10 en relación con el diseño.

**Interpretación (significado) de una relación.** El esquema de relación puede interpretarse como una declaración o un tipo de **aserción**. Por ejemplo, el esquema de la relación ESTUDIANTE de la Figura 5.1 afirma que, en general, una entidad estudiante tiene un Nombre, un Dni, un TlfParticular, una Dirección, un TlfTrabajo, una Edad y una Mnc. Cada tupla de la relación puede ser interpretada entonces como un **hecho** o una instancia particular de la aserción. Por ejemplo, la primera tupla de la Figura 5.1 asevera el hecho de que hay un ESTUDIANTE cuyo Nombre es Benjamín Bayer, su Dni es 305612435, su Edad es 19, etc.

Observe que algunas relaciones pueden representar hechos sobre *entidades,* mientras que otras pueden hacerlo sobre *relaciones.* Por ejemplo, un esquema de relación ESPECIALIDAD (DniEstudiante, CodDpto) afirma que los estudiantes están especializados en disciplinas académicas. Una tupla en esta relación une un estudiante con su especialidad. Por tanto, el modelo relacional representa hechos sobre entidades y relaciones uniformemente como relaciones. Esto puede comprometer a veces la comprensibilidad porque se tiene que adivinar si una relación representa un tipo de entidad o de relación. Los procedimientos de asignación del Capítulo 7 muestran cómo diferentes construcciones de los modelos ER y EER logran convertirse en relaciones.

---

[5] Como veremos, la definición alternativa de relación será útil cuando tratemos el procesamiento de sentencias en los Capítulos 15 y 16.

[6] Tratamos con más detalle este supuesto en el Capítulo 10.

[7] Las extensiones del modelo relacional eliminan estas restricciones. Por ejemplo, los sistemas objeto-relacional permiten atributos complejos estructurados, como hace los modelos relacionales **forma normal no-primera** o **anidados**, tal y como veremos en el Capítulo 22.

Una interpretación alternativa de un esquema de relación es como un **predicado**; en este caso, los valores de cada tupla se representan como valores que *satisfacen* el predicado. Por ejemplo, el predicado ESTUDIANTE (Nombre, Dni, . . . ) se cumple para las cinco tuplas de la relación ESTUDIANTE de la Figura 5.1. Estas tuplas representan cinco proposiciones o hechos diferentes del mundo real. Esta interpretación es poco útil en el contexto de la lógica de los lenguajes de programación, como Prolog, porque permite que el modelo relacional sea usado dentro de esos lenguajes (consulte la Sección 24.4). La **suposición del mundo cerrado** afirma que los únicos hechos ciertos en el universo son aquéllos presentes dentro de la extensión de la relación o relaciones. Cualquier otra combinación de valores hace que el predicado sea falso.

### 5.1.3 Notación del modelo relacional

Usaremos la siguiente notación en nuestra presentación:

- Un esquema de relación $R$ de grado $n$ se designa como $R(A1, A2, \ldots, An)$.
- Las letras $Q$, $R$, $S$ especifican nombres de relación.
- Las letras $q$, $r$, $s$ especifican estados de relación.
- Las letras $t$, $u$, $v$ indican tuplas.
- En general, el nombre de una relación como ESTUDIANTE indica también el conjunto real de tuplas de la misma (el *estado actual de la relación*) mientras que ESTUDIANTE(Nombre, Dni, . . .) se refiere sólo a su esquema.
- Un atributo $A$ puede cualificarse con el nombre de relación $R$ al cual pertenece usando la notación de punto, $R.A$: por ejemplo, ESTUDIANTE.Nombre o ESTUDIANTE.Edad. Esto es así porque dos atributos en relaciones diferentes pueden usar el mismo nombre. Sin embargo, todos los nombres de atributo en una *relación particular* deben ser distintos.
- Una $n$-tupla $t$ en una relación $r(R)$ está designada por $t = <v_1, v_2, \ldots, v_n>$, donde $v_i$ es el valor correspondiente al atributo $A_i$. La siguiente notación se refiere a los **valores componente** de las tuplas:
  - Tanto $t[A_i]$ como $t.A_i$ (y, a veces, $t[_i]$) hacen referencia al valor $v_i$ de $t$ del atributo $A_i$.
  - Tanto $t[A_u, A_w, \ldots, A_z]$ como $t.(A_u, A_w, \ldots, A_z)$, donde $A_u, A_w, \ldots, A_z$ es una lista de atributos de $R$, hacen referencia a la subtupla de valores $<v_u, v_w, \ldots, v_z>$ de $t$ correspondientes a los atributos especificados en la lista.

Como ejemplo, considere la tupla $t$ = <‘Bárbara Campos’, ‘533-69-1238', ‘911000550’, ‘El Suspiro Verde, 56’, NULL, 19, 3.25> de la relación ESTUDIANTE de la Figura 5.1; tenemos que $t$ [Nombre] = <‘Barbara Campos’>, y $t$ [Dni, Mnc, Edad] = <‘14875693', 3.25, 19>.

# 5.2 Restricciones del modelo relacional y esquemas de bases de datos relacionales

Hasta el momento hemos estudiado las características de las relaciones sencillas. En una base de datos relacional, existirán por lo general muchas relaciones, y las tuplas de las mismas estarán relacionadas de muy diferentes formas. El estado de toda la base de datos se corresponderá con los estados de todas sus relaciones en un momento de tiempo concreto. Generalmente, existen muchas **restricciones**, o *constraints*, en los valores de un estado de base de datos. Estas restricciones están derivadas de las reglas del minimundo que dicha base de datos representa, tal y como vimos en la Sección 1.6.8.

En esta sección, vamos a estudiar las diversas restricciones de datos que pueden especificarse en una base de datos relacional. Éstas pueden dividirse generalmente en tres categorías principales:

1. Restricciones que son inherentes al modelo de datos y que reciben el nombre de **restricciones implícitas** o **inherentes basadas en el modelo**.
2. Restricciones que pueden expresarse directamente en los esquemas del modelo de datos, por lo general especificándolas en el DDL (Lenguaje de definición de datos, *Data Definition Language*; consulte la Sección 2.3.1). Las llamaremos **restricciones explícitas** o **basadas en el esquema**.
3. Restricciones que no pueden expresarse directamente en los esquemas del modelo de datos, y que por consiguiente deben ser expresadas e implementadas por los programas. Las llamaremos **restricciones semánticas, basadas en aplicación** o **reglas de negocio**.

Las características de las relaciones que hemos tratado en la Sección 5.1.2 son las restricciones inherentes del modelo relacional y pertenecen a la primera categoría; por ejemplo, la restricción de que una relación no puede tener tuplas duplicadas en una restricción inherente. Las restricciones que tratamos en esta sección son las de la segunda categoría, es decir, las que pueden ser expresadas en el esquema del modelo relacional a través del DDL. Las de tercera categoría son más generales, están relacionadas con el significado y con el comportamiento de los atributos, y son difíciles de expresar e implementar dentro del modelo de datos, razón por la cual suelen comprobarse dentro de las aplicaciones.

Otra categoría importante de restricciones son las *dependencias de datos*, las cuales incluyen las *dependencias funcionales* y las *dependencias multivalor*. Suelen emplearse para comprobar la corrección del diseño de una base de datos relacional y en un proceso llamado *normalización*, del cual hablamos en los Capítulos 10 y 11.

Vamos a ver los tipos de restricciones principales que pueden aplicarse en el modelo relacional: las basadas en esquema. Entre ellas se incluyen las de dominio, las de clave, las restricciones en valores NULL, las de integridad de entidad y las de integridad referencial.

## 5.2.1 Restricciones de dominio

Las restricciones de dominio especifican que dentro de cada tupla, el valor de un atributo $A$ debe ser un valor atómico del dominio dom($A$). En la Sección 5.1.1 ya hemos explicado las formas en las que pueden especificarse los dominios. Los tipos de datos asociados a ellos suelen incluir valores numéricos estándar para datos enteros (como entero corto, entero o entero largo) y reales (de coma flotante de simple y doble precisión). También están disponibles tipos de datos para el almacenamiento de caracteres, valores lógicos, cadenas de longitud fija y variable, fechas, horas y moneda. Es posible describir otros dominios como un subrango de valores de un tipo de dato, o como un tipo de dato enumerado en el que todos sus posibles valores están explícitamente listados. En lugar de describirlos aquí con detalle, en la Sección 8.1 abordaremos los que están contenidos en el estándar relacional SQL-99.

## 5.2.2 Restricciones de clave y restricciones en valores NULL

Una *relación* está definida como un *conjunto de tuplas*. Por definición, todos los elementos de un conjunto son distintos; por tanto, todas las tuplas en una relación también deben serlo. Esto significa que dos tuplas no pueden tener la misma combinación de valores para *todos* sus atributos. Habitualmente existen otros **subconjuntos de atributos** de una relación $R$ con la propiedad de que dos tuplas en cualquier relación $r$ de $R$ no deben tener la misma combinación de valores para estos atributos. Suponga que designamos uno de estos subconjuntos de atributos de SK; a continuación, para dos tuplas cualesquiera *distintas* $t1$ y $t2$ en una relación $r$ de $R$, tenemos la restricción:

$$t_1[\text{SK}] \neq t_2[\text{SK}]$$

Cualquier conjunto de atributos SK recibe el nombre de **superclave** del esquema de relación $R$. Una superclave especifica una *restricción de exclusividad* por la que dos tuplas distintas en cualquier estado $r$ de $R$ pue-

den tener el mismo valor para SK. Cada relación tiene, al menos, una superclave predeterminada: el conjunto de todos sus atributos. Sin embargo, una superclave puede contar con atributos redundantes, por lo que un concepto más importante es el de clave, que no tiene redundancia. Una **clave** *K* de un esquema de relación *R* es una superclave de *R* con la propiedad adicional que eliminando cualquier atributo *A* de *K* deja un conjunto de atributos *K'* que ya no es una superclave de *R*. Por tanto, una clave satisface dos restricciones:

1. Dos tuplas diferentes en cualquier estado de la relación no pueden tener valores idénticos para (todos) los atributos de la clave.
2. Es una *superclave mínima*, es decir, una superclave de la cual no podemos eliminar ningún atributo y seguiremos teniendo almacenada la restricción de exclusividad de la condición 1.

La primera condición se aplica tanto a las claves como a las superclaves, mientras que la segunda sólo a las claves. Por ejemplo, considere la relación ESTUDIANTE de la Figura 5.1. El conjunto de atributo {Dni} es una clave de ESTUDIANTE porque dos tuplas de estudiantes distintas no pueden tener el mismo valor para el Dni.[8] Cualquier conjunto de atributos que incluya el Dni (por ejemplo, {Dni, Nombre, Edad} es una superclave. Sin embargo, la superclave {Dni, Nombre, Edad} no es una clave de ESTUDIANTE porque la eliminación del Nombre, la Edad, o ambas, del conjunto aun no deja una superclave. En general, cualquier superclave formada a partir de un único atributo es también una clave. Una clave con múltiples atributos debe exigir *todos* ellos para mantener la condición de exclusividad.

El valor de un atributo clave puede usarse para identificar de forma única cada tupla en la relación. Por ejemplo, el Dni 30561243 identifica de forma inequívoca la tupla correspondiente a Benjamín Bayer en la relación ESTUDIANTE. Observe que un conjunto de atributos que constituyen una clave son una propiedad del esquema de relación; es una restricción que debe mantenerse en *cada* estado de relación válido del esquema. Una clave está determinada por el significado de los atributos, y la propiedad es *fija en el tiempo*: debe mantenerse cuando insertemos nuevas tuplas en la relación. Por ejemplo, no podemos, y no debemos, designar el atributo Nombre de la relación ESTUDIANTE de la Figura 5.1 como una clave porque es posible que dos estudiantes con nombres idénticos existieran en algún punto en un estado válido.[9]

En general, un esquema de relación puede contar con más de una clave. En este caso, cada una de ellas recibe el nombre de **clave candidata**. Por ejemplo, la relación COCHE de la Figura 5.4 tiene dos claves candidatas: NumeroPermisoConducir y NumeroBastidor. Es común designar una de ellas como la **clave principal** de la relación, y será la que se utilice para *identificar* las tuplas en la relación. Usamos la convención de que los atributos que forman la clave principal de un esquema de relación están subrayados, tal y como puede verse en la Figura 5.4. Observe que cuando una relación cuenta con varias claves candidatas, la elección de una de ellas como clave principal es algo arbitrario; sin embargo, es preferible elegir una que tenga un solo atributo, o un pequeño número de ellos.

Otra restricción en los atributos especifica si se permiten o no los valores NULL. Por ejemplo, si cada tupla ESTUDIANTE debe contar con un valor válido y no nulo para el atributo Nombre, entonces el Nombre de ESTUDIANTE esta obligado a ser NOT NULL.

### 5.2.3 Bases de datos relacionales y esquemas de bases de datos relacionales

Las definiciones y restricciones que hemos visto hasta ahora se aplican a las relaciones individuales y a sus atributos. Una base de datos relacional suele contener muchas relaciones, con tuplas que están relacionadas de diversas formas. En esta sección vamos a definir una base de datos relacional y un esquema del mismo

[8] Observe que el Dni es también una superclave.

[9] A veces se utilizan los nombres como claves, pero entonces se utilizan algunos subterfugios (como la incorporación de un número de orden) para distinguir entre dos nombres idénticos.

**Figura 5.4.** La relación COCHE, con dos claves candidatas: NumeroPermisoConducir y NumeroBastidor.

**COCHE**

| NumeroPermisoConducir | NumeroBastidor | Marca | Modelo | Año |
|---|---|---|---|---|
| Texas ABC-739 | A69352 | Ford | Mustang | 02 |
| Florida TVP-347 | B43696 | Oldsmobile | Cutlass | 05 |
| New Cork MPO-22 | X83554 | Oldsmobile | Delta | 01 |
| California 432-TFY | C43742 | Mercedes | 190-D | 99 |
| California RSK-629 | Y82935 | Toyota | Camry | 04 |
| Texas RSK-629 | U028365 | Jaguar | XJS | 04 |

tipo. Un **esquema de base de datos relacional** *S* es un conjunto de esquemas de relación S = {$R_1$, $R_2$, . . . , $R_m$} y de **restricciones de integridad** RI. Un **estado de base de datos relacional**[10] DB de *S* es un conjunto de estado de relación DB = {$r_1$, $r_2$, . . . , $r_m$} en el que cada $r_i$ es un estado de $R_i$ y satisface las restricciones de integridad especificadas en RI. La Figura 5.5 muestra un esquema de base de datos relacional que llamamos EMPRESA = {EMPLEADO, DEPARTAMENTO, LOCALIZACIONES_DPTO, PROYECTO, TRABAJA_EN, SUBORDINADO}. Los atributos subrayados representan las claves primarias. La Figura 5.6 muestra un estado de la base de datos que se corresponde con el esquema EMPRESA. Usaremos este esquema y el estado de base de datos a lo largo de este capítulo y en los que van del 6 al 9 para el desarrollo de consultas en diferentes lenguajes relacionales. En el sitio web de este libro puede encontrar algunos ejemplos (en inglés) que le servirán para realizar los ejercicios del final de los capítulos.

Cuando nos referimos a una base de datos relacional, incluimos implícitamente tanto su esquema como su estado actual. Un estado de base de datos que no cumple todas sus restricciones de integridad se dice que está en un **estado incorrecto**, mientras que aquél que sí las cumple está en un **estado correcto**.

En la Figura 5.5, el atributo NumDpto de DEPARTAMENTO y LOCALIZACIONES_DPTO representa el mismo concepto del mundo real: el número asignado a un departamento. Este mismo concepto recibe el nombre de Dno en EMPLEADO y NumDptoProyecto en PROYECTO. Los atributos que representan el mismo concepto del mundo real pueden tener o no los mismos nombres en relaciones diferentes. Por otro lado, los atributos que representan diferentes conceptos pueden tener el mismo nombre en relaciones distintas. Por ejemplo, podríamos haber usado el nombre de atributo Nombre tanto para el NombreProyecto de PROYECTO como para el NombreDpto de DEPARTAMENTO; en este caso, tendríamos dos atributos con el mismo nombre pero que representarían dos conceptos diferentes: nombres de proyecto y de departamento.

En algunas de las primeras versiones del modelo relacional, se presupuso que el mismo concepto del mundo real, cuando era representado por un atributo, debería tener *idéntico* nombre de atributo en todas las relaciones. Esto crea problemas cuando ese concepto se emplea en distintos papeles (significados) dentro de la misma relación. Por ejemplo, el concepto de Documento Nacional de Identidad aparece dos veces en la relación EMPLEADO de la Figura 5.5: una como el DNI del empleado y otra como el del supervisor. Les dimos distintos nombres de atributo (Dni y SuperDni, respectivamente) para distinguirlos.

Cada DBMS relacional debe tener un DDL para la definición del esquema de la base de datos relacional. Los DBMS relacionales actuales utilizan casi en su totalidad SQL para ello. Veremos este lenguaje en las Secciones de la 8.1 a la 8.3.

---

[10] Un *estado* de base de datos relacional suele recibir a veces el nombre de *instancia* de base de datos. Sin embargo, como ya se comentó anteriormente, no lo utilizaremos porque se aplica también a las tuplas individuales.

**Figura 5.5.** Diagrama del esquema de la base de datos relacional EMPRESA.

**EMPLEADO**

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|

**DEPARTAMENTO**

| NombreDpto | NumeroDpto | DniDirector | FechaIngresoDirector |
|---|---|---|---|

**LOCALIZACIONES_DPTO**

| NumeroDpto | UbicacionDpto |
|---|---|

**PROYECTO**

| NombreProyecto | NumProyecto | UbicacionProyecto | NumDptoProyecto |
|---|---|---|---|

**TRABAJA_EN**

| DniEmpleado | NumProy | Horas |
|---|---|---|

**SUBORDINADO**

| DniEmpleado | NombSubordinado | Sexo | FechaNac | Relación |
|---|---|---|---|---|

Las restricciones de integridad se especifican en un esquema de base de datos y deben cumplirse en cada estado válido de esa base de datos. Además del dominio, la clave y las restricciones NOT NULL, hay otros dos tipos de restricciones que forman parte del modelo relacional: la integridad de entidad y la referencial.

## 5.2.4 Integridad de entidad, integridad referencial y *foreign keys*

Las **restricciones de integridad de entidad** declaran que el valor de ninguna clave principal puede ser NULL. Esto se debe a que dicha clave se emplea para identificar tuplas individuales en una relación. Si se permitiera este valor, significaría que no se podrían identificar ciertas tuplas.

Por ejemplo, si dos o más tuplas tuvieran NULL en sus claves primarias, no seríamos capaces de diferenciarlas si intentásemos hacer referencia a ellas desde otras relaciones.

Las restricciones de clave y las de integridad de entidad se especifican en relaciones individuales. Las de **integridad referencial** están especificadas entre dos relaciones y se utilizan para mantener la consistencia entre las tuplas de dos relaciones. Informalmente, las restricciones de integridad referencial dicen que una tupla de una relación que hace referencia a otra relación debe hacer referencia a una *tupla existente* de esa relación. Por ejemplo, en la Figura 5.6, el atributo Dno de EMPLEADO devuelve el número de departamento en el que trabaja cada empleado; por tanto, su valor en cada tupla EMPLEADO debe coincidir con el valor NumeroDpto de alguna tupla de la relación DEPARTAMENTO.

Para expresar de un modo más formal la integridad referencial, primero debemos definir el concepto de una *foreign key* (clave externa). Las condiciones de una *foreign key*, dadas más abajo, especifican una restricción de integridad referencial entre dos esquemas de relación $R_1$ y $R_2$. Un conjunto de atributos FK en una relación $R_1$ es una ***foreign key*** de $R_1$ que **referencia** a la relación $R_2$ si satisface las siguientes reglas:

1. Los atributos en FK tienen el mismo dominio, o dominios, que los atributos de clave principal PK de $R_2$; se dice que los atributos FK **referencian** o **hacen referencia a** la relación $R_2$.
2. Un valor de FK en una tupla $t_1$ del estado actual $r_1(R_1)$ tampoco aparece como valor de PK en alguna tupla $t_2$ del estado actual $r_2(R_2)$ *o es NULL*. En el caso anterior, tenemos que $t_1[\text{FK}] = t_2[\text{PK}]$, y decimos que la tupla $t_1$ **referencia** o **hace referencia a** la tupla $t_2$.

En esta definición, $R_1$ recibe el nombre de **relación de referencia** y $R_2$ es **relación referenciada**. Si se mantienen ambas condiciones, se establece una **restricción de integridad referencial** de $R_1$ a $R_2$. En una base de datos de muchas relaciones, suelen existir muchas de estas restricciones.

Para declararlas, primero debemos tener muy claro el papel que juega cada conjunto de atributos en los distintos esquemas de relación de la base de datos. Las restricciones de integridad referencial suelen originarse a partir de las *relaciones entre las entidades* representadas por los esquemas de relación. Por ejemplo, considere la base de datos mostrada en la Figura 5.6. En la relación EMPLEADO, el atributo Dno hace referencia al departamento en el que éste trabaja; por consiguiente, designamos Dno como una *foreign key* de EMPLEADO que hace referencia a la relación DEPARTAMENTO. Esto implica que un valor de Dno en cualquier tupla $t_1$ de la relación EMPLEADO debe coincidir con otro de la clave principal de DEPARTAMENTO (el atributo NumeroDpto) en alguna tupla $t_2$ de la relación DEPARTAMENTO, o *puede ser NULL* si el empleado no pertenece a un departamento o será asignado más adelante. En la Figura 5.6, la tupla del empleado 'José Pérez' hace referencia a la tupla del departamento 'Investigación', lo que nos dice que 'José Pérez' trabaja en este departamento.

Observe que una *foreign key puede hacer referencia a su propia relación.* Por ejemplo, el atributo SuperDni de EMPLEADO se refiere al supervisor de un empleado, el cual es a su vez otro empleado representado por una tupla en la misma relación. Por tanto, SuperDni es una *foreign key* que enlaza con la propia relación EMPLEADO. En la Figura 5.6, la tupla de 'José Pérez' está unida a la de 'Alberto Campos, lo que indica que éste es el supervisor de aquél.

Podemos *mostrar en forma de diagrama las restricciones de integridad referencial* dibujando un arco que vaya desde cada *foreign key* a la relación a la que referencia. Para aclarar los términos, la punta de la flecha debe apuntar a la clave principal de la relación referenciada. La Figura 5.7 muestra el esquema de la Figura 5.5 con las restricciones de integridad referencial expresadas de este modo.

**Figura 5.6.** Un posible estado de base de datos para el esquema relacional EMPRESA.

**EMPLEADO**

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|
| José | Pérez | Pérez | 123456789 | 01-09-1965 | Eloy I, 98 | H | 30000 | 333445555 | 5 |
| Alberto | Campos | Sastre | 333445555 | 08-12-1955 | Avda. Ríos, 9 | H | 40000 | 888665555 | 5 |
| Alicia | Jiménez | Celaya | 999887777 | 12-05-1968 | Gran Vía, 38 | M | 25000 | 987654321 | 4 |
| Juana | Sainz | Oreja | 987654321 | 20-06-1941 | Cerquillas, 67 | M | 43000 | 888665555 | 4 |
| Fernando | Ojeda | Ordóñez | 666884444 | 15-09-1962 | Portillo, s/n | H | 38000 | 333445555 | 5 |
| Aurora | Oliva | Avezuela | 453453453 | 31-07-1972 | Antón, 6 | M | 25000 | 333445555 | 5 |
| Luis | Pajares | Morera | 987987987 | 29-03-1969 | Enebros, 90 | H | 25000 | 987654321 | 4 |
| Eduardo | Ochoa | Paredes | 888665555 | 10-11-1937 | Las Peñas, 1 | H | 55000 | NULL | 1 |

**DEPARTAMENTO**

| NombreDpto | NumeroDpto | DniDirector | FechaIngresoDirector |
|---|---|---|---|
| Investigación | 5 | 333445555 | 22-05-1988 |
| Administración | 4 | 987654321 | 01-01-1995 |
| Sede Central | 1 | 888665555 | 19-06-1981 |

**LOCALIZACIONES_DPTO**

| NumeroDpto | UbicacionDpto |
|---|---|
| 1 | Madrid |
| 4 | Gijón |
| 5 | Valencia |
| 5 | Sevilla |
| 5 | Madrid |

**Figura 5.6.** (*Continuación*).

**TRABAJA_EN**

| DniEmpleado | NumProy | Horas |
|---|---|---|
| 123456789 | 1 | 32,5 |
| 123456789 | 2 | 7,5 |
| 666884444 | 3 | 40,0 |
| 453453453 | 1 | 20,0 |
| 453453453 | 2 | 20,0 |
| 333445555 | 2 | 10,0 |
| 333445555 | 3 | 10,0 |
| 333445555 | 10 | 10,0 |
| 333445555 | 20 | 10,0 |
| 999887777 | 30 | 30,0 |
| 999887777 | 10 | 10,0 |
| 987987987 | 10 | 35,0 |
| 987987987 | 30 | 5,0 |
| 987654321 | 30 | 20,0 |
| 987654321 | 20 | 15,0 |
| 888665555 | 20 | NULL |

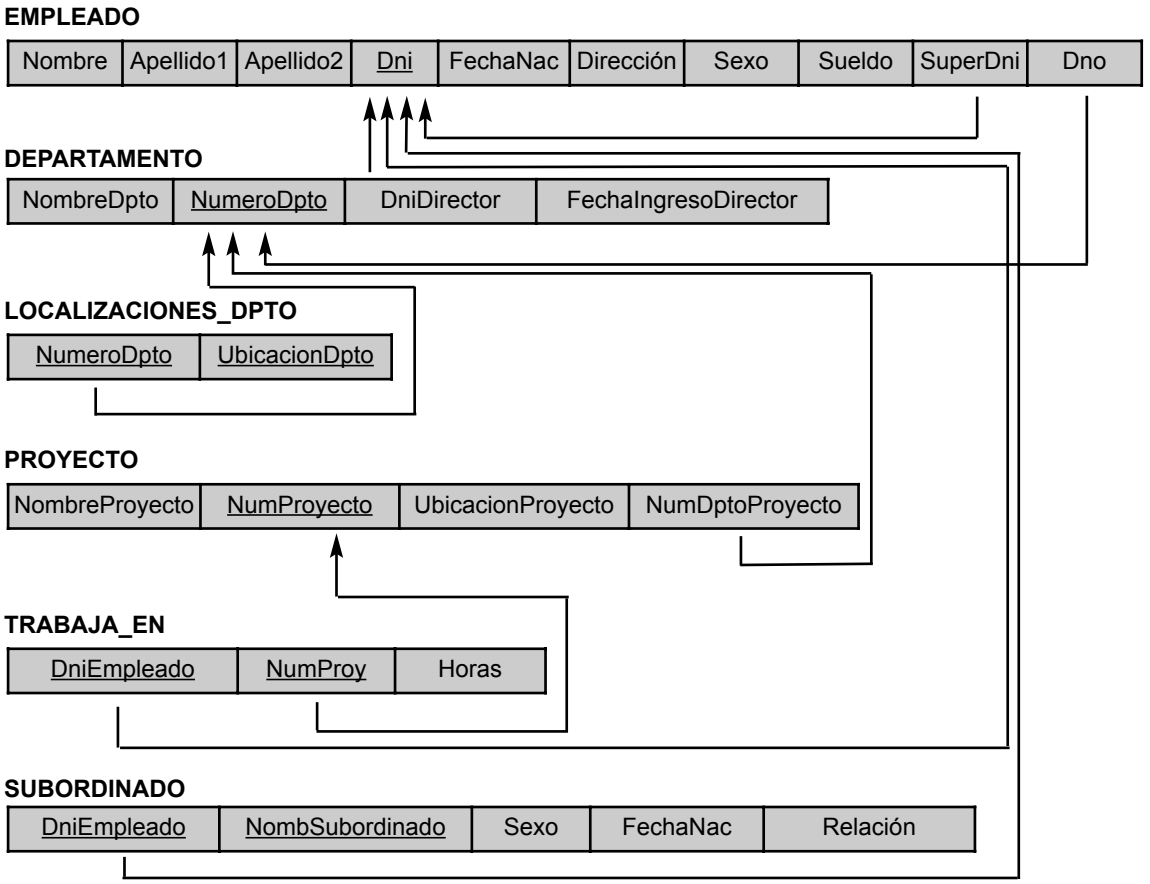
**PROYECTO**

| NombreProyecto | NumProyecto | UbicacionProyecto | NumDptoProyecto |
|---|---|---|---|
| ProductoX | 1 | Valencia | 5 |
| ProductoY | 2 | Sevilla | 5 |
| ProductoZ | 3 | Madrid | 5 |
| Computación | 10 | Gijón | 4 |
| Reorganización | 20 | Madrid | 1 |
| Comunicaciones | 30 | Gijón | 4 |

**SUBORDINADO**

| DniEmpleado | NombSubordinado | Sexo | FechaNac | Relación |
|---|---|---|---|---|
| 333445555 | Alicia | M | 05-04-1986 | Hija |
| 333445555 | Teodoro | H | 25-10-1983 | Hijo |
| 333445555 | Luisa | M | 03-05-1958 | Esposa |
| 987654321 | Alfonso | H | 28-02-1942 | Esposo |
| 123456789 | Miguel | H | 04-01-1988 | Hijo |
| 123456789 | Alicia | M | 30-12-1988 | Hija |
| 123456789 | Elisa | M | 05-05-1967 | Esposa |

**Figura 5.7.** Restricciones de integridad referencial mostradas en el esquema relacional EMPRESA.




Todas las restricciones de integridad deben estar especificadas en el esquema de una base de datos relacional (esto es, definida como parte de la definición) si queremos implementarlas en los estados de la misma. Así pues, el DDL ofrece la forma de especificar todas estas restricciones de forma que el DBMS pueda aplicarlas automáticamente. La mayoría de DBMSs relacionales soportan las restricciones de clave y de integridad de entidad, y hacen provisiones para ofrecer también las de integridad referencial. Todas estas restricciones se especifican como parte de la definición de los datos.

## 5.2.5 Otros tipos de restricciones

Entre las anteriores no está incluida una gran clase general de restricciones, llamadas a veces *restricciones de integridad semántica,* que pueden especificarse e implementarse en una base de datos relacional. Como ejemplos de ellas podemos citar la de que *el salario de un empleado no debe exceder el de su supervisor* y que *el número máximo de horas que un empleado puede trabajar a la semana es de 40*. Estas restricciones pueden implementarse dentro de las propias aplicaciones que actualizan la base de datos, o usar un **lenguaje de especificación de restricciones** de propósito general. Para ello existen unos mecanismos llamados ***triggers*** y **aserciones**. En SQL-99 se emplea una sentencia CREATE ASSERTION para este propósito (consulte el Capítulo 8). Debido a la complejidad y la dificultad en el uso de estos lenguajes de programación (tal y como se verá en la Sección 24.1), es más común comprobar este tipo de restricciones dentro del propio programa.

Existe otro tipo de restricción que es la de *dependencia funcional*, la cual establece una relación funcional entre dos conjuntos de atributos $X$ e $Y$. Esta restricción especifica que el valor de $X$ determina el de $Y$ en todos

los estados de una relación; está indicada como una dependencia funcional $X \rightarrow Y$. Usamos este tipo de dependencias, y otras más, en los Capítulos 10 y 11 como herramientas para analizar la calidad de los diseños de relación y para "normalizar" las relaciones que mejoran su calidad.

Las restricciones que hemos estudiado hasta ahora podrían llamarse **de estado** porque definen las restricciones que un *estado válido* de una base de datos debe satisfacer. Pueden definirse **restricciones de transición** para negociar con los cambios de estado de la base de datos.[11] Un ejemplo de este tipo de restricción es: "el sueldo de un empleado sólo puede aumentar". Como este tipo de restricción suele estar implementada en las aplicaciones o mediante reglas activas y *triggers*, las trataremos con detalle en la Sección 24.1.

# 5.3 Actualizaciones, transacciones y negociado de la violación de una restricción

Las operaciones del modelo relacional pueden clasificarse en *recuperaciones* y *actualizaciones*. Las operaciones de álgebra relacional, que puede usarse para especificar **recuperaciones**, se tratan con detalle en el Capítulo 6. Una expresión de álgebra relacional conforma una nueva relación una vez aplicados una serie de operadores algebraicos a un conjunto de relaciones ya existente; su uso principal es para preguntar a una base de datos. El usuario formula una consulta que especifica los datos que le interesan, tras lo cual se establece una nueva relación, aplicando los operadores relacionales para recuperar esos datos. Esa relación se convierte entonces en la respuesta a la pregunta del usuario. El Capítulo 6 presenta también el lenguaje llamado cálculo relacional, el cual se emplea para definir una nueva relación sin indicar un orden específico de operaciones.

En esta sección vamos a concentrarnos en las operaciones de **modificación** o **actualización** de una base de datos. Existen tres tipos de operaciones de actualización básicas: inserción, borrado y modificación. **Insert** se utiliza para insertar una nueva tupla o tuplas en una relación, **Delete** se encarga de borrarlas y **Update** (o **Modify**) cambia los valores de algunos atributos de las tuplas que ya existen. Siempre que se aplique cualquiera de estas operaciones, deberán respetarse las restricciones de integridad especificadas en el esquema de la base de datos.

En esta sección se tratan los tipos de restricciones que podrían violarse en cada operación de actualización, y las acciones que se deben tomar en cada caso. Usaremos como ejemplo la base de datos de la Figura 5.6, y sólo trataremos las restricciones de clave, las de integridad de entidad y las de integridad referencial mostradas en la Figura 5.7. Por cada tipo de actualización se mostrará algún ejemplo y se abordará cualquier restricción que dicha operación pudiera violar.

## 5.3.1 La operación Insert

**Insert** proporciona una lista de los valores de atributo para una nueva tupla $t$ que será insertada en una relación $R$. Esta operación puede violar cualquiera de las cuatro restricciones estudiadas en la sección anterior: las de dominio, si el valor dado a un atributo no aparece en el dominio correspondiente; las de clave, si el valor de dicha clave en la nueva tupla $t$ ya existe en otra tupla en la relación $r(R)$; las de integridad de entidad, si la clave principal de la nueva tupla $t$ es NULL; y las de integridad referencial, si el valor de cualquier *foreign key* en $t$ se refiere a una tupla que no exista en la relación referenciada. Aquí tiene algunos ejemplos que ilustran este debate.

- *Operación.*
  Insert <'Cecilia', 'Santos', 'García', NULL, '04-05-1960', 'Misericordia, 23', M, 28000, NULL, 4> into EMPLEADO.

[11] Las restricciones de estado suelen recibir a veces el nombre de restricciones estáticas, mientras que las de transición se conocen como *restricciones dinámicas*.

*Resultado*. Esta inserción viola la restricción de integridad de entidad (NULL para la clave principal Dni), por lo que es rechazada.

- *Operación*.

  Insert <'Alicia', 'Jiménez', 'Celaya', '999887777', '05-04-1960', 'Cercado, 38', M, 28000, '987654321', 4> into EMPLEADO.
  *Resultado*. Esta inserción viola la restricción de clave porque ya existe otra tupla en la relación EMPLEADO con el mismo valor de Dni, por lo que es rechazada.

- *Operación*.

  Insert <'Cecilia', 'Santos', 'García', '677678989', '04-05-1960', 'Misericordia, 23', M, 28000, '987654321', 7> into EMPLEADO.
  *Resultado*. Esta inserción viola la restricción de integridad referencial especificada en Dno en EMPLEADO porque no existe ningún DEPARTAMENTO cuyo NumeroDpto = 7.

- *Operación*.

  Insert <'Cecilia', 'Santos', 'García', '677678989', '04-05-1960', 'Misericordia, 23', M, 28000, NULL, 4> into EMPLEADO.
  *Resultado*. Esta inserción satisface todas las restricciones, por lo que es aceptada.

Si una inserción viola una o más restricciones, la opción predeterminada es *rechazarla.* En este caso, resultaría útil que el DBMS explicara el motivo de dicho rechazo. Otra posibilidad es intentar *corregir el motivo del rechazo*, aunque esto no suele ser muy habitual en inserciones, aunque sí en borrados y actualizaciones. En la primera operación antes mostrada, el DBMS podría solicitar al usuario un valor de Dni y aceptar la inserción en el caso de que éste fuera correcto. En la tercera operación, la base de datos podría sugerir al usuario el cambio de Dno por otro correcto (o establecerlo a NULL), o podría permitirle insertar una nueva tupla DEPARTAMENTO con un NumeroDpto = 7 y aceptar la inserción original, aunque sólo después de que ésta última fuera aceptada. Observe que en este último caso se podría producir una **vuelta atrás** en EMPLEADO si el usuario intenta insertar una tupla para el departamento 7 con un valor de DniDirector que no exista en EMPLEADO.

### 5.3.2 La operación Delete

**Delete** sólo puede violar la integridad referencial en caso de que la tupla a eliminar esté referenciada por las *foreign keys* de otras tuplas de la base de datos. Para especificar un borrado, una condición en los atributos de la relación es la que selecciona la tupla (o tuplas) a eliminar. Aquí tiene algunos ejemplos:

- *Operación*.

  Borrar la tupla TRABAJA_EN cuyo DniEmpleado = '999887777' y NumProy = 10.
  *Resultado.* Este borrado se acepta, eliminándose sólo una tupla.

- *Operación*.

  Borrar la tupla EMPLEADO cuyo Dni = '999887777'.
  *Resultado.* Este borrado no se acepta porque existen tuplas en TRABAJA_EN que hacen referencia a ella. Por tanto, si se elimina la tupla en EMPLEADO, se producirán violaciones de la integridad referencial.

- *Operación*.

  Borrar la tupla EMPLEADO cuyo Dni = '333445555'.
  *Resultado.* Este borrado provocará incluso más violaciones de integridad referencial, ya que la tupla implicada está referenciada desde las relaciones EMPLEADO, DEPARTAMENTO, TRABAJA_EN y SUBORDINADO.

Son varios los caminos que pueden tomarse si un borrado provoca una violación. El primero consiste en *rechazar el borrado.* El segundo pasa por *intentar propagar el borrado* eliminando las tuplas que hacen referencia a la que estamos intentado borrar. Por ejemplo, en la segunda operación, el DBMS podría borrar automáticamente las tuplas de TRABAJA_EN cuyo DniEmpleado = '999887777'. Una tercera posibilidad es *modificar los valores del atributo referenciado* que provocan la violación; a cada uno de ellos podría asignársele NULL, o modificarlo de forma que haga referencia a otra tupla válida. Tenga en cuenta que si el atributo referenciado que provoca la violación es *parte de la clave principal, no puede* ser establecido a NULL; de lo contrario, podría violar la integridad de entidad.

Son posibles combinaciones de estas tres posibilidades. Por ejemplo, para evitar que la tercera operación falle, la base de datos podría borrar automáticamente todas las tuplas de TRABAJA_EN y SUBORDINADO con un DniEmpleado = '333445555'. Las tuplas de EMPLEADO cuyo SuperDni = '333445555' y la de DEPARTAMENTO con un DniDirector = '333445555' podrían cambiar sus valores por otros válidos o por NULL. Aunque podría parecer lógico borrar automáticamente las tuplas de TRABAJA_EN y SUBORDINADO que hacen referencia a una tupla EMPLEADO, no lo tendría en el caso de estar hablando de EMPLEADO o DEPARTAMENTO.

En general, cuando se especifica una restricción de integridad referencial en el DDL, el DBMS permitirá al usuario *especificar las opciones* que se aplicarán en el caso de producirse una violación de dichas restricciones. En el Capítulo 8 veremos el modo de hacer esto en el DDL SQL-99.

### 5.3.3 La operación Update

**Update** (o **Modify**) se emplea para cambiar los valores de uno o más atributos de una tupla (o tuplas) de una relación *R*. Para seleccionar la información a modificar es necesario indicar una condición en los atributos de la relación. He aquí algunos ejemplos:

- *Operación*.

  Actualizar el salario de la tupla EMPLEADO cuyo Dni = '999887777' a 28000.
  *Resultado*. Aceptable.

- *Operación*.

  Actualizar el Dno de la tupla EMPLEADO con Dni = '999887777' a 1.
  *Resultado.* Aceptable.

- *Operación.*

  Actualizar el Dno de la tupla EMPLEADO con Dni = '999887777' a 7.
  *Resultado.* Inaceptable porque viola la integridad referencial.

- *Operación*.

  Actualizar el Dni de la tupla EMPLEADO cuyo Dni = '999887777' a '987654321'.
  *Resultado.* Inaceptable porque viola la restricción de clave principal repitiendo un valor que ya existe en otra tupla; viola las restricciones de integridad referencial ya que existen otras relaciones que hacen referencia a un Dni que ya existe.

La actualización de un atributo que no forma parte ni de una clave principal ni de una *foreign key* no suele plantear problemas; el DBMS sólo tiene que verificar que el nuevo valor tiene el tipo de dato y dominio correctos. La modificación de una clave principal es una operación similar al borrado de una tupla y la inserción de otra en su lugar, ya que usamos esta clave principal para identificar dichas tuplas. Por tanto, los problemas mostrados en las Secciones 5.3.1 (Insert) y 5.3.2 (Delete) pueden aparecer. Si una *foreign key* se modifica, la base de datos debe asegurarse de que el nuevo valor hace referencia a una tupla existente en la relación referenciada (o es NULL). Existen opciones similares para tratar con las violaciones de integridad referencial provocadas por Update a las comentadas para la operación Delete. De hecho, cuando se especifi-

ca una restricción de este tipo en el DDL, el DBMS permitirá que el usuario elija de forma separada las acciones a tomar en cada caso (consulte la Sección 8.2).

### 5.3.4 El concepto de transacción

Una aplicación de bases de datos ejecutándose contra una base de datos referencial suele ejecutar una serie de transacciones. Este proceso implica tanto la lectura desde una base de datos como efectuar inserciones, borrados y actualizaciones en los valores de la misma. Es necesario dejar la base de datos en un estado coherente; este estado debe cumplir todas las restricciones comentadas en la Sección 5.2. Una transacción simple puede implicar cualquier número de operaciones de recuperación (se comentará como parte del álgebra relacional y los cálculos en el Capítulo 6, y del lenguaje SQL en los Capítulos 8 y 9) que lean información desde la base de datos y otras de actualización. Existen un gran número de aplicaciones comerciales funcionando contra bases de datos relacionales en los sistemas **OLTP (Procesamiento de transacciones en línea, *Online Transaction Processing*)** que realizan transacciones a velocidades cercanas a varios cientos de ellas por segundo. Los conceptos de procesamiento de una transacción, ejecución concurrente de transacciones y recuperación ante fallos serán tratados en los Capítulos del 17 al 19.

## 5.4 Resumen

En este capítulo se han tratado los conceptos de modelado, estructuras de datos y restricciones ofrecidas por el modelo relacional de datos. Empezamos presentando la definición de dominios, atributos y tuplas. Después definimos un esquema de relación como una lista de atributos que describen la estructura de la misma. Una relación, o un estado de relación, es un conjunto de tuplas que se adapta al esquema.

Son varias las características que diferencian las relaciones de las tablas corrientes o los ficheros. La primera es que una relación no es sensible al orden de las tuplas. La segunda compete a la ordenación de los atributos en un esquema de relación y la ordenación de valores correspondiente dentro de una tupla. Ofrecimos una definición alternativa de relación que no requiere estas dos ordenaciones aunque, por conveniencia, seguimos usando la primera de ellas, que precisa que los atributos y los valores de las tuplas estén ordenados. A continuación, comentamos los valores en la tuplas y presentamos los valores NULL para representar información desaparecida o desconocida, aunque acentuamos también el hecho de que hay que evitar en lo posible el uso de NULL.

Clasificamos las restricciones de la base de datos en inherentes basadas en el modelo, explícitas basadas en el esquema y basadas en aplicación, conocidas también estas últimas como restricciones semánticas o reglas de negocio. Seguidamente, explicamos el esquema de restricciones perteneciente al modelo relacional, empezando con las de dominio, siguiendo con las de clave, incluyendo los conceptos de superclave, clave candidata y clave principal, y las restricciones NOT NULL en los atributos. Definimos bases de datos relacionales y esquemas para ellas. Las restricciones de integridad prohíben que las claves primarias sean de tipo NULL. Describimos la restricción de integridad referencial, que se usa para mantener la coherencia en las referencias entre tuplas de distintas relaciones.

Las operaciones de modificación en el modelo relacional son Insert, Delete y Update. Cada una de ellas puede violar ciertos tipos de restricción (consulte la Sección 5.3). Siempre que se aplica una operación, es necesario comprobar el estado de la base de datos para garantizar que no se ha violado ninguna restricción. Por último, presentamos el concepto de transacción, la cual es importante en los DBMSs relacionales.

### Preguntas de repaso

**5.1.** Defina los siguientes términos: dominio, atributo, *n*-tupla, esquema de relación, estado de relación, grado de una relación, esquema de base de datos relacional y estado de una base de datos relacional.

**5.2.** ¿Por qué hay tuplas sin ordenar en una relación?

**5.3.** ¿Por qué no están permitidas las tuplas duplicadas en una relación?

**5.4.** ¿Cuál es la diferencia entre una clave y una superclave?

**5.5.** ¿Por qué designamos a una de las claves candidatas de una relación como clave principal?

**5.6.** Comente las características que hacen diferentes a las relaciones de las tablas corrientes y los ficheros.

**5.7.** Comente las distintas razones que llevan a la aparición de valores NULL en las relaciones.

**5.8.** Comente las restricciones de integridad de entidad y referencial. ¿Por qué es importante cada una de ellas?

**5.9.** Defina *foreign key*. ¿Para qué se usa este concepto?

**5.10.** ¿Qué es una transacción? ¿En qué se diferencia de una actualización?

## Ejercicios

**5.11.** Supongamos que las siguientes actualizaciones se aplican directamente a la base de datos mostrada en la Figura 5.6. Comente *todas* las restricciones de integridad que se violan en cada una de ellas, en caso de que existan, y las distintas formas de hacer que se cumplan.

- **a.** Insert <'Roberto', 'Flandes', 'Martín', '943775543', '21-06-1952', 'Campanillas, 189',H, 58000, '888665555', 1> into EMPLEADO.
- **b.** Insert <'ProductoA', 4, 'Buenos Aires', 2> into PROYECTO.
- **c.** Insert <'Producción', 4, '943775543', '01-10-1998'> into DEPARTAMENTO.
- **d.** Insert <'677678989', NULL, '40.0'> into TRABAJA_EN.
- **e.** Insert <'453453453', 'Juan', 'Martín', '12-12-1970', 'Cónyuge'> into SUBORDINADO.
- **f.** Borrar las tuplas TRABAJA_EN cuyo DniEmpleado = '333445555'.
- **g.** Borrar la tupla EMPLEADO cuyo Dni = '987654321'.
- **h.** Borrar la tupla PROYECTO cuyo NombreProyecto = 'ProductoX'.
- **i.** Modificar DniDirector y FechaIngresoDirector de la tupla DEPARTAMENTO cuyo NumeroDpto = 5 por '123456789' y '01-10-1999', respectivamente.
- **j.** Modificar el atributo SuperDni de la tupla EMPLEADO con Dni = '999887777' a '943775543'.
- **k.** Modificar el atributo Horas de la tupla TRABAJA_EN con DniEmpleado = '999887777' y NumProy = 10 a '5.0'.

**5.12.** Considere el esquema de base de datos relacional LINEA_AEREA de la Figura 5.8. Cada VUELO está identificado por un NumVuelo compuesto por uno o más PLAN_VUELO con los NumPlan 1, 2, 3, etc. Cada PLAN_VUELO tiene programadas una hora de llegada y de partida, los aeropuertos y una o más INSTANCIA_PLAN (una por cada Fecha en la que viaja el vuelo). Cada VUELO mantiene un PRECIO_BILLETE. Para cada instancia de PLAN_VUELO, se mantiene una RESERVA_ASIENTO, así como el AVION usado y las fechas y aeropuertos actuales de origen y destino. Un AVION está identificado por un IdAvion y si es de un TIPO_AVION. PUEDE_ATERRIZAR relaciona cada TIPO_AVION con el AEROPUERTO en el que puede aterrizar. Un AEROPUERTO está identificado por un CodAeropuerto. Considere una actualización de la base de datos LINEA_AEREA para introducir una reserva para un vuelo, o plan de vuelo, concretos en una fecha dada.

- **a.** Realice las operaciones para llevar a cabo esta actualización.
- **b.** ¿Qué tipo de restricciones cree que deberá comprobar?

**Figura 5.8.** El esquema de base de datos relacional LINEA_AEREA.

**AEROPUERTO**

| CodAeropuerto | Nombre | Ciudad | Provincia |
|---|---|---|---|

**VUELO**

| NumVuelo | Aerolínea | DiasSemana |
|---|---|---|

**PLAN_VUELO**

| NumVuelo | NumPlan | CodAeropuertoSalida | HoraSalidaProgramada | CodAeropuertoLlegada | HoraLlegadaProgramada |
|---|---|---|---|---|---|

**INSTANCIA_PLAN**

| NumVuelo | NumPlan | Fecha | NumAsientosLibres | IdAvion | CodAeropuertoSalida | HoraSalida | CodAeropuertoLlegada | HoraLlegada |
|---|---|---|---|---|---|---|---|---|

**PRECIO_BILLETE**

| NumVuelo | CodTarifa | Cantidad | Restricciones |
|---|---|---|---|

**TIPO_AVIÓN**

| NombreTipoAvion | MaxAsientos | Compañía |
|---|---|---|

**PUEDE_ATERRIZAR**

| NombreTipoAvion | CodAeropuerto |
|---|---|

**AVIÓN**

| IdAvion | NumTotalAsientos | TipoAvion |
|---|---|---|

**RESERVA_ASIENTO**

| NumVuelo | NumPlan | Fecha | NumAsiento | NombreCliente | TlfCliente |
|---|---|---|---|---|---|

**c.** ¿Cuáles de estas restricciones son de clave, de integridad de entidad y de integridad referencial, y cuáles no?

**d.** Especifique todas las restricciones de integridad referencial que existen en el esquema de la Figura 5.8.

**5.13.** Considere la relación CURSO (NumeroCurso, NumeroAreaUniversidad, NombreInstructor, Semestre, CodigoEdificio, NumeroSala, PeriodoTiempo, DiasVacaciones, HorasCredito). Esta relación representa los cursos impartidos en una universidad, con un NumeroAreaUniversidad único. Indique sus impresiones acerca de las posibles claves candidatas, y escriba con sus propias palabras las restricciones bajo las que cada una de las claves candidatas sería válida.

**5.14.** Considere las siguientes seis relaciones de una aplicación de base de datos para el procesamiento de los pedidos de una empresa:

CLIENTE(NumeroCliente, NombreCliente, Ciudad)

PEDIDO(NumeroPedido, FechaPedido, NumeroCliente, TotalBruto)

LINEA_PEDIDO(NumeroPedido, NumeroProducto, Cantidad)

PRODUCTO(NumeroProducto, PrecioUnitario)

ENTREGA(NumeroEntrega, NumeroAlmacen, FechaSalida)

ALMACEN(NumeroAlmacen, Ciudad)

En este ejemplo, TotalBruto hace referencia a la cantidad total de dólares de un pedido; FechaPedido es la fecha en la que fue realizado y FechaSalida es la fecha de salida del pedido del almacén. Tenga en cuenta que un pedido puede ser emitido desde varios almacenes. Especifique las *foreign keys* de este esquema, argumentando todas sus decisiones. ¿Qué otras restricciones piensa que son necesarias para esta base de datos?

**5.15.** Considere las siguientes relaciones para una base de datos que mantiene los viajes de negocios de los comerciales de una empresa:

COMERCIAL(Dni, Nombre, FechaIngreso, NumeroDpto)

VIAJE (Dni, CiudadOrigen, CiudadDestino, FechaSalida, FechaRegreso, IdViaje)

GASTO(IdViaje, NumeroCuenta, Cantidad)

Especifique las *foreign keys* necesarias para este esquema, argumentando todas sus decisiones.

**5.16.** Considere las siguientes relaciones para una base de datos que controla las matriculaciones de los estudiantes en los cursos y los libros utilizados en los mismos:

ESTUDIANTE(Dni, Nombre, Asignatura, FechaNac)

CURSO(NumeroCurso, NombreCurso, Departamento)

MATRICULACION(Dni, NumeroCurso, Trimestre, Nota)

LIBRO_USADO(NumeroCurso, Trimestre, ISBNLibro)

TEXTO(ISBNLibro, TituloLibro, Editorial, Autor)

Especifique las *foreign keys* necesarias para este esquema, argumentando todas sus decisiones.

**5.17.** Considere las siguientes relaciones para una base de datos que controla las ventas de coches de un concesionario (EXTRAS hace referencia al equipamiento que no es de serie en el vehículo):

COCHE(NumeroBastidor, Modelo, Fabricante, PrecioCoche)

EXTRAS(NumeroBastidor, NombreExtra, PrecioExtra)

VENTA(IdVendedor, NumeroBastidor, Fecha, PrecioVenta)

VENDEDOR (IdVendedor, Nombre, Telefono)

En primer lugar, especifique las *foreign keys* necesarias para este esquema, argumentando todas sus decisiones. A continuación, rellene las relaciones con varias tuplas de ejemplo e indique algún ejemplo de una inserción en VENTA y en VENDEDOR que *viole* las restricciones de integridad referencial y otra que no lo haga.

**5.18.** El diseño de una base de datos suele implicar a veces la toma de decisiones acerca del almacenamiento de los atributos. Por ejemplo, el Número de la Seguridad Social en Estados Unidos puede almacenarse como un único atributo o dividirse en tres (uno por cada grupo de números en los que está dividido: XXX-XX-XXXX). Sin embargo, este dato suele representarse como un solo atribu-

to. La decisión que tome está basada en la forma en que se utilizará la base de datos. Este ejercicio pretende que argumente las situaciones concretas en las que la división del Número de la Seguridad Social puede resultar útil.

**5.19.** Considere una relación ESTUDIANTE en una base de datos UNIVERSIDAD con los siguientes atributos (Nombre, Dni, TlfParticular, Dirección, TlfTrabajo, Edad, Mnc). Tenga en cuenta que el TlfTrabajo puede ser de una provincia diferente al TlfParticular. A continuación se muestra una posible tupla de la relación:

| Nombre | Dni | TlfParticular | Direccion | TlfTrabajo | Edad | Mnc |
|---|---|---|---|---|---|---|
| Pedro Blas Fiz | 12-453-678 | 935551234 | Martinicos, 34 | 965554321 | 19 | 3,75 |

**a.** Identifique la información crítica perdida de los atributos TlfParticular y TelefonoMovil del ejemplo anterior. (*Advertencia*: Hasta hace poco, los números de teléfono en España identificaban la provincia con un número de prefijo que era necesario marcar para contactar con los usuarios de dicha provincia.)

**b.** ¿Almacenaría esta información adicional en los atributos TlfParticular y TlfTrabajo o añadiría otros nuevos al esquema ESTUDIANTE?

**c.** Considere el atributo Nombre. ¿Cuáles son las ventajas de dividir este valor en tres atributos (nombre, apellido1, apellido2)?

**d.** ¿Cuál sería la línea maestra que recomendaría para decidir cuándo almacenar la información en un atributo único y cuándo dividirlo en varios?

**e.** Suponga que el estudiante puede tener entre ninguno y cinco teléfonos. Sugiera dos tipos de diseños diferentes que permitan este tipo de información.

## Bibliografía seleccionada

El modelo relacional fue presentado por Codd (1970) en un artículo. Codd también introdujo el álgebra relacional y trazó los fundamentos teóricos del modelo relacional en una serie de artículos (Codd 1971, 1972, 1972a, 1974); más tarde obtuvo el premio Turing, la máxima distinción de la ACM, por este trabajo. En un escrito posterior, Codd (1979) propuso la extensión del modelo relacional para incorporar más metadatos y semánticas sobre las relaciones; también planteó una lógica *three-valued* para tratar la incertidumbre en las relaciones e incorporó los NULL en el álgebra relacional. El modelo resultante es conocido como RM/T. Childs (1968) había usado anteriormente la teoría de conjuntos para modelar bases de datos. Posteriormente, Codd (1990) publicó un libro en el que examinaba alrededor de 300 características del modelo de datos relacional y los sistemas de bases de datos. Date (2001) mostró una retrospectiva y un análisis del modelo de datos relacional.

Desde el trabajo pionero de Codd, han sido muchos los trabajos que se han realizado sobre diversos aspectos del modelo relacional. Todd (1976) describe un DBMS experimental llamado PRTV que implementa directamente las operaciones del álgebra relacional. Schmidt y Swenson (1975) muestra una semántica adicional en el modelo relacional clasificando diferentes tipos de relaciones. El modelo Entidad-Relación de Chen (1976), que ya se comentó en el Capítulo 3, es un intento de comunicar las semánticas del mundo real de una base de datos a un nivel conceptual. Wiederhold y Elmasri (1979) presentan varios tipos de conexiones entre relaciones para mejorar sus restricciones.

Las extensiones del modelo relacional se tratan en el Capítulo 24. Los Capítulos del 6 al 11, 15, 16, 17, 22, 23 y 25 contienen notas bibliográficas adicionales acerca de otros aspectos del modelo relacional y sus lenguajes, sistemas, extensiones y teoría. Maier (1983) y Atzeni y otros (1993) ofrecen un tratamiento teórico extenso del modelo de datos relacional.

CAPÍTULO 6

# El álgebra relacional y los cálculos relacionales

En este capítulo vamos a tratar los dos lenguajes formales del modelo relacional: el álgebra relacional y los cálculos relacionales. Como ya comentamos en el Capítulo 2, un modelo de datos debe incluir un conjunto de operaciones para manipular la base de datos junto con los conceptos necesarios para la definición de su estructura y restricciones. El conjunto de operaciones básicas del modelo relacional es el **álgebra relacional**, el cual permite al usuario especificar las peticiones fundamentales de recuperación. El resultado de una recuperación es una nueva relación, la cual puede estar constituida por una o más relaciones. Por consiguiente, las operaciones de álgebra producen nuevas relaciones que pueden ser manipuladas más adelante usando operaciones del mismo álgebra. Una secuencia de operaciones de álgebra relacional conforma una expresión de álgebra relacional, cuyo resultado será también una nueva relación que representa el resultado de una consulta a la base de datos (o una petición de recuperación).

El álgebra relacional es muy importante por varias razones. La primera, porque proporciona un fundamento formal para las operaciones del modelo relacional. La segunda razón, y quizá la más importante, es que se utiliza como base para la implementación y optimización de consultas en los RDBMS (Sistemas de administración de bases de datos relacionales, *Relational DataBase Management Systems*), tal y como se comentará en la Parte 4. Tercera, porque algunos de sus conceptos se han incorporado al lenguaje estándar de consultas SQL para los RDBMS.

Aunque ninguno de los RDBMS comerciales actuales proporcionan una interfaz para las consultas de álgebra relacional, las funciones y operaciones centrales de cualquier sistema relacional están basadas en estas operaciones, que explicamos con detalle en las siguientes secciones.

Mientras que el álgebra define un conjunto de operaciones del modelo relacional, los **cálculos relacionales** ofrecen una notación declarativa de alto nivel para especificar las consultas relacionales. Una expresión de cálculo relacional crea una nueva relación, la cual está especificada en términos de variables que engloban filas de las relaciones almacenadas en la base de datos (en cálculos de tupla) o columnas de las relaciones almacenadas (para los cálculos de dominio). En una expresión de cálculo, *no existe un orden de operaciones* para recuperar los resultados de la consulta: la expresión sólo especifica la información que el resultado debería contener. Ésta es la diferencia principal entre el álgebra relacional y los cálculos relacionales. Éste último es importante porque tiene una base firme en la lógica matemática y porque el SQL (Lenguaje de consulta estándar, *Standard Query Language*) para los RDBMS tiene alguno de sus fundamentos en los cálculos de tupla relacional.[1]

[1] SQL está basado en los cálculos relacionales de tupla, aunque también incorpora algunas de las operaciones del álgebra relacional y sus extensiones, como veremos en los Capítulos 8 y 9.

El álgebra relacional tiende a ser considerado como una parte integral del modelo de datos relacional. Sus operaciones pueden dividirse en dos grupos. Uno de ellos incluye el conjunto de operaciones de la teoría matemática de conjuntos, los cuales son aplicables porque cada relación está definida de modo que sea un conjunto de tuplas en el modelo relacional formal. Estas operaciones incluyen UNIÓN (UNION), INTERSECCIÓN (*INTERSECTION*), DIFERENCIA DE CONJUNTOS (*SET DIFFERENCE*) y PRODUCTO CARTESIANO (*CARTESIAN PRODUCT*). El otro grupo está constituido por las operaciones desarrolladas específicamente para las bases de datos relacionales, como la SELECCIÓN (SELECT), la PROYECCIÓN (*PROJECT*), la CONCATENACIÓN o COMBINACIÓN (JOIN) y otras. La Sección 6.1 empieza tratando las operaciones SELECCIÓN y PROYECCIÓN porque son **operaciones unarias** que operan en relaciones individuales. La Sección 6.2 se encarga del conjunto de operaciones, mientras que la 6.3 se centra en la CONCATENACIÓN y otras **operaciones binarias** complejas que operan sobre dos tablas. En los ejemplos utilizaremos la base de datos EMPRESA de la Figura 5.6.

Algunas de las peticiones de base de datos más comunes no pueden llevarse a cabo con las operaciones del álgebra relacional originales, por lo que se desarrollaron otras nuevas para lograrlo. Entre estas operaciones se incluyen las **funciones agregadas**, que son operaciones que pueden *resumir* datos a partir de tablas, así como operaciones CONCATENACIÓN y UNIÓN adicionales. Estas operaciones fueron añadidas al álgebra relacional original debido a su importancia para muchas aplicaciones de bases de datos, y se describen en la Sección 6.4. En la Sección 6.5 ofreceremos ejemplos de consultas que usan operaciones relacionales, y algunas de ellas se utilizan posteriormente en otros capítulos para ilustrar varios lenguajes.

En las Secciones 6.6 y 6.7 se describe el otro tipo de lenguaje formal para las bases de datos relacionales, los **cálculos relacionales**. Existen dos variantes del mismo. El cálculo relacional de tupla se explica en la Sección 6.6, mientras que el de dominio se detalla en la Sección 6.7. Algunas de las construcciones SQL tratadas en el Capítulo 8 están basadas en la primera de estas variantes. El cálculo relacional es un lenguaje formal basado en una rama de la lógica matemática llamada cálculos de predicado[2]. En los cálculos relacionales de tupla, las variables alcanzan a las tuplas, mientras que en los de dominio "atacan" a los valores de los atributos. En el Apéndice D podrá encontrar una descripción del QBE (Consulta mediante ejemplo, *Query-By-Example*), un lenguaje relacional gráfico y agradable para el usuario basado en los cálculos relacionales de dominio. La Sección 6.8 resume todo este capítulo.

Los lectores interesados en una introducción menos detallada de los lenguajes relacionales formales pueden saltarse las Secciones 6.4, 6.6 y 6.7.

# 6.1 Operaciones relacionales unarias: SELECCIÓN (*SELECT*) y PROYECCIÓN (*PROJECT*)

## 6.1.1 La operación SELECCIÓN

SELECCIÓN se emplea para seleccionar un *subconjunto* de las tuplas de una relación que satisfacen una **condición de selección**. Se puede considerar esta operación como un *filtro* que mantiene sólo las tuplas que satisfacen una determinada condición. SELECCIÓN puede visualizarse también como una *partición horizontal* de la relación en dos conjuntos de tuplas: las que satisfacen la condición son seleccionadas y las que no, descartadas. Por ejemplo, para seleccionar las tuplas de EMPLEADO cuyo departamento sea 4, o cuyo salario sea mayor de 30.000 euros, podemos especificar individualmente cada una de estas condiciones con una operación SELECCIÓN como ésta:

$$\sigma_{Dno=4}(\text{EMPLEADO})$$

$$\sigma_{Sueldo>30000}(\text{EMPLEADO})$$

[2] En este capítulo se asume que no tiene ningún conocimiento sobre los cálculos de predicado de primer orden (los que tratan con variables y valores cuantificados).

En general, SELECCIÓN está designada como:

$$\sigma_{\text{<condición de selección>}}(R)$$

donde el símbolo σ (sigma) se utiliza para especificar el operador de SELECCIÓN, mientras que la condición de selección es una expresión lógica (o booleana) especificada sobre los atributos de la relación *R*. Observe que *R* es, generalmente, una *expresión de álgebra relacional* cuyo resultado es una relación: la más sencilla de estas expresiones es sólo el nombre de una relación de base de datos. El resultado de SELECCIÓN tiene los *mismos atributos* que *R*.

La expresión lógica especificada en <condición de selección> está constituida por un número de **cláusulas** de la forma:

<nombre de atributo> <operador de comparación> <valor constante>,

o bien:

<nombre de atributo> <operador de comparación> <nombre de atributo>

donde <nombre de atributo> es el nombre de un atributo de *R*, <operador de comparación> suele ser uno de los operadores {=, <, ≤, >, ≥, ≠ } y <valor constante> es un valor del dominio del atributo. Las cláusulas pueden estar conectadas arbitrariamente por operadores lógicos *and*, *or* y *not* para formar una condición de selección general. Por ejemplo, para seleccionar las tuplas de todos los empleados que trabajan en el departamento 4 y ganan sobre 25.000 euros al año, o los que trabajan en el 5 y ganan alrededor de 30.000, podemos especificar la siguiente operación de SELECCIÓN:

$$\sigma_{(\text{Dno=4 } \mathbf{AND} \text{ Sueldo>25000}) \mathbf{OR} (\text{Dno=5 } \mathbf{AND} \text{ Sueldo>30000})}(\text{EMPLEADO})$$

El resultado aparece en la Figura 6.1(a).

**Figura 6.1.** Resultado de las operaciones SELECCIÓN y PROYECCIÓN. (a) s (Dno=4 AND Sueldo>25000) OR (Dno=5 AND Sueldo > 30000) (EMPLEADO). (b) pApellido1, Nombre, Sueldo(EMPLEADO). (c) pSexo, Sueldo(EMPLEADO).

**(a)**

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|
| Alberto | Campos | Sastre | 333445555 | 08-12-1955 | Avda. Ríos, 9 | H | 40000 | 888665555 | 5 |
| Juana | Sainz | Oreja | 987654321 | 20-06-1941 | Cerquillas, 67 | M | 43000 | 888665555 | 4 |
| Fernando | Ojeda | Ordóñez | 666884444 | 15-09-1962 | Portillo, s/n | H | 38000 | 333445555 | 5 |

**(b)**

| Apellido1 | Nombre | Sueldo |
|---|---|---|
| Pérez | José | 30000 |
| Campos | Alberto | 40000 |
| Jiménez | Alicia | 25000 |
| Sainz | Juana | 43000 |
| Ojeda | Fernando | 38000 |
| Oliva | Aurora | 25000 |
| Pajares | Luis | 25000 |
| Ochoa | Eduardo | 55000 |

**(c)**

| Sexo | Sueldo |
|---|---|
| H | 30000 |
| H | 40000 |
| M | 25000 |
| M | 43000 |
| H | 38000 |
| H | 25000 |
| H | 55000 |

Observe que los operadores de comparación del conjunto {=, <, ≤, >, ≥, ≠} se aplican a los atributos cuyos dominios son *valores ordenados*, como los de tipo numérico o de fecha. Los de cadenas de caracteres también se consideran del mismo tipo debido a la secuencia en los códigos. Si el dominio de un atributo es un conjunto de *valores desordenados*, sólo pueden usarse los operadores {=, ≠}. Color = {'rojo', 'azul', 'verde', 'blanco', 'amarillo', . . .} es un ejemplo de dominio desordenado. Algunos de ellos permiten operadores de comparación adicionales; por ejemplo, un dominio de cadenas de caracteres permite el operador SUBCADENA_DE.

En general, el resultado de una operación SELECCIÓN puede determinarse como sigue. La <condición de selección> se aplica independientemente a cada tupla *t* de *R*. Esto se realiza sustituyendo cada ocurrencia de un atributo $A_i$ en la condición de selección por su valor en la tupla $t[A_i]$. Si la condición se evalúa como VERDADERO (*TRUE*), la tupla *t* se **selecciona**.

Todas las tuplas seleccionadas aparecen en el resultado de SELECCIÓN. Las condiciones lógicas AND, OR y NOT tienen la siguiente interpretación:

- (condición1 **AND** condición2) es VERDADERO si las dos condiciones (condición1 y condición2) lo son; en cualquier otro caso, es FALSO (FALSE).
- (condición1 **OR** condición2) es VERDADERO si (condición1) o (condición2) o ambas son verdaderas; en cualquier otro caso, es FALSO.
- (**NOT** condición) es VERDADERO si condición es FALSO; en cualquier otro caso es FALSO.

SELECCIÓN es **unaria**, es decir, se aplica a una sola relación. Además, esta operación se aplica a *cada tupla individualmente*; por consiguiente, las condiciones de selección no pueden implicar a más de una tupla. El **grado** de la relación resultante de una operación SELECCIÓN (su número de atributos) es el mismo que el de *R*. El número de tuplas en la relación resultante es siempre *menor que o igual que* el número de tuplas en *R*. Esto es, $\sigma_c(R)| \leq |R|$ para cualquier condición *C*. La fracción de tuplas seleccionadas por una condición de relación está referida como la **selectividad** de la condición.

Observe que la operación SELECCIÓN es **conmutativa**, es decir:

$$\sigma_{<\text{condición1}>}(\sigma_{<\text{condición2}>}(R)) = \sigma_{<\text{condición2}>}(\sigma_{<\text{condición1}>}(R))$$

Por tanto, puede aplicarse una secuencia de SELECCIONES en cualquier orden. Además, siempre podemos combinar una **cascada** de operaciones SELECCIÓN en una sola a través de un (AND):

$$\sigma_{<\text{condición1}>}(\sigma_{<\text{condición2}>}(\ldots(\sigma_{<\text{condición}>}(R))\ldots)) = \sigma_{<\text{condición1}>\ \textbf{AND}\ <\text{condición2}>\ \textbf{AND}\ \ldots\ \textbf{AND}\ <\text{condición}>}(R)$$

## 6.1.2 La operación PROYECCIÓN

Si pensamos en una relación como en una tabla, la operación SELECCIÓN elige algunas de las *filas* de la tabla a la vez que descarta otras. Por otro lado, PROYECCIÓN selecciona ciertas *columnas* de la tabla y descarta otras. Si sólo estamos interesados en algunos atributos de una relación, usamos la operación PROYECCIÓN para *planear* la relación sólo sobre esos atributos. Por consiguiente, el resultado de esta operación puede visualizarse como una *partición vertical* de la relación en otras dos: una contiene las columnas (atributos) necesarias y otra las descartadas. Por ejemplo, para listar el nombre, el primer apellido y el sueldo de cada empleado, podemos usar PROYECCIÓN de la siguiente forma:

$$\pi_{\text{Apellido1, Nombre, Sueldo}}(\text{EMPLEADO})$$

La relación resultante se muestra en la Figura 6.1(b). La forma general de la operación PROYECCIÓN es:

$$\pi_{<\text{lista de atributos}>}(R)$$

donde $\pi$ (pi) es el símbolo usado para representar la operación PROYECCIÓN, mientras que <lista de atributos> contiene la lista de campos de la relación *R* que queremos. De nuevo, observe que *R* es, en general, una *expresión de álgebra relacional* cuyo resultado es una relación, cuyo caso más simple es obtener sólo el nombre de una relación de base de datos. El resultado de la operación PROYECCIÓN sólo tiene los atributos especificados en <lista de atributos> *en el mismo orden a como aparecen en la lista*. Por tanto, su **grado** es igual al número de atributos contenidos en <lista de atributos>.

Si la lista de atributos sólo incluye atributos no clave de *R*, es posible que se dupliquen tuplas. La operación PROYECCIÓN *elimina cualquier tupla duplicada*, por lo que el resultado de la misma es un conjunto de tuplas y, por consiguiente, una relación válida. Esto se conoce como **eliminación de duplicados.** Por ejemplo, considere la siguiente operación PROYECCIÓN:

$$\pi_{\text{Sexo, Sueldo}}(\text{EMPLEADO})$$

El resultado aparece en la Figura 6.1(c). Observe que la tupla <'M', 25000> sólo aparece una vez en dicha figura, aun cuando su combinación de valores aparezca dos veces en EMPLEADO. La eliminación de duplicados lleva implícito un proceso de ordenación para detectar esos registros repetidos y, por tanto, añadir más capacidad de procesamiento. Si no se llevara a cabo este proceso, el resultado sería un **multiconjunto** o **bolsa** de tuplas en lugar de un conjunto. Esto no estaba permitido en el modelo relacional formal, aunque sí lo estaba en la práctica. En el Capítulo 8 mostraremos que el usuario puede optar por eliminar o no los registros duplicados.

El número de tuplas de una relación resultante de una operación PROYECCIÓN es siempre menor o igual que el de las contenidas en *R*. Si la lista de proyección es una superclave de *R* (esto es, incluye alguna clave de *R)* la relación resultante tiene el mismo número de tuplas que *R*. Además,

$$\pi_{<\text{lista1}>}(\pi_{<\text{lista2}>}(R)) = \pi_{<\text{lista1}>}(R)$$

con tal de que <lista2> contenga los atributos de <lista1>; de otro modo, la parte de la izquierda es una expresión incorrecta. También resulta digno de mención el hecho de que la conmutatividad *no* se almacena en una PROYECCIÓN.

### 6.1.3 Secuencias de operaciones y la operación RENOMBRAR (*RENAME*)

Las relaciones mostradas en la Figura 6.1 no tienen ningún nombre. En general, podemos querer aplicar varias operaciones de álgebra relacional una tras otra. De cualquier forma, podemos escribir las operaciones como una única **expresión de álgebra relacional** anidando dichas operaciones, o aplicar una sola expresión una única vez y crear relaciones intermedias. En el último caso, puede que queramos asignar nombres a dichas relaciones intermedias. Por ejemplo, para recuperar el nombre, el primer apellido y el sueldo de todos los empleados que trabajan en el departamento 5, debemos aplicar una SELECCIÓN y una PROYECCIÓN. Podemos escribir una expresión de álgebra relacional sencilla de la siguiente forma:

$$\pi_{\text{Nombre, Apellido1, Sueldo}}(\sigma_{\text{Dno=5}}(\text{EMPLEADO}))$$

La Figura 6.2(a) muestra el resultado de esta operación. Alternativamente, podemos mostrar la secuencia de operaciones, dando un nombre a cada relación intermedia:

$$\text{DEP5_EMPS} \leftarrow \sigma_{\text{Dno=5}}(\text{EMPLEADO})$$
$$\text{RESULTADO} \leftarrow \pi_{\text{Nombre, Apellido1, Sueldo}}(\text{DEP5_EMPS})$$

Con frecuencia, resulta más simple partir una secuencia compleja de operaciones en relaciones intermedias que escribir una única expresión de álgebra relacional. Podemos usar también esta técnica para **renombrar**

**Figura 6.2.** Resultado de una secuencia de operaciones. (a) $\pi_{\text{Nombre, Apellido1, Sueldo}}(\sigma_{\text{Dno=5}}(\text{EMPLEADO}))$. (b) Usando relaciones intermedias y renombrando los atributos.

**(a)**

| Nombre | Apellido1 | Sueldo |
|---|---|---|
| José | Pérez | 30000 |
| Alberto | Campos | 40000 |
| Fernando | Ojeda | 38000 |
| Aurora | Oliva | 25000 |

**(b)**

**TEMP**

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|
| José | Pérez | Pérez | 123456789 | 01-09-1965 | Eloy I, 98 | H | 30000 | 333445555 | 5 |
| Alberto | Campos | Sastre | 333445555 | 08-12-1955 | Avda. Ríos, 9 | H | 40000 | 888665555 | 5 |
| Fernando | Ojeda | Ordóñez | 666884444 | 15-09-1962 | Portillo, s/n | H | 38000 | 333445555 | 5 |
| Aurora | Oliva | Avezuela | 453453453 | 31-07-1972 | Antón, 6 | M | 25000 | 333445555 | 5 |

R

| NuevoNombre | NuevoApellido | NuevoSueldo |
|---|---|---|
| José | Pérez | 30000 |
| Alberto | Campos | 40000 |
| Fernando | Ojeda | 38000 |
| Aurora | Oliva | 25000 |

los atributos en las relaciones intermedias y resultantes, lo que puede resultar útil cuando se emplea junto con operaciones más complejas como UNIÓN y CONCATENACIÓN. Para renombrar los atributos de una relación, simplemente enumeramos los nuevos nombres de atributos dentro de los paréntesis, como puede verse en el siguiente ejemplo:

$$\text{TEMP} \leftarrow \sigma_{\text{Dno=5}}(\text{EMPLEADO})$$

$$R(\text{NuevoNombre, NuevosApellido, NuevoSueldo}) \leftarrow \pi_{\text{Nombre, Apellido1, Sueldo}}(\text{TEMP})$$

Ambas operaciones se ilustran en la Figura 6.2(b).

Si no se realiza un renombrado, los nombres de atributo de la relación resultante de una SELECCIÓN son los mismos que los de la relación original y aparecen en el mismo orden. Para el caso de una operación PROYECCIÓN, la relación resultante tiene los mismos nombres de atributo que los indicados en la lista de proyección y están en el mismo orden en que aparecen en dicha lista.

Podemos definir una operación **RENOMBRAR** como un operador unario. Una operación RENOMBRAR aplicada a una relación $R$ de grado $n$ aparece denotada de cualquiera de estas tres formas:

$$\rho_{S(B1, B2, ..., Bn)}(R) \quad \text{o} \quad \rho_S(R) \text{ o } \rho_{(B1, B2, ..., Bn)}(R)$$

donde el símbolo ρ (rho) se utiliza para especificar el operador RENOMBRAR, $S$ es el nombre de la nueva relación y $B_1$, $B_2$, . . ., $B_n$ son los de los nuevos atributos. La primera expresión renombra tanto la relación

como sus atributos, la segunda sólo lo hace con la relación y la tercera sólo con los atributos. Si los atributos de $R$ son $(A_1, A_2, \ldots, A_n)$ por este orden, entonces cada $A_i$ es renombrado como $B_i$.

# 6.2 Operaciones de álgebra relacional de la teoría de conjuntos

## 6.2.1 Las operaciones UNIÓN (*UNION*), INTERSECCIÓN (*INTERSECTION*) y MENOS (*MINUS*)

El siguiente grupo de operaciones de álgebra relacional son las correspondientes a la operativa matemática sobre conjuntos. Por ejemplo, para recuperar los Documentos Nacionales de Identidad de todos los empleados que, o bien trabajan en el departamento 5 o supervisan a éstos, podemos usar la operación UNIÓN del siguiente modo:[3]

DEP5_EMPS ← $\sigma_{Dno=5}$(EMPLEADO)
RESULTADO1←$\pi_{Dni}$(DEP5_EMPS)
RESULTADO2(Dni). ← $\pi_{SuperDni}$(DEP5_EMPS)
RESULTADO ← RESULTADO1 ∪ RESULTADO2

La relación RESULTADO1 tiene el Dni de todos los empleados del departamento 5, mientras que RESULTADO2 contiene el de aquellos empleados que supervisan directamente a los del primer grupo. La UNIÓN produce las tuplas que están en RESULTADO1 o RESULTADO2, o en ambas (consulte la Figura 6.3). De este modo, el Dni '333445555' sólo aparece una vez en el resultado.

Para combinar los elementos de dos conjuntos se utilizan varias operaciones de la teoría de conjuntos, como la **UNIÓN**, la **INTERSECCIÓN** y la **DIFERENCIA DE CONJUNTOS** (llamada también a veces **MENOS**, o ***MINUS***). Todas ellas son operaciones **binarias**, es decir, se aplican a dos conjuntos (de tuplas).

Cuando se refieren a las bases de datos relacionales, las relaciones sobre las que se aplican estas tres operaciones deben tener el mismo **tipo de tuplas**; esta condición recibe el nombre de *compatibilidad de unión*. Dos relaciones $R(A_1, A_2, \ldots, A_n)$ y $S(B_1, B_2, \ldots, B_n)$ se dice que son de **unión compatible** si tienen el mismo grado $n$ y si el $\text{dom}(A_i) = \text{dom}(B_i)$ para $1 \leqslant i \leqslant n$. Esto significa que ambas relaciones tienen el mismo número de atributos y que cada par correspondiente cuenta con el mismo dominio.

**Figura 6.3.** Resultado de la operación de UNIÓN RESULTADO ← RESULTADO1 ∪ RESULTADO2.

**RESULTADO1**

| Dni |
|---|
| 123456789 |
| 333445555 |
| 666884444 |
| 453453453 |

**RESULTADO2**

| Dni |
|---|
| 333445555 |
| 888665555 |

**RESULTADO**

| Dni |
|---|
| 123456789 |
| 333445555 |
| 666884444 |
| 453453453 |
| 888665555 |

[3] Denotado como una expresión de álgebra relacional sencilla, esto se convierte en: Resultado ← $\pi_{Dni}$ ($\sigma_{Dno=5}$ (EMPLEADO) ) ∪. $\pi_{SuperDni}$ ($\sigma_{Dno=5}$ (EMPLEADO)).

Podemos definir las tres operaciones UNIÓN, INTERSECCIÓN y DIFERENCIA DE CONJUNTO en dos relaciones de unión compatible $R$ y $S$ del siguiente modo:

- UNIÓN. El resultado de esta operación, especificada como $R \cup S$, es una relación que incluye todas las tuplas que están tanto en $R$ como en $S$ o en ambas, $R$ y $S$. Las tuplas duplicadas se eliminan.
- INTERSECCIÓN. El resultado de esta operación, especificada como $R \cap S$, es una relación que incluye todas las tuplas que están en $R$ y en $S$.
- DIFERENCIA DE CONJUNTO (o MENOS). El resultado de esta operación, especificada como $R - S$, es una relación que incluye todas las tuplas que están en $R$ pero no en $S$.

Adoptaremos por convenio que los nombres de atributo de la relación resultante son los mismos que los de $R$. Siempre es posible cambiar el nombre de estos valores a través del operador renombrar.

La Figura 6.4 ilustra las tres operaciones. Las relaciones ESTUDIANTE y PROFESOR de la Figura 6.4(a) son una unión compatible y sus tuplas representan los nombres de los estudiantes y los profesores respectivamente. El resultado de la UNIÓN de la Figura 6.4(b) muestra los nombres de todos los estudiantes y profesores. Indicar que las tuplas duplicadas aparecen una sola vez en el resultado. La INTERSECCIÓN de la Figura 6.4(c) incluye sólo aquéllos que son estudiantes y profesores a la vez.

Observe que tanto la UNIÓN como la INTERSECCIÓN son *operaciones conmutativas*, esto es,

$$R \cup S = S \cup R \text{ y } R \cap S = S \cap R$$

La UNIÓN y la INTERSECCIÓN pueden tratarse como operaciones $n$-ary aplicables a cualquier número de relaciones porque son *estructuras asociativas*, es decir,

$$R \cup (S \cup T) = (R \cup S) \cup T \text{ y } (R \cap S) \cap T = R \cap (S \cap T)$$

La operación MENOS *no es conmutativa;* por tanto, en general,

$$R - S \neq S - R$$

La Figura 6.4(d) muestra los nombres de los estudiantes que no son profesores, mientras que la 6.4(e) contiene los profesores que no son estudiantes.

Observe que la INTERSECCIÓN puede expresarse en términos de unión y diferencia de conjuntos del siguiente modo:

$$R \cap S = R \cup S - (R - S) - (S - R)$$

### 6.2.2 El producto cartesiano (producto cruzado)

Ahora vamos a tratar la operación **PRODUCTO CARTESIANO (*CARTESIAN PRODUCT*)**, conocida también como **PRODUCTO CRUZADO (*CROSS PRODUCT*)** o **CONCATENACIÓN CRUZADA (*CROSS JOIN*),** que se identifica por ×. Se trata también de una operación de conjuntos binarios, aunque no es necesario que las relaciones en las que se aplica sean una unión compatible. En su forma binaria produce un nuevo elemento combinando cada miembro (tupla) de una relación (conjunto) con los de la otra. En general, el resultado de $R(A_1, A_2, \ldots, A_n) \times S(B_1, B_2, \ldots, B_m)$ es una relación $Q$ con un grado de n + m atributos $Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$, en este orden. La relación resultante $Q$ tiene una tupla por cada combinación de éstas (una para $R$ y otra para $S$). Por tanto, si $R$ tiene $n_R$ tuplas (indicado como $|R| = n_R$), y $S$ cuenta con $n_S$ tuplas, $R \times S$ tendrá $n_R * n_S$ tuplas.

La operación PRODUCTO CARTESIANO $n$-ary es una extensión del concepto indicado más arriba que produce nuevas tuplas concatenando todas las posibles combinaciones de tuplas desde n relaciones subyacentes. La operación aplicada es, por sí misma, absurda. Es útil cuando va seguida por una selección que correlacione los valores de los atributos procedentes de las relaciones componentes. Por ejemplo, suponga que

**Figura 6.4.** Las operaciones de conjunto UNIÓN, INTERSECCIÓN y MENOS. (a) Dos relaciones de unión compatible. (b) ESTUDIANTE ∪ PROFESOR. (c) ESTUDIANTE ∩ PROFESOR. (d) ESTUDIANTE − PROFESOR. (e) PROFESOR − ESTUDIANTE.

**(a)** **ESTUDIANTE**

| Nombre | Apellido |
|---|---|
| Susana | Gómez |
| Luis | Campos |
| Juan | Garrido |
| Bárbara | Durán |
| Amanda | González |
| Joaquín | Martín |
| Ernesto | Flores |

**PROFESOR**

| Nom | Apell |
|---|---|
| Antonio | Fernández |
| Ricardo | Adriano |
| Susana | Gómez |
| Francisco | Peláez |
| Luis | Campos |

**(b)**

| Nombre | Apellido |
|---|---|
| Susana | Gómez |
| Luis | Campos |
| Juan | Garrido |
| Bárbara | Durán |
| Amanda | González |
| Joaquín | Martín |
| Ernesto | Flores |
| Antonio | Fernández |
| Ricardo | Adriano |
| Francisco | Peláez |

**(c)**

| Nombre | Apellido |
|---|---|
| Susana | Gómez |
| Luis | Campos |

**(d)**

| Nombre | Apellido |
|---|---|
| Juan | Garrido |
| Bárbara | Durán |
| Amanda | González |
| Joaquín | Martín |
| Ernesto | Flores |

**(e)**

| Nom | Apell |
|---|---|
| Antonio | Fernández |
| Ricardo | Adriano |
| Francisco | Peláez |

queremos recuperar una lista de nombres de cada subordinado de una empleada femenina. Podemos realizar esta operación como sigue:

$$\text{EMPLEADAS_FEMENINAS} \leftarrow \sigma_{\text{Sexo ='M'}}(\text{EMPLEADO})$$
$$\text{NOMBRES_EMPLEADOS} \leftarrow \pi_{\text{Nombre, Apellido1, Dni}}(\text{EMPLEADAS_FEMENINAS})$$
$$\text{EMPLEADOS_SUBORDINADOS} \leftarrow \text{NOMBRES_EMPLEADOS} = \text{SUBORDINADO}$$
$$\text{SUBORDINADOS_ACTUALES} \leftarrow \sigma_{\text{Dni=DniEmpleado}}(\text{EMPLEADOS_SUBORDINADOS})$$
$$\text{RESULTADO} \leftarrow \pi_{\text{Nombre, Apellido1, NombreSubordinado}}(\text{SUBORDINADOS_ACTUALES})$$

Las relaciones resultantes de esta secuencia de operaciones se muestran en la Figura 6.5. EMPLEADOS_SUBORDINADOS es el resultado de aplicar el PRODUCTO CARTESIANO a los NOMBRES_EMPLEADOS de la Figura 6.5 con los SUBORDINADOs de la Figura 5.6. En EMPLEADOS_SUBORDINADOS cada tupla de NOMBRES_EMPLEADOS se combina con las de SUBORDINADO, obteniéndose un resultado que no tiene sentido. Queremos combinar una tupla de empleada femenina sólo con las de sus subordinados particulares; digamos, las tuplas de SUBORDINADO cuyos valores de DniEmpleado coincidan con el Dni de EMPLEADO. La relación SUBORDINADOS_ACTUALES consigue esto. EMPLEADOS_SUBORDINADOS es un buen ejemplo de cómo puede aplicarse correctamente el álgebra relacional para producir resultados que no tengan

**Figura 6.5.** El PRODUCTO CARTESIANO (PRODUCTO CRUZADO).

**EMPLEADAS_FEMENINAS**

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|
| Alicia | Jiménez | Celaya | 999887777 | 12-05-1968 | Gran Vía, 38 | M | 25000 | 987654321 | 4 |
| Juana | Sainz | Oreja | 987654321 | 20-06-1941 | Cerquillas, 67 | M | 43000 | 888665555 | 4 |
| Aurora | Oliva | Avezuela | 453453453 | 31-07-1972 | Antón, 6 | M | 25000 | 333445555 | 5 |

**NOMBRES_EMPLEADOS**

| Nombre | Apellido1 | Dni |
|---|---|---|
| Alicia | Jiménez | 999887777 |
| Juana | Sainz | 987654321 |
| Aurora | Oliva | 453453453 |

**EMPLEADOS_SUBORDINADOS**

| Nombre | Apellido1 | Dni | DniEmpleado | NombreSubordinado | Sexo | FechaNac | ... |
|---|---|---|---|---|---|---|---|
| Alicia | Jiménez | 999887777 | 333445555 | Ana | M | 05-04-1986 | ... |
| Alicia | Jiménez | 999887777 | 333445555 | Teodoro | H | 25-10-1983 | ... |
| Alicia | Jiménez | 999887777 | 333445555 | Ruth | M | 03-05-1958 | ... |
| Alicia | Jiménez | 999887777 | 987654321 | Augusto | H | 28-02-1942 | ... |
| Alicia | Jiménez | 999887777 | 123456789 | Miguel | H | 01-04-1988 | ... |
| Alicia | Jiménez | 999887777 | 123456789 | Ana | M | 30-12-1988 | ... |
| Alicia | Jiménez | 999887777 | 123456789 | Elisa | M | 05-05-1967 | ... |
| Juana | Sainz | 987654321 | 333445555 | Ana | M | 05-04-1986 | ... |
| Juana | Sainz | 987654321 | 333445555 | Teodoro | H | 25-10-1983 | ... |
| Juana | Sainz | 987654321 | 333445555 | Ruth | M | 03-05-1958 | ... |
| Juana | Sainz | 987654321 | 987654321 | Augusto | H | 28-02-1942 | ... |
| Juana | Sainz | 987654321 | 123456789 | Miguel | H | 04-01-1988 | ... |
| Juana | Sainz | 987654321 | 123456789 | Ana | M | 30-12-1988 | ... |
| Juana | Sainz | 987654321 | 123456789 | Elisa | M | 05-05-1967 | ... |
| Aurora | Oliva | 453453453 | 333445555 | Ana | M | 05-04-1986 | ... |
| Aurora | Oliva | 453453453 | 333445555 | Teodoro | H | 25-10-1983 | ... |
| Aurora | Oliva | 453453453 | 333445555 | Ruth | M | 03-05-1958 | ... |
| Aurora | Oliva | 453453453 | 987654321 | Augusto | H | 28-02-1942 | ... |
| Aurora | Oliva | 453453453 | 123456789 | Miguel | H | 04-01-1988 | ... |
| Aurora | Oliva | 453453453 | 123456789 | Ana | M | 30-12-1988 | ... |
| Aurora | Oliva | 453453453 | 123456789 | Elisa | M | 05-05-1967 | ... |

**Figura 6.5.** (*Continuación*).

**SUBORDINADOS_ACTUALES**

| Nombre | Apellido1 | Dni | DniEmpleado | NombreSubordinado | Sexo | FechaNac | ... |
|---|---|---|---|---|---|---|---|
| Juana | Sainz | 987654321 | 987654321 | Augusto | H | 28-02-1942 | ... |

**RESULTADO**

| Nombre | Apellido1 | NombreSubordinado |
|---|---|---|
| Juana | Sainz | Augusto |

ningún sentido. Por consiguiente, es responsabilidad del usuario aplicar a las relaciones sólo operaciones que sean coherentes.

El PRODUCTO CARTESIANO crea tuplas con los atributos combinados de ambas relaciones. Sólo podemos hacer una SELECCIÓN de tuplas de las dos relaciones especificando una condición de selección apropiada (tal y como se comentó en el ejemplo precedente). Ya que esta secuencia de PRODUCTO CARTESIANO seguido de una SELECCIÓN se emplea con mucha frecuencia para identificar y elegir tuplas de dos relaciones, existe una operación especial llamada CONCATENACIÓN que permite especificar esta secuencia como una operación única. Vamos a estudiar esta operación a continuación.

# 6.3 Operaciones relacionales binarias: CONCATENACIÓN (*JOIN*) y DIVISIÓN (*DIVISION*)

## 6.3.1 La operación CONCATENACIÓN

**CONCATENACIÓN**, especificada mediante $\bowtie$, se emplea para combinar *tuplas relacionadas* de dos relaciones en una sola. Esta operación es muy importante para cualquier base de datos relacional que cuente con más de una relación, ya que nos permite procesar relaciones entre relaciones. Para ilustrar CONCATENACIÓN, supongamos que queremos recuperar el nombre del director de cada departamento. Para ello, necesitamos combinar las tuplas de departamento y empleado cuyos valores de DniDirector y Dni, respectivamente, sean iguales. Esto se consigue mediante la operación CONCATENACIÓN y extrapolando después el resultado sobre los atributos necesarios de la siguiente forma:

$$\text{DIRECTOR_DPTO} \leftarrow \text{DEPARTAMENTO} \bowtie_{\text{DniDirector=Dni}} \text{EMPLEADO}$$
$$\text{RESULTADO} \leftarrow \pi_{\text{NombreDpto, Apellido1, Nombre}}(\text{DIRECTOR_DPTO})$$

La primera operación se ilustra en la Figura 6.6. Observe que DniDirector es una *foreign key*, y que las restricciones de integridad referencial juegan un papel a la hora de hacer la correspondencia de tuplas en la relación EMPLEADO.

La CONCATENACIÓN puede ser enunciada en términos de un PRODUCTO CARTESIANO seguido de una SELECCIÓN. Sin embargo, CONCATENACIÓN es muy importante porque se utiliza con mucha frecuencia cuando se definen consultas a la base de datos. Considere el ejemplo que mostramos anteriormente para explicar el PRODUCTO CARTESIANO, el cual incluía la siguiente secuencia de operaciones:

$$\text{EMPLEADOS_SUBORDINADOS} \leftarrow \text{NOMBRES_EMPLEADOS} \times \text{SUBORDINADO}$$
$$\text{SUBORDINADOS_ACTUALES} \leftarrow \sigma_{\text{Dni=DniEmpleado}}(\text{EMPLEADOS_SUBORDINADOS})$$

**Figura 6.6.** Resultado de la operación CONCATENACIÓN DIRECTOR_DPTO ← DEPARTAMENTO ⋈ DniDirector=DniEMPLEADO.

**DIRECTOR_DPTO**

| NombreDpto | NúmeroDpto | DniDirector | ... | Nombre | Apellido1 | Apellido2 | Dni | ... |
|---|---|---|---|---|---|---|---|---|
| Investigación | 5 | 333445555 | ... | Alberto | Campos | Sastre | 333445555 | ... |
| Administración | 4 | 987654321 | ... | Juana | Sainz | Oreja | 987654321 | ... |
| Sede Central | 1 | 888665555 | ... | Eduardo | Ochoa | Paredes | 888665555 | ... |

Esas dos operaciones pueden sustituirse por una única operación CONCATENACIÓN de la siguiente forma:

$$\text{SUBORDINADOS_ACTUALES} \leftarrow \text{NOMBRES_EMPLEADOS} \bowtie_{\text{Dni=DniEmpleado}} \text{SUBORDINADO}$$

La forma general de una CONCATENACIÓN en dos relaciones[4] $R(A_1, A_2, \ldots, A_n)$ y $S(B_1, B_2, \ldots, B_m)$ es:

$$R \bowtie_{\text{<condición de conexión>}} S$$

El resultado de la CONCATENACIÓN es una relación $Q$ de $n + m$ atributos $Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$ por este orden; $Q$ tiene una tupla por cada combinación de éstas (una para $R$ y otra para $S$) *siempre que dicha combinación satisfaga la condición de conexión*. Ésta es la principal diferencia existente entre el PRODUCTO CARTESIANO y la CONCATENACIÓN. En la CONCATENACIÓN sólo aparecen en el resultado las combinaciones de tuplas que *satisfacen la condición de conexión*, mientras que en el PRODUCTO CARTESIANO se incluyen *todas* las combinaciones de tuplas. La condición de conexión está especificada sobre los atributos de las dos relaciones $R$ y $S$ y es evaluada para cada combinación de tuplas, incluyéndose en la relación $Q$ resultante en forma de una *única tupla combinada* sólo aquéllas cuya condición de conexión se evalúe como VERDADERO. Una condición general de conexión tiene la forma:

<condición> **AND** <condición> **AND** . . . **AND** <condición>

donde cada condición es de la forma $A_i\ \theta\ B_j$, $A_i$ es un atributo de $R$, $B_j$ lo es de $S$, $A_i$ y $B_j$ tienen el mismo dominio y $\theta$ (*theta*) es uno de los operadores de comparación $\{=, <, \leq, >, \geq, \neq\}$. Una CONCATENACIÓN con una condición de conexión de este tipo recibe el nombre de **ASOCIACIÓN (*THETA JOIN*)**. Las tuplas cuyos atributos de conexión son NULL, o aquéllas cuya condición de conexión es FALSA, *no* aparecen en el resultado. En este caso, CONCATENACIÓN *no* preserva necesariamente toda la información de las relaciones participantes.

## 6.3.2 Variaciones de CONCATENACIÓN: EQUIJOIN y CONCATENACIÓN NATURAL (*NATURAL JOIN*)

El uso más habitual de CONCATENACIÓN supone el uso de condiciones de conexión sólo con comparaciones de igualdad, en cuyo caso recibe el nombre de **EQUIJOIN**. Observe que en el resultado de una EQUIJOIN siempre tenemos uno o más pares de atributos que cuentan con *valores idénticos* en cada tupla. Por ejemplo, en la Figura 6.6, DniDirector y Dni tienen valores idénticos en cada tupla de DIRECTOR_DPTO debido a la condición de conexión especificada ente estos atributos. Ya que uno de estos valores idénticos es innecesario, se creó una nueva operación llamada **CONCATENACIÓN NATURAL** (identificada por $*$) para

[4] De nuevo, observe que tanto $R$ como $S$ pueden ser cualquier relación resultante de expresiones generales de *álgebra relacional*.

deshacerse del segundo atributo superfluo en una condición EQUIJOIN.[5] La definición estándar de esta operación precisa que los dos atributos de conexión tengan el mismo nombre en ambas relaciones. Si éste no es el caso, se aplica en primer lugar una operación de renombrado.

En el siguiente ejemplo, primero renombramos el atributo NúmeroDpto de DEPARTAMENTO como NumDptoProyecto, con lo que éste y el de PROYECTO tendrán el mismo nombre, y después se aplica CONCATENACIÓN NATURAL:

$$\text{PROYECTO_DPTO} \leftarrow .\text{PROYECTO } \rho_{(\text{NombreDpto, NumDptoProyecto, DniDirector,FechaIngresoDirector})}(\text{DEPARTAMENTO})$$

Se puede realizar la misma consulta en dos pasos creando una tabla intermedia DEPT de la siguiente forma:

$$\text{DEPT} \leftarrow \rho_{(\text{NombreDpto, NumDptoProyecto, DniDirector, FechaIngresoDirector}}(\text{DEPARTAMENTO})$$
$$\text{PROYECTO_DPTO} \leftarrow \text{PROYECTO} * \text{DEPT}$$

El atributo NumDptoProyecto recibe el nombre de **atributo de conexión**. La Figura 6.7(a) muestra la relación resultante. En la relación PROYECTO_DPTO, cada tupla combina una de tipo PROYECTO con otra de DEPARTAMENTO para el departamento que controla el proyecto, aunque *sólo se mantiene un atributo de conexión*.

Si los atributos en los que se aplicará la concatenación natural ya *tienen el mismo nombre en ambas relaciones*, el renombrado es innecesario. Por ejemplo, para aplicar esta operación en los atributos NúmeroDpto de DEPARTAMENTO y LOCALIZACIONES_DPTO, es suficiente con escribir:

$$\text{LOC_DPTO} \leftarrow . \text{ DEPARTAMENTO} * \text{LOCALIZACIONES_DPTO}$$

La relación resultante aparece en la Figura 6.7(b), y en ella se combina cada departamento con sus localizaciones, exigiendo una tupla por cada una de estas localizaciones. En general, una CONCATENACIÓN NATURAL se lleva a cabo equiparando *todos* los pares de atributos que tengan el mismo nombre en las dos relaciones. Puede existir una lista de atributos de conexión para cada relación, y los pares correspondientes deben tener el mismo nombre.

De forma general, *aunque no estandarizada*, ésta es una definición de CONCATENACIÓN NATURAL:

$$Q \leftarrow R *_{(<\text{lista1}>),(<\text{lista2}>)} S$$

En este caso, <lista1> especifica una lista de $i$ atributos de $R$, mientras que <lista2> especifica una lista de $i$ atributos de $S$. Estas listas se emplean para formar condiciones de comparación coherentes entre los atributos correspondientes para, a continuación, evaluarlas juntas mediante un operador AND. Sólo se mantiene en el resultado $Q$ la lista de atributos correspondiente a la primera relación $R$ (<lista1>).

Observe que si ninguna combinación de tuplas satisface la condición de conexión, el resultado de una CONCATENACIÓN es una relación vacía. En general, si $R$ tiene $n_R$ tuplas y $S$ $n_S$, el resultado de una operación de CONCATENACIÓN $R \bowtie_{<\text{condición de conexión}>} S$ tendrá entre cero y $n_R * n_S$ tuplas. El tamaño estimado del resultado dividido entre el valor $n_R * n_S$ máximo da como resultado un cociente llamado **selectividad de concatenación (*join selectivity*)**, que es una propiedad de cada condición de conexión. Si no existe ninguna de ellas, todas las combinaciones de tuplas cualificadas y la CONCATENACIÓN degenera en un PRODUCTO CARTESIANO, llamado también PRODUCTO CRUZADO o CONCATENACIÓN CRUZADA.

Como podemos ver, la CONCATENACIÓN se emplea para combinar datos procedentes de múltiples relaciones, de forma que la información pueda presentarse en una única tabla. Estas operaciones se conocen también como **concatenaciones internas** (***inner joins***) para distinguirlas de una variación llamada *concatenaciones externas* (*outer joins)* (consulte la Sección 6.4.4). Informalmente, una *concatenación interna* es un tipo de operación de correspondencia y asociación definida formalmente como una combinación de un PRODUCTO

[5] La CONCATENACIÓN NATURAL es, básicamente, una EQUIJOIN seguida de la eliminación de los atributos superfluos.

**Figura 6.7.** Resultado de dos operaciones CONCATENACIÓN NATURAL. (a) PROYECTO_DPTO ← PROYECTO * DEPT. (b) LOC_DPTO ← DEPARTAMENTO * LOCALIZACIONES_DPTO.

**(a)**

**PROYECTO_DPTO**

| NombreProyecto | NumProyecto | UbicacionProyecto | NumDpto-Proyecto | NombreDpto | DniDirector | FechaIngresoDirector |
|---|---|---|---|---|---|---|
| ProductoX | 1 | Valencia | 5 | Investigación | 333445555 | 22-05-1988 |
| ProductoY | 2 | Sevilla | 5 | Investigación | 333445555 | 22-05-1988 |
| ProductoZ | 3 | Madrid | 5 | Investigación | 333445555 | 22-05-1988 |
| Computación | 10 | Gijón | 4 | Administración | 987654321 | 01-01-1995 |
| Reorganización | 20 | Madrid | 1 | Sede Central | 888665555 | 19-06-1981 |
| Comunicaciones | 30 | Gijón | 4 | Administración | 987654321 | 01-01-1995 |

**(b)**

**LOC_DPTO**

| NombreDpto | NúmeroDpto | DniDirector | FechaIngresoDirector | Lugar |
|---|---|---|---|---|
| Sede Central | 1 | 888665555 | 19-06-1981 | Madrid |
| Administración | 4 | 987654321 | 01-01-1995 | Gijón |
| Investigación | 5 | 333445555 | 22-05-1988 | Valencia |
| Investigación | 5 | 333445555 | 22-05-1988 | Sevilla |
| Investigación | 5 | 333445555 | 22-05-1988 | Madrid |

CARTESIANO y una SELECCIÓN. Una *concatenación externa* es otra versión más permisiva de la otra. Observe que puede especificarse una concatenación entre una relación y ella misma (consulte la Sección 6.4.3). La CONCATENACIÓN NATURAL o la EQUIJOIN pueden establecerse también entre múltiples tablas, lo que lleva a una *concatenación de n-vías*. Por ejemplo, considere la siguiente concatenación de tres vías:

$$((\text{PROYECTO} \bowtie_{\text{NúmeroDptoProyecto=NúmeroDpto}} \text{DEPARTAMENTO}) \bowtie_{\text{DniDirector=Dni}} \text{EMPLEADO})$$

Esta operación enlaza cada proyecto con el departamento al que pertenece para, a continuación, relacionarlo con su director. La malla resultante es una relación consolidada en la que cada tupla contiene esta información proyecto-departamento-director.

### 6.3.3 Un conjunto completo de operaciones de álgebra relacional

Ya hemos visto que todas las operaciones de álgebra relacional $\{\sigma, \pi, \cup, -, \times\}$ conforman un conjunto **completo**, es decir, que cualquiera de las operaciones originales puede expresarse como una *secuencia de operaciones de este conjunto*. Por ejemplo, la INTERSECCIÓN puede expresarse usando UNIÓN y MENOS del siguiente modo:

$$R \cap S \equiv (R \cup S) - ((R - S) \cup (S - R))$$

Aunque, estrictamente hablando, la INTERSECCIÓN no es necesaria, no es conveniente especificar esta compleja operación cada vez que queramos llevar a cabo una intersección. Por otro lado, y como ya se comentó, una CONCATENACIÓN puede especificarse como un PRODUCTO CARTESIANO seguido de una SELECCIÓN:

$$R \bowtie_{\text{<condición>}} S \equiv \sigma_{\text{<condición>}}(R \times S)$$

De forma análoga, una CONCATENACIÓN NATURAL es un PRODUCTO CARTESIANO precedido por una operación RENOMBRAR y seguido de una SELECCIÓN y una PROYECCIÓN. Así pues, las distintas operaciones CONCATENACIÓN no son *estrictamente necesarias* desde el elocuente poder del álgebra relacional. Sin embargo, es importante considerarlas como operaciones separadas porque es conveniente usarlas y se utilizan mucho en las aplicaciones de bases de datos más comunes. Alguna puede considerar RENOMBRAR como una operación esencial en el caso de que la necesidad de cambiar el nombre de una expresión de álgebra relacional sea imprescindible. Otras operaciones se han incluido en el álgebra relacional por conveniencia, más que por necesidad. Trataremos una de ellas en la sección siguiente.

### 6.3.4 La operación DIVISIÓN (*DIVISION*)

La DIVISIÓN, especificada mediante ÷, es útil para cierto tipo de consultas que a veces se realizan en aplicaciones de bases de datos. Un ejemplo es, *Recuperar los nombre de los empleados que trabajan en todos los proyectos en los que también lo haga 'José Pérez'*. Para expresar esta consulta usando una DIVISIÓN, proceda del siguiente modo. Primero, recupere en la relación intermedia PEREZ_PNOS la lista de números de proyecto en los que trabaja 'José Pérez':

$$\text{PEREZ} \leftarrow .\sigma_{\text{Nombre='Jo´se' AND Apellido1='Pérez'}}(\text{EMPLEADO})$$
$$\text{PEREZ_PNOS} \leftarrow \pi_{\text{NumProy}}(\text{TRABAJA_EN} \bowtie_{\text{DniEmpleado=Dni}} \text{PEREZ})$$

A continuación, cree una relación que incluya una tupla <NumProy, DniEmpleado> siempre que el empleado cuyo Dni es DniEmpleado trabaje en el proyecto cuyo número es NumProy en la relación intermedia DNI_PNOS:

$$\text{DNI_PNOS} \leftarrow \pi_{\text{DniEmpleado, NumProy}}(\text{TRABAJA_EN})$$

Por último, aplique la DIVISIÓN a ambas relaciones, lo que nos facilita los Números Nacionales de Identidad de los empleados que queremos:

$$\text{DNIS(Dni)} \leftarrow \text{DNI_PNOS} \div \text{PEREZ_PNOS}$$
$$\text{RESULTADO} \leftarrow \pi_{\text{Nombre, Apellido1}}(\text{DNIS} * \text{EMPLEADO})$$

Las operaciones precedentes aparecen en la Figura 6.8(a).

En general, la operación DIVISIÓN se aplica a dos relaciones $R(Z) \div S(X)$, donde $X \subseteq Z$. Permite $Y = Z - X$ (y, por tanto, $Z = X \cup Y$); es decir, consiente que $Y$ sea el conjunto de atributos de $R$ que no lo son de S. El resultado de una DIVISIÓN es una relación $T(Y)$ que incluye una tupla $t$ si las tuplas $t_R$ aparecen en $R$ con $t_R[Y] = t$, y con $t_R[X] = t_S$ para *cada* tupla $t_S$ en $S$. Esto significa que, para que una tupla $t$ aparezca en el resultado $T$ de la DIVISIÓN, los valores de aquélla deben aparecer en $R$ en combinación con cada tupla en $S$. Observe que en la formulación de la operación DIVISIÓN, las tuplas de la relación denominador restringen la relación numerador seleccionando aquellas tuplas del resultado que sean iguales a todos los valores presentes en el denominador. No es necesario saber qué valores son los que están presentes.

La Figura 6.8(b) ilustra una operación DIVISIÓN en la que $X = \{A\}$, $Y = \{B\}$ y $Z = \{A, B\}$. Observe que la tuplas (valores) $b_1$ y $b_4$ aparecen en $R$ en combinación con las tres de $S$; este es el motivo por el que aparecen en la relación resultante $T$. El resto de valores de $B$ en $R$ no están en todas las tuplas de $S$, por lo que no se seleccionan: $b_2$ no aparece con $a_2$ ni $b_3$ con $a_1$.

La DIVISIÓN puede expresarse como una secuencia de operaciones $\pi$, $\times$ y $-$ del siguiente modo:

$$T1 \leftarrow \pi_Y(R)$$
$$T2 \leftarrow \pi_Y((S \times T1) - R)$$
$$T \leftarrow T1 - T2$$

**Figura 6.8.** La operación DIVISIÓN. (a) Dividiendo DNI_PNOS entre PEREZ_PNOS. (b) $T \leftarrow R \div S$.

**(a)**

**DNI_PNOS**

| DniEmpleado | NumProy |
|---|---|
| 123456789 | 1 |
| 123456789 | 2 |
| 666884444 | 3 |
| 453453453 | 1 |
| 453453453 | 2 |
| 333445555 | 2 |
| 333445555 | 3 |
| 333445555 | 10 |
| 333445555 | 20 |
| 999887777 | 30 |
| 999887777 | 10 |
| 987987987 | 10 |
| 987987987 | 30 |
| 987654321 | 30 |
| 987654321 | 20 |
| 888665555 | 20 |

**PEREZ_PNOS**

| NumProy |
|---|
| 1 |
| 2 |

**DNIS**

| Dni |
|---|
| 123456789 |
| 453453453 |

**(b)**

**R**

| A | B |
|---|---|
| a1 | b1 |
| a2 | b1 |
| a3 | b1 |
| a4 | b1 |
| a1 | b2 |
| a3 | b2T |
| a2 | b3 |
| a3 | b3 |
| a4 | b3 |
| a1 | b4 |

**S**

| A |
|---|
| a1 |
| a2 |
| a3 |

**T**

| B |
|---|
| b1 |
| b4 |

La operación DIVISIÓN está definida por conveniencia para gestionar las consultas que implican una *cuantificación universal* (consulte la Sección 6.6.7) o la condición *todo*. La mayoría de implementaciones RDBMS que cuentan con SQL como lenguaje de consulta primario no implementan directamente esta operación. SQL dispone de un camino alternativo para tratar el tipo de consulta mostrado más arriba (consulte la Sección 8.5.4). La Tabla 6.1 enumera las distintas operaciones de álgebra relacional básicas que hemos visto.

## 6.3.5 Notación para los árboles de consultas

En esta sección vamos a tratar una notación usada habitualmente en sistemas relacionales para representar consultas internamente. Dicha notación recibe el nombre de árbol de consulta, o también árbol de evaluación de consulta o árbol de ejecución de consulta. Permite la ejecución de las operaciones del álgebra relacional y se utiliza como una posible estructura de datos para la representación interna de la consulta en un RDBMS.

Un árbol de consulta es una estructura de datos en árbol que se corresponde con una expresión de álgebra relacional. Representa las relaciones de entrada de la consulta como los *nodos hoja* del árbol y las operaciones como nodos internos. La ejecución de uno de estos árboles supone la ejecución de la operación de un nodo interno, siempre que estén disponibles sus operandos, para, a continuación, reemplazar ese nodo interno por la relación que resulta de la ejecución de la operación. El proceso concluye cuando se ejecuta el nodo raíz y se obtiene la relación resultante de la consulta.

La Figura 6.9 muestra un árbol de consulta para la consulta Q2: *Para cada proyecto localizado en 'Gijón', recuperar el número del mismo, el número del departamento que lo controla y la fecha de nacimiento, direc-*

**Tabla 6.1.** Operaciones del álgebra relacional.

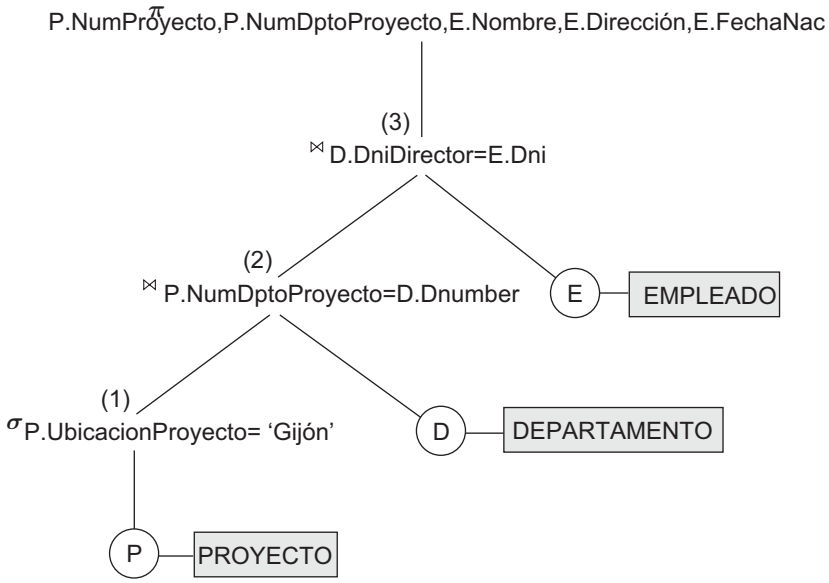
| Operación | Objetivo | Notación |
|---|---|---|
| SELECCIÓN | Selecciona todas las tuplas de una relación $R$ que satisfacen la condición de selección. | $\sigma_{<\text{condición de selección}>}(R)$ |
| PROYECCIÓN | Produce una nueva relación en la que sólo existen algunos de los atributos de $R$, y elimina las tuplas duplicadas. | $\pi_{<\text{lista de atributos}>}(R)$ |
| ASOCIACIÓN (*THETA JOIN*) | Genera todas las combinaciones de tuplas de $R_1$ y $R_2$ que satisfacen la condición de conexión. | $R_1 \bowtie_{<\text{condición de conexión}>} R_2$ |
| EQUIJOIN | Genera todas las combinaciones de tuplas de $R_1$ y $R_2$ que satisfacen una condición de conexión sólo con comparaciones de igualdad. | $R_1 \bowtie_{<\text{condición de conexión}>} R_2$ OR $R_1 \bowtie_{(<\text{atributos de conexión 1}>), (<\text{atributos de conexión 2}>)} R_2$ |
| CONCATENACIÓN NATURAL | Es lo mismo que EQUIJOIN excepto por el hecho de que los atributos de conexión de $R_2$ no están incluidos en la relación resultante; si estos atributos tienen los mismos nombres, no tienen que especificarse. | $R_1 *_{<\text{condición de conexión}>} R_2$ OR $R_1 *_{(<\text{atributos de conexión 1}>), (<\text{atributos de conexión 2}>)} R_2$ OR $R_1 * R_2$ |
| UNIÓN | Produce una relación que incluye todas las tuplas de $R_1$ o $R_2$ o de ambas; $R_1$ y $R_2$ deben ser de unión compatible. | $R_1 \cup R_2$ |
| INTERSECCIÓN | Produce una relación que incluye todas las tuplas que están en $R_1$ y $R_2$; $R_1$ y $R_2$ deben ser compatibles con la unión. | $R_1 \cap R_2$ |
| DIFERENCIA | Produce una relación que incluye todas las tuplas de $R_1$ que no están en $R_2$; $R_1$ y $R_2$ deben ser compatibles con la unión. | $R_1 - R_2$ |
| PRODUCTO CARTESIANO | Produce una relación que tiene los atributos de $R_1$ y $R_2$ e incluye tantas tuplas como posibles combinaciones de tuplas de $R_1$ y $R_2$. | $R_1$ X $R_2$ |
| DIVISIÓN | Produce una relación $R(X)$ que incluye todas las tuplas $t[X]$ en $R_1(Z)$ que aparecen en $R_1$ en combinación con cada tupla de $R_2(Y)$, donde $Z = X \cup Y$. | $R_1(Z) \div R_2(Y)$ |

*ción, nombre y apellidos de su director*. Esta consulta está especificada en el esquema relacional de la Figura 5.5 y se corresponde con la siguiente expresión de álgebra relacional:

$$\pi_{\text{NumProyecto, NumDptoProyecto, Apellido1, Dirección, FechaNac}}(((\sigma_{\text{UbicacionProyecto='Gijón'}}(\text{PROYECTO})) \bowtie_{\text{NumDptoProyecto=NúmeroDpto}}(\text{DEPARTAMENTO})) \bowtie_{\text{DniDirector=Dni}}(\text{EMPLEADO}))$$

En la Figura 6.9, las tres relaciones PROYECTO, DEPARTAMENTO y EMPLEADO están representadas por los nodos hoja P, D y E, mientras que las operaciones de álgebra relacional de la expresión lo están por tres nodos árbol internos. Esto supone el siguiente orden de ejecución de Q2. El nodo marcado como (1) en la Figura 6.9 debe ejecutarse antes que el (2) porque algunas de las tuplas resultantes de la operación (1) deben estar disponibles antes de ejecutar la operación (2). De forma análoga, el nodo (2) debe ejecutarse y producir

**Figura 6.9.** Árbol de consulta correspondiente a la expresión de álgebra de Q2.




resultados antes que lo haga el (3), y así sucesivamente. En general, un árbol de consulta ofrece una correcta representación visual y comprensión de la consulta en términos de las operaciones relacionales que usa, y está recomendado como medio adicional de expresar consultas en el álgebra relacional. Volveremos a los árboles de consulta cuando tratemos el procesamiento y la optimización de consultas en el Capítulo 15.

# 6.4 Operaciones relacionales adicionales

Existen algunas peticiones habituales a bases de datos, las cuales son necesarias en aplicaciones comerciales para los RDBMS, que no pueden llevarse a cabo con las operaciones de álgebra relacional descritas en las Secciones de la 6.1 a la 6.3. A continuación vamos a ver esas expresiones, que mejoran considerablemente la potencia del álgebra relacional original.

## 6.4.1 Proyección generalizada

La proyección generalizada es una operación que amplía las posibilidades de la proyección original permitiendo la inclusión de funciones de atributos en la lista de proyección. La forma generalizada puede expresarse del siguiente modo:

$$\pi_{F_1, F_2, \ldots, F_n}(R)$$

donde $F_1$, $F_2$, ..$F_n$ son funciones sobre los atributos de la relación $R$ y pueden involucrar constantes. Esta operación está ideada como una ayuda a la hora de desarrollar informes en los que los valores calculados deben generarse en columnas.

Como ejemplo, considere la relación:

EMPLEADO (Dni, Sueldo, Deducción, Antiguedad)

Un informe podría necesitar mostrar:

SalarioNeto = Sueldo − Deducción,

Gratificaciones = 2000 * Antiguedad e

Impuestos = 0,25 * Sueldo.

De este modo, puede usarse una proyección generalizada combinada con una operación de renombrado de la siguiente forma:

INFORME. ← $\rho_{(\text{Dni, SalarioNeto, Gratificaciones, Impuestos})}$
($\pi_{\text{Dni, Sueldo – Deducción, 2000 * Antiguedad, 0.25 * Sueldo}}$(EMPLEADO)).

## 6.4.2 Funciones de agregación y agrupamiento

Otro tipo de peticiones que no pueden expresarse a través del álgebra relacional básico son las que se utilizan para calcular **funciones matemáticas de agregación** en las colecciones de valores de la base de datos. Como ejemplos podemos citar funciones para recuperar la media o el sueldo total de todos los empleados o el número total de tuplas de empleados. Todas estas operaciones se utilizan en consultas estadísticas sencillas que resumen información procedente de las tuplas de la base de datos, y las más comunes son SUMA (SUM), MEDIA (*AVERAGE*), MÁXIMO (*MAXIMUM*) y MÍNIMO (*MINIMUM*). La función CONTAR (*COUNT*) se emplea para contar tuplas o valores.

Otro tipo común de petición supone realizar la agrupación de las tuplas de una relación por el valor de uno de sus atributos y la aplicación posterior de una función de agregación independiente a cada grupo. Un ejemplo podría ser clasificar tuplas por Dno, de modo que cada grupo incluya sólo los empleados que trabajan en el mismo departamento. A continuación queremos listar cada valor Dno junto con, por ejemplo, la media del sueldo de todos esos empleados, o el número de los que trabajan en ese departamento.

Podemos definir una operación FUNCIÓN AGREGADA (*AGGREGATE FUNCTION*) usando el símbolo $\Im$ (pronunciado *script F*)[6], para especificar este tipo de peticiones:

<atributos de agrupamiento> $\Im$ <lista de funciones> ($R$)

donde <atributos de agrupamiento> es una lista de los atributos de la relación especificados en $R$, y <lista de funciones> es una lista de parejas (<función> <atributo>). En cada una de estas parejas, <función> puede ser SUMA, MEDIA, MÁXIMO, MÍNIMO o CONTAR, mientras que <atributo> es un atributo de la relación especificada por $R$. La relación resultante cuenta con los atributos de agrupamiento además de otro por cada elemento de la lista de funciones. Por ejemplo, para recuperar cada número de departamento, el número de empleados del mismo y la media de sueldos, renombrando los atributos resultantes tal y como se especifica más adelante, podemos escribir:

$\rho_{R(\text{Dno, NumEmpleados, MediaSueldos})}$ (Dno $\Im$ COUNT Dni, AVERAGE Sueldo (EMPLEADO))

El resultado de esta operación de la relación EMPLEADO de la Figura 5.6 se muestra en la Figura 6.10(a).

En el ejemplo anterior, especificamos una lista de nombres de atributos (entre los paréntesis de la operación RENOMBRAR) para la relación resultante $R$. En caso de no aplicarse este cambio de nombre, los atributos correspondientes a la lista de funciones serán el resultado de la concatenación del nombre de la función con el del atributo en la forma <función>_<atributo>.[7] Por ejemplo, la Figura 6.10(b) muestra el resultado de la siguiente operación:

Dno $\Im$ COUNT Dni, AVERAGE Sueldo (EMPLEADO)

Si no se especifican atributos de agrupamiento, las funciones se aplican a *todas las tuplas* de la relación, por lo que obtendremos como resultado una única *tupla*. Por ejemplo, la Figura 6.10(c) muestra el resultado de la siguiente operación:

---

[6] No existe ninguna notación añadida para representar las funciones de agregación. En algunos casos se emplea un "script A".

[7] Tenga en cuenta que ésta es una notación arbitraria que hemos sugerido. No existe una notación estándar.

**Figura 6.10.** Operativa de una función de agregación.
(a) $\rho_{R(\text{Dno, NumEmpleados, MediaSueldos})}$ (Dno $\Im$ COUNT Dni, AVERAGE Sueldo (EMPLEADO)).
(b) Dno $\Im$ COUNT Dni, AVERAGE Sueldo (EMPLEADO).
(c) $\Im$ COUNT Dni, AVERAGE Sueldo (EMPLEADO).

**(a) R**

| Dno | NumEmpleados | MediaSueldos |
|---|---|---|
| 5 | 4 | 33250 |
| 4 | 3 | 31000 |
| 1 | 1 | 55000 |

**(b)**

| Dno | ContarDni | PromedioSueldo |
|---|---|---|
| 5 | 4 | 33250 |
| 4 | 3 | 31000 |
| 1 | 1 | 55000 |

**(c)**

| ContarDni | PromedioSueldo |
|---|---|
| 8 | 35125 |

$\Im$ COUNT Dni, AVERAGE Sueldo (EMPLEADO)

Es importante indicar que, en general, las duplicaciones *no se eliminan* cuando se aplica una función de agregación; de esta forma, la interpretación normal de funciones como SUMA y MEDIA es calculada.[8] Merece la pena enfatizar que el resultado de aplicar una función de agregación es una relación, y no un número escalar, aun cuando sólo tenga un valor. Esto hace del álgebra relacional un sistema cerrado.

### 6.4.3 Operaciones de cierre recursivo

Otro tipo de operación que, en general, no puede especificarse en el álgebra relacional básico, es el **cierre recursivo**. Esta operación se aplica a una **relación recursiva** entre las tuplas del mismo tipo, como la que se establece entre un empleado y un supervisor. Esta relación está descrita por la *foreign key* SuperDni de EMPLEADO de las Figuras 5.5 y 5.6, y relaciona cada tupla de empleado supervisado con otra que lo supervisa. Un ejemplo de operación recursiva es la recuperación de las supervisiones de un empleado *e* a todos sus niveles, es decir, todos los empleados *e*' supervisados directamente por *e*, todos los *e*'' que lo son por *e*', los *e*''' que los son por *e*'', y así sucesivamente.

Aunque en el álgebra relacional especificar todos los empleados supervisados por *e a un nivel específico* es directo, es complicado hacerlo a *todos* los niveles. Por ejemplo, para indicar los Dni de todos los empleados *e*' supervisados directamente (a *nivel uno*) por el empleado *e* cuyo nombre es 'Eduardo Ochoa' (consulte la Figura 5.6), podemos aplicar la siguiente operación:

DNI_OCHOA $\leftarrow \pi$Dni($\sigma_{\text{Nombre='Eduardo' AND Apellido1='Ochoa'}}$(EMPLEADO))
SUPERVISION(Dni1, Dni2) $\leftarrow \pi_{\text{Dni, SuperDni}}$(EMPLEADO)
RESULTADO1(DNI) $\leftarrow \pi$Dni1(SUPERVISION $\bowtie_{\text{Dni2=Dni}}$ DNI_OCHOA)

Para recuperar todos los empleados supervisados por Ochoa a nivel 2 (esto es, los empleados *e*'' supervisados por algún *e*' que está supervisado directamente por Ochoa) podemos aplicar otra CONCATENACIÓN al resultado de la primera consulta de la siguiente forma:

RESULTADO2(Dni) $\leftarrow \pi_{\text{Dni1}}$(SUPERVISION$\bowtie_{\text{Dni2=Dni}}$ RESULTADO1)

[8] En SQL, existe la posibilidad de eliminar duplicados antes de aplicar la función de agregación incluyendo la palabra DISTINCT (consulte la Sección 8.4.4).

**Figura 6.11.** Una consulta recursiva a dos niveles.

**SUPERVISION**

(El DNI de Ochoa es 888665555)

(Dni) (SuperDni)

| Dni1 | Dni2 |
|---|---|
| 123456789 | 333445555 |
| 333445555 | 888665555 |
| 999887777 | 987654321 |
| 987654321 | 888665555 |
| 666444444 | 333445555 |
| 453453453 | 333445555 |
| 987987987 | 987654321 |
| 888665555 | null |

**RESULTADO1**

| Dni |
|---|
| 333445555 |
| 987654321 |

(Supervisado por Ochoa)

**RESULTADO2**

| Dni |
|---|
| 123456789 |
| 999887777 |
| 666884444 |
| 453453453 |
| 987987987 |

(Supervisado por subordinados de Ochoa)

**RESULTADO**

| Dni |
|---|
| 123456789 |
| 999887777 |
| 666884444 |
| 453453453 |
| 987987987 |
| 333445555 |
| 987654321 |

(RESULTADO1 ∪ RESULTADO2)

Para obtener los conjuntos de empleados supervisados a los niveles 1 y 2 por 'Eduardo Ochoa', podemos aplicar la operación UNIÓN a los dos resultados:

RESULTADO ← RESULTADO2 ∪ RESULTADO1

Los resultados de estas consultas aparecen en la Figura 6.11. Aunque es posible recuperar los empleados de cada nivel y después realizar su UNIÓN no podemos, en general, especificar una consulta del tipo "recuperar los empleados supervisados por 'Eduardo Ochoa' a todos los niveles" sin emplear un mecanismo de bucle.[9] Se ha propuesto una operación llamada *cierre transitivo* de relaciones para procesar la relación recursiva hasta donde procede la recursión.

### 6.4.4 Operaciones CONCATENACIÓN EXTERNA (OUTER JOIN)

A continuación vamos a tratar algunas extensiones de la operación CONCATENACIÓN necesarias para especificar ciertos tipos de consultas. Las operaciones CONCATENACIÓN descritas anteriormente emparejan

[9] El estándar SQL3 incluye sintaxis para el cierre recursivo.

tuplas que satisfacen la condición de conexión. Por ejemplo, para una CONCATENACIÓN NATURAL $R * S$, sólo aparecen en el resultado las tuplas de $R$ que coinciden con las de $S$ (y viceversa). Por consiguiente, aquellas tuplas sin *coincidencia* (o *relacionadas*) se eliminan del resultado. Las tuplas con valores NULL en los atributos de conexión también se eliminan. Esto equivale a una pérdida de información en el caso de que CONCATENACIÓN se utilice para generar un informe basado en todos los datos de las relaciones.

Existe un conjunto de operaciones, llamadas **concatenaciones externas**, que pueden usarse cuando queremos mantener en el resultado todas las tuplas de $R$, o de $S$, o de ambas, independientemente de si tienen correspondencias o no en la otra relación. Esto permite ejecutar consultas en las que tuplas procedentes de dos tablas se combinan emparejando las filas correspondientes, pero sin perder aquéllas que no tienen ese "compañero". Las operaciones de concatenación descritas en la Sección 6.3, que sólo mantenían las tuplas coincidentes, reciben el nombre de **concatenaciones internas**.

Por ejemplo, supongamos que queremos una lista de todos los nombres de empleados, junto con los de los departamentos que controlan, en el *caso de que dirijan un departamento*; en caso de no hacerlo, podemos indicarlo con un valor NULL. Podemos aplicar una operación **CONCATENACIÓN EXTERNA IZQUIERDA** (***LEFT OUTER JOIN***), indicada por $\ltimes$, para recuperar el resultado:

$$\text{TEMP} \leftarrow (\textbf{EMPLEADO} \ltimes_{\text{Dni}=\text{DniDirector}} \text{DEPARTAMENTO})$$
$$\text{RESULTADO} \leftarrow \pi_{\text{Nombre, Apellido1, Apellido2, NombreDpto}}(\text{TEMP})$$

La operación CONCATENACIÓN EXTERNA IZQUIERDA mantiene cada tupla de la *primera* relación, o relación *izquierda*, $R$ en $R \ltimes S$; si no se encuentra ninguna tupla en $S$, sus atributos en la concatenación resultante se rellenan con valores NULL. La Figura 6.12 muestra el resultado de esta operación.

Una operación parecida, la **CONCATENACIÓN EXTERNA DERECHA (*RIGHT OUTER JOIN*)**, especificada por $\rtimes$, es una operación similar que mantiene cada tupla de la *segunda* relación o relación *derecha*, $S$ en el resultado de $R \rtimes S$. Una tercera operación, **CONCATENACIÓN EXTERNA COMPLETA (*FULL OUTER JOIN*)**, expresada por $\lfloor\!\rtimes\!\ltimes\!\rfloor$, mantiene todas las tuplas de ambas relaciones (las de la derecha y las de la izquierda) cuando no existen tuplas coincidentes, rellenándolas con valores NULL cuando sea necesario. Las tres operaciones forman parte del estándar SQL2 (consulte el Capítulo 8), y fueron incorporadas más tarde como una extensión del álgebra relacional en respuesta a las necesidades de las aplicaciones empresariales relacionadas con la información procedente de múltiples tablas. A veces se hace necesario generar informes de datos procedentes de varias tablas independientemente de que existan o no valores coincidentes.

**Figura 6.12.** El resultado de una operación CONCATENACIÓN EXTERNA IZQUIERDA.

**RESULTADO**

| Nombre | Apellido1 | Apellido2 | NombreDpto |
|---|---|---|---|
| José | Pérez | Pérez | NULL |
| Alberto | Campos | Sastre | Investigación |
| Alicia | Jiménez | Celaya | NULL |
| Juana | Sainz | Oreja | Administración |
| Fernando | Ojeda | Ordóñez | NULL |
| Aurora | Oliva | Avezuela | NULL |
| Luis | Pajares | Morera | NULL |
| Eduardo | Ochoa | Paredes | Sede Central |

### 6.4.5 La operación UNIÓN EXTERNA (*OUTER UNION*)

La **UNIÓN EXTERNA** fue desarrollada para obtener la unión de tuplas de dos relaciones en el caso de que esas relaciones *no sean compatibles con la unión*. Esta operación tomará la UNIÓN de tuplas de dos relaciones $R(X, Y)$ y $S(X, Z)$ que son **parcialmente compatibles**, lo que significa que sólo algunos de sus atributos son compatibles con la unión. Estos atributos sólo aparecen una vez en el resultado, y los que no son compatibles con la unión también se mantienen en la relación resultante $T(X, Y, Z)$.

Dos tuplas, $t_1$ en $R$ y $t_2$ en $S$, se dice que son **coincidentes** si $t_1[X]=t_2[X]$, y se considera que representan la misma entidad o instancia de relación. Se combinarán en una única tupla en $T$. Las tuplas de una relación que no coinciden con las de la otra relación se rellenan con valores NULL. Por ejemplo, puede aplicarse una UNIÓN EXTERNA a dos relaciones cuyos esquemas son ESTUDIANTE(Nombre, Dni, Departamento, Tutor) y PROFESOR(Nombre, Dni, Departamento, Cargo). Las tuplas de dos relaciones se emparejan en función a la misma combinación de valores de los atributos compartidos (Nombre, Dni, Departamento). La relación resultante, ESTUDIANTE_O_PROFESOR, tendrá los siguientes atributos:

ESTUDIANTE_O_PROFESOR(Nombre, Dni, Departamento, Tutor, Cargo)

Todas las tuplas de ambas relaciones están incluidas en el resultado, aunque las que tienen la combinación (Nombre, Dni, Departamento) aparecerán sólo una vez. Las contenidas solamente en la relación ESTUDIANTE tendrán NULL para el atributo Cargo, mientras que las de PROFESOR lo tendrán en Tutor. Una tupla existente en ambas relaciones, como un estudiante que también es un profesor, tendrá valor en todos sus atributos.[10]

Observe que la misma persona podría aparecer dos veces en el resultado. Por ejemplo, podríamos tener un estudiante graduado en el departamento de Matemáticas que fuera profesor del de Informática. Aunque las tuplas que representan a esa persona en ESTUDIANTE y PROFESOR tienen los mismos valores para (Nombre, Dni), no lo tienen para Departamento, lo que hará que no sean coincidentes. Esto se debe a que Departamento tiene dos significados distintos dependiendo de si se trata de un ESTUDIANTE (el departamento en el que esa persona estudia) o un PROFESOR (en el que trabaja como profesor). Si queremos unir personas en base a la misma combinación (Nombre, Dni), deberemos renombrar el atributo Departamento en cada tabla de modo que indiquen que tienen significados distintos y hacer que no formen parte de los atributos compatibles con la unión.

Otra capacidad disponible en la mayoría de lenguajes comerciales (aunque no en el álgebra relacional básica) es la posibilidad de especificar operaciones en los valores una vez extraídos de la base de datos. Por ejemplo, puede aplicarse operaciones aritméticas como +, − y ∗ a los valores numéricos que aparecen en el resultado de una consulta, tal y como ya comentamos en la Sección 6.4.1.

## 6.5 Ejemplos de consultas del álgebra relacional

A continuación vamos a ver algunos ejemplos adicionales que ilustran el uso de las operaciones de álgebra relacional. Todos ellos están referidos a la base de datos de la Figura 5.6. En general, la misma consulta puede declararse de diversas formas usando distintas operaciones. Aquí utilizaremos una de esas nomenclaturas, y le dejaremos al lector el trabajo de obtener formulaciones equivalentes.

**Consulta 1.** Recupere el nombre y la dirección de todos los empleados que trabajan en el departamento 'Investigación'.

$$\text{DPTO_INVESTIGACION} \leftarrow \sigma_{\text{NombreDpto='Investigación'}}(\text{DEPARTAMENTO})$$

$$\text{EMPS_INVESTIGACION} \leftarrow (\text{DPTO_INVESTIGACION} \rtimes_{\text{NúmeroDpto=Dno}} \text{EMPLEADO})$$

[10] Observe que UNIÓN EXTERNA es equivalente a una CONCATENACIÓN EXTERNA COMPLETA si los atributos de conexión son todos los atributos comunes de las dos relaciones.

$$\text{RESULTADO} \leftarrow \pi_{\text{Nombre, Apellido1, Dirección}}(\text{EMPS_INVESTIGACION})$$

Como una expresión única, esta consulta se convierte en:

$$\pi_{\text{Nombre, Apellido1, Dirección}}(\sigma_{\text{NombreDpto= 'Investigación'}}(\text{DEPARTAMENTO} \bowtie_{\text{NúmeroDpto=Dno}} \text{EMPLEADO}))$$

La consulta podría expresarse de otras formas; por ejemplo, podría invertirse el orden de las operaciones CONCATENACIÓN y SELECCIÓN, o CONCATENACIÓN podría sustituirse por una CONCATENACIÓN NATURAL después de cambiar el nombre a uno de los atributos de conexión.

**Consulta 2.** Por cada proyecto ubicado en 'Gijón', enumere su número, el número de departamento que lo gestiona y los apellidos, dirección y fecha de nacimiento del director del departamento.

$$\text{PROYECTOS_GIJON} \leftarrow \sigma_{\text{UbicacionProyecto='Gijón'}}(\text{PROYECTO})$$

$$\text{DEPT_CONTROL} \leftarrow (\text{PROYECTOS_GIJON} \bowtie_{\text{NumDptoProyecto=NúmeroDpto}} \text{DEPARTAMENTO})$$

$$\text{DIRECTOR_DPTO_PROYECTO?}(\text{DEPT_CONTROL} \bowtie_{\text{DniDirector=Dni}} \text{EMPLEADO})$$

$$\text{RESULTADO} \leftarrow \pi_{\text{NumProyecto, NumDptoProyecto, Apellido1, Dirección, FechaNac}}(\text{DIRECTOR_DPTO_PROYECTO})$$

**Consulta 3**. Localice los nombres de los empleados que trabajan en *todos* los proyectos gestionados por el departamento número 5.

$$\text{PROYECTOS_DEPT5(NumProy)} \leftarrow \pi_{\text{NumProyecto}}(\sigma_{\text{NumDptoProyecto=5}}(\text{PROYECTO}))$$

$$\text{EMP_PROYECTO(Dni, NumProy)} \leftarrow \pi_{\text{DniEmpleado, NumProy}}(\text{TRABAJA_EN})$$

$$\text{DNI_EMPS_RESULTADO} \leftarrow \text{EMP_PROYECTO} \div \text{PROYECTOS_DEPT5}$$

$$\text{RESULTADO} \leftarrow \pi_{\text{Apellido1, Nombre}}(\text{DNI_EMPS_RESULTADO} * \text{EMPLEADO})$$

**Consulta 4.** Haga una lista de los números de proyecto en los que esté involucrado cualquier empleado cuyo primer apellido sea 'Pérez', ya sean trabajadores o directores del departamento que gestiona ese proyecto.

$$\text{PEREZ(DniEmpleado)} \leftarrow \pi_{\text{Dni}}(\sigma_{\text{Apellido1='Pérez'}}(\text{EMPLEADO}))$$

$$\text{PROYS_PEREZ} \leftarrow \pi_{\text{NumProy}}(\text{TRABAJA_EN} * \text{PEREZ})$$

$$\text{DIRECTORES} \leftarrow \pi_{\text{Apellido1, NúmeroDpto}}(\text{EMPLEADO}_{\text{Dni=DniDirector}}\text{DEPARTAMENTO})$$

$$\text{DEPTS_ADMINISTRADOS_PEREZ(NumDptoProyecto)} \leftarrow \pi_{\text{NúmeroDpto}}(\sigma_{\text{Apellido1='Pérez'}}(\text{DIRECTORES}))$$

$$\text{DEPTS_DIRECTOR_PEREZ(NumProy)} \leftarrow \pi_{\text{NumProyecto}}(\text{DEPTS_ADMINISTRADOS_PEREZ} * \text{PROYECTO})$$

$$\text{RESULTADO} \leftarrow (\text{PROJS_PEREZ} \cup \text{DEPTS_DIRECTOR_PEREZ})$$

Como una única expresión, esta consulta se transforma en

$$\pi_{\text{NumProy}}(\text{TRABAJA_EN} \bowtie_{\text{DniEmpleado=Dni}} (\pi_{\text{Dni}}(\sigma_{\text{Apellido1='Pérez'}}(\text{EMPLEADO}))) \cup \pi_{\text{NumProy}}((\pi_{\text{NúmeroDpto}}(\sigma_{\text{Apellido1='Pérez'}}(\pi_{\text{Apellido1, NúmeroDpto}}(\text{EMPLEADO}))) \bowtie_{\text{Dni=DniDirector}} \text{DEPARTAMENTO})) \bowtie_{\text{NúmeroDpto=NumDptoProyecto}} \text{PROYECTO})$$

**Consulta 5.** Liste los nombres de todos los empleados con dos o más subordinados.

Estrictamente hablando, esta consulta no puede llevarse a cabo con el *álgebra relacional básica* (*original*). Tenemos que usar la operación FUNCIÓN AGREGADA con la función CONTAR. Asumimos que los asalariados del *mismo* empleado tienen *distinto* valor NOMBRE_SUBORDINADO.

$T1(\text{Dni, NumSubordinados}) \leftarrow_{\text{DniEmpleado}} \Im_{\text{COUNT NombreSubordinado}}(\text{SUBORDINADO})$

$T2 \leftarrow \sigma_{\text{NumSubordinados} \geqslant 2}(T1)$

$\text{RESULTADO} \leftarrow \pi_{\text{Apellido1, Nombre}}(T2 * \text{EMPLEADO})$

**Consulta 6.** Recuperar los nombres de los empleados que no tienen subordinados.

Se trata de un ejemplo del tipo de consulta que utiliza la operación MENOS (DIFERENCIA DE CONJUNTOS).

$\text{TODOS_EMPLEADOS} \leftarrow \pi_{\text{Dni}}(\text{EMPLEADO})$

$\text{EMPS_CON_SUBORDINADOS(Dni)} \leftarrow \pi_{\text{DniEmpleado}}(\text{SUBORDINADO})$

$\text{EMPS_SIN_SUBORDINADOS} \leftarrow (\text{TODOS_EMPLEADOS} - \text{EMPS_CON_SUBORDINADOS})$

$\text{RESULTADO} \leftarrow \pi_{\text{Apellido1, Nombre}}(\text{EMPS_SIN_SUBORDINADOS} * \text{EMPLEADO})$

Como una única expresión, esta consulta queda del siguiente modo:

$$\pi_{\text{Apellido1, Nombre}}((\pi_{\text{Dni}}(\text{EMPLEADO}) - \rho_{\text{Dni}}(\pi_{\text{DniEmpleado}}(\text{SUBORDINADO}))) * \text{EMPLEADO})$$

**Consulta 7.** Enumerar los nombres de los directivos que tienen, al menos, un subordinado.

$\text{DIRECTORES(Dni)} \leftarrow \pi_{\text{DniDirector}}(\text{DEPARTAMENTO})$

$\text{EMPS_CON_SUBORDINADOS(Dni)} \leftarrow \pi_{\text{DniEmpleado}}(\text{SUBORDINADO})$

$\text{DIRECTORES_CON_SUBORDINADOS} \leftarrow (\text{DIRECTORES} \cap \text{EMPS_CON_SUBORDINADOS})$

$\text{RESULTADO} \leftarrow \pi_{\text{Apellido1, Nombre}}(\text{DIRECTORES_CON_SUBORDINADOS} * \text{EMPLEADO})$

Como ya se comentó anteriormente, en general, la misma consulta puede especificarse de diferentes formas. Por ejemplo, las operaciones pueden aplicarse a menudo en órdenes diferentes. Además, algunas de ellas pueden sustituirse por otras; por ejemplo, la INTERSECCIÓN en Q7 puede sustituirse por una CONCATENACIÓN NATURAL. Como ejercicio, intente redefinir los ejemplos anteriores con operaciones diferentes.[11] Le mostramos cómo escribir consultas como expresiones simples de álgebra relacional para las consultas Q1, Q4 y Q6. Intente desarrollar las restantes como expresiones sencillas. En el Capítulo 8 y en las Secciones 6.6 y 6.7 mostramos el modo de reescribir estas consultas en otros lenguajes relacionales.

## 6.6 Cálculos relacionales de tupla

En éste y en el siguiente ejercicio vamos a estudiar otro lenguaje de consulta formal para el modelo relacional llamado **cálculo relacional**. En él escribimos una expresión **declarativa** para especificar una consulta de recuperación; por tanto, no hay una descripción del modo de evaluar una consulta. Una expresión de cálculo especifica *qué* se quiere recuperar en lugar de *cómo* hacerlo, por lo que se le considera un lenguaje **no procedural**. Esto le hace diferente del álgebra relacional, en donde debemos escribir una *secuencia de operaciones* para especificar la consulta de recuperación, razón por la que se le puede considerar como una forma **procedural** de declarar la misma. Es posible anidar operaciones de álgebra para formar una única expresión; sin embargo, siempre es necesario indicar explícitamente un cierto orden en una expresión de álgebra relacional. Este orden influye también en la estrategia de evaluación de la consulta. Una expresión de cálculo puede escribirse de diferentes formas, aunque esto no influye en el modo en que dicha consulta será evaluada.

Hemos visto también que cualquier recuperación que pueda especificarse mediante el álgebra relacional básico puede hacerse del mismo modo a través de cálculos relacionales, y viceversa; en otras palabras, la

[11] Cuando las consultas están optimizadas (consulte el Capítulo 15), el sistema elegirá una secuencia particular de operaciones que se corresponde con la mejor estrategia de ejecución.

**potencia expresiva** de ambos lenguajes es *idéntica*. Esto conduce a la definición del concepto de un lenguaje relacionalmente completo. Se considera que un lenguaje de consulta relacional *L* es **relacionalmente completo** si podemos expresar en él cualquier consulta que pueda realizarse mediante un cálculo relacional. La integridad relacional se ha convertido en una base importante para la comparación de la potencia expresiva de los lenguajes de consulta de alto nivel. Sin embargo, como ya vimos en la Sección 6.4, ciertas consultas muy frecuentes en las aplicaciones de bases de datos no pueden expresarse con ninguno de estos dos métodos. La mayoría de los lenguajes de consulta relacionales son relacionalmente completos, aunque tienen *más potencia expresiva* que el álgebra o los cálculos relacionales debido a ciertas operaciones añadidas, como las funciones agregadas, la agrupación y la ordenación.

En esta sección y en la siguiente, todos los ejemplos se refieren a la base de datos mostrada en las Figuras 5.6 y 5.7. Usaremos las mismas consultas de la Sección 6.5. Las Secciones 6.6.6, 6.6.7 y 6.6.8 tratan de los cuantificadores universales y los problemas de seguridad de una expresión. Estas secciones pueden saltárselas aquellos estudiantes interesados en una introducción general a los cálculos de tupla.

## 6.6.1 Variables de tupla y relaciones de rango

Los cálculos relacionales de tupla están basados en la especificación de un número de **variables de tupla**. Cada una de ellas suele *aplicarse sobre* una relación de base de datos particular, lo que significa que la variable podría tomar su valor de cualquier tupla individual de esa relación. Una consulta de cálculo relacional sencilla tiene la siguiente forma:

{*t* | COND(*t*)}

donde *t* es una variable de tupla y COND(*t*) es una expresión condicional que implica a *t*. El resultado es el conjunto de todas las tuplas *t* que satisfacen COND(*t*). Por ejemplo, para localizar todos los empleados cuyo salario es superior a 50.000 euros podemos escribir la siguiente expresión:

{*t* | EMPLEADO(*t*) **AND** *t*.Sueldo>50000}

La condición EMPLEADO(*t*) especifica que la **relación de rango** de la variable de tupla *t* es EMPLEADO. Cada EMPLEADO *t* que satisface la condición *t*.Sueldo>50000 será recuperada. Observe que *t*.Sueldo hace referencia al atributo Sueldo de la variable de tupla *t*; esta notación se asemeja a cómo se cualifican los nombres de atributo con los de relación, o alias, en SQL, tal y como veremos en el Capítulo 8. En la notación del Capítulo 5, *t*.Sueldo es lo mismo que decir *t* [Sueldo].

La consulta anterior recupera todos los valores de atributo de cada tupla *t* EMPLEADO seleccionada. Para obtener sólo *algunos* de los atributos (por ejemplo, el nombre y el primer apellido) escribimos:

{*t*.Nombre, *t*.Apellido1 | EMPLEADO(*t*) **AND** *t*.Sueldo>50000}

Informalmente, tenemos que especificar la siguiente información en una expresión de cálculo de tupla:

- Para cada variable de tupla *t*, la **relación de rango** *R* de *t*. Este valor se indica con una condición de la forma *R*(*t*).
- Una condición para seleccionar combinaciones de tuplas particulares. Como las variables de tupla alcanzan a sus respectivas relaciones de rango, la condición se evalúa por cada combinación posible de tuplas para identificar las **combinaciones seleccionadas** que la condición evalúa como VERDADERO.
- El conjunto de los atributos a recuperar, los **atributos solicitados**. Los valores de estos atributos se recuperan por cada combinación de tuplas seleccionada.

Antes de entrar a comentar la sintaxis formal de los cálculos relacionales de tupla, consideremos otra consulta.

**Consulta 0.** Recuperar la fecha de nacimiento y la dirección del empleado (o empleados) cuyo nombre sea José Pérez Pérez.

**C0**: {*t*.FechaNac, *t*.Dirección | EMPLEADO(*t*) **AND** *t*.Nombre='José'
**AND** *t*.Apellido1='Pérez' **AND** *t*.Apellido2='Pérez'}

En los cálculos relacionales de tupla, primero se especifican los atributos que queremos (*t*.FechaNac y *t*.Dirección) de cada tupla *t* seleccionada. A continuación, especificamos la condición de selección seguida de una barra vertical (|). En resumen, esto significa que *t* es una tupla de la relación EMPLEADO cuyos valores de los atributos Nombre, Apellido1 y Apellido2 son, respectivamente, 'José', 'Pérez' y 'Pérez'.

## 6.6.2 Expresiones y fórmulas en los cálculos relacionales de tupla

Una expresión genérica de cálculo relacional de tupla tiene esta forma:

$$\{t_1.A_j, t_2.A_k, \ldots, t_n.A_m \mid \text{COND}(t_1, t_2, \ldots, t_n, t_{n+1}, t_{n+2}, \ldots, t_{n+m})\}$$

donde $t_1, t_2, \ldots, t_n, t_{n+1}, \ldots, t_{n+m}$ son variables de tupla, cada $A_i$ es un atributo de la relación a la que $t_i$ engloba y COND es una condición o fórmula[12] de cálculo relacional de tupla. Una fórmula está compuesta por alguno de los siguientes elementos de cálculo más pequeños (llamados átomos):

1. Un átomo de la forma $R(t_i)$, donde $R$ es un nombre de relación y $t_i$ es una variable de tupla. Este átomo identifica el ámbito de la variable de tupla $t_i$ como la relación cuyo nombre es $R$.
2. Un átomo de la forma $t_i.A$ **op** $t_j.B$, donde **op** es uno de los operadores de comparación del conjunto $\{=, <, \leq, >, \geq, \neq\}$, $t_i$ y $t_j$ son variables de tupla y $A$ y $B$ son atributos de las relaciones $t_i$ y $t_j$ a las que, respectivamente, engloban.
3. Un átomo de la forma $t_i.A$ **op** $c$ o $c$ **op** $t_j.B$, donde **op** es uno de los operadores de comparación del conjunto $\{=, <, \leq, >, \geq, \neq\}$, $t_i$ y $t_j$ son variables de tupla, $A$ y $B$ son atributos de las relaciones $t_i$ y $t_j$ a las que, respectivamente, engloban y $c$ es una constante.

Cada uno de los átomos anteriores se evalúa como VERDADERO o FALSO para una combinación específica de tuplas; esto recibe el nombre de **valor de verdad (o de veracidad)** de un átomo. En general, una variable de tupla *t* abarca a todas las posibles tuplas *del universo*. Para los átomos de la forma $R(t)$, si *t* está asignada a una tupla que es *miembro de la relación R especificada*, ese átomo es VERDADERO; en cualquier otro caso es FALSO. En los átomos de tipo 2 y 3, si las variables de tupla están asignadas a tuplas en las que los valores de los atributos especificados de las mismas satisfacen la condición, entonces el átomo es VERDADERO.

Una **fórmula** (condición) está compuesta por uno o varios átomos conectados mediante los operadores lógicos **AND**, **OR** y **NOT** y definidas recursivamente del siguiente modo por las Reglas 1 y 2:

- *Regla 1*: Cada átomo es una fórmula.
- *Regla 2*: Si $F_1$ y $F_2$ son fórmulas, entonces también lo son ($F_1$ **AND** $F_2$), ($F_1$ **OR** $F_2$), **NOT** ($F_1$) y **NOT** ($F_2$). Los valores de veracidad de estas fórmulas se derivan de los obtenidos para $F_1$ y $F_2$ de la siguiente forma:
  - **a.** ($F_1$ **AND** $F_2$) es VERDADERO si $F_1$ y $F_2$ lo son; en cualquier otro caso, es FALSO.
  - **b.** ($F_1$ **OR** $F_2$) es FALSO si $F_1$ y $F_2$ lo son; en cualquier otro caso, es VERDADERO.
  - **c.** **NOT** ($F_1$) es VERDADERO si $F_1$ es FALSO; es FALSO si $F_1$ es VERDADERO.
  - **d.** **NOT** ($F_2$) es VERDADERO si $F_2$ es FALSO; es FALSO si $F_2$ es VERDADERO.

## 6.6.3 Los cuantificadores Existencial y Universal

Además de los ya comentados, existen otros dos símbolos especiales llamados **cuantificadores** que pueden aparecer en las fórmulas: el **universal** (∀) y el **existencial** (∃). Los valores de comprobación de las fórmulas

[12] Llamada también WFF (Fórmula bien formada, *Well-Formed Formula*) en lógica matemática.

con estos cuantificadores están descritos en las Reglas 3 y 4; sin embargo, primero tenemos que definir los conceptos de variables de tupla libre y de tupla acotada en una fórmula. De manera informal, una variable de tupla *t* es ligada si está cuantificada, lo que significa que aparece en una cláusula ($\exists t$) o ($\forall t$); en cualquier otro caso, es libre. Formalmente, definimos en una fórmula una variable de tupla como **libre** o **acotada** según las siguientes reglas:

- Una variable de tupla en una fórmula *F* que *es un átomo* es libre en *F*.
- Una variable de tupla *t* es libre o acotada en una fórmula construida mediante conexiones lógicas [($F_1$ **AND** $F_2$), ($F_1$ **OR** $F_2$), **NOT**($F_1$) y **NOT**($F_2$)] dependiendo de su estado en $F_1$ o $F_2$. Observe que en una fórmula de la forma $F = (F_1$ **AND** $F_2)$ o $F = (F_1$ **OR** $F_2)$, una variable de tupla puede ser libre en $F_1$ y acotada en $F_2$, o viceversa; en este caso, una de ellas será acotada y la otra libre en *F*.
- Todas las ocurrencias *libres* de una variable de tupla *t* en *F* son **acotadas** en una fórmula *F'* de la forma *F'*= ($\exists$ *t*)(*F*) o *F'* = ($\forall$ *t*)(*F*). La variable de tupla es acotada al cuantificador especificado en *F'*. Por ejemplo, considere las siguientes fórmulas:

  $F_1$ : *d*.NombreDpto='Investigación'

  $F_2$ : ($\exists t$)(*d*.NúmeroDpto=*t*.Dno)

  $F_3$ : ($\forall d$)(*d*.DniDirector='333445555')

La variable de tupla *d* es libre tanto en $F_1$ como en $F_2$, mientras que es acotada respecto al cuantificador ($\forall$) en $F_3$. *t* es acotada respecto al cuantificador ($\exists$) en $F_2$.

Ahora estamos en condiciones de ofrecer las Reglas 3 y 4 para la definición de la fórmula que empezamos anteriormente:

- *Regla 3*: Si *F* es una fórmula, entonces ($\exists t$)(*F*) también lo es, donde *t* es una variable de tupla. La fórmula ($\exists t$)(*F*) es VERDADERO si *F* se evalúa como tal en *alguna* (al menos una) tupla asignada a las ocurrencias libres de *t* en *F*; en cualquier otro caso, ($\exists t$)(*F*) es FALSO.
- *Regla 4*: Si *F* es una fórmula, entonces ($\forall t$)(*F*) lo es también, donde *t* es una variable de tupla. La fórmula ($\forall t$)(*F*) es VERDADERO si *F* se evalúa como tal para *cada tupla* asignada a las ocurrencias libres de *t* en *F*; en cualquier otro caso, ($\forall t$)(*F*) es FALSO.

$\exists$ se dice que es un cuantificador existencial porque una fórmula ($\exists t$)(*F*) es VERDADERO si *existe* alguna tupla que haga que *F* sea VERDADERO. Para el cuantificador universal, ($\forall t$)(*F*) es VERDADERO si cada posible tupla que puede asignarse a las ocurrencias libres de *t* en *F* es sustituida por *t*, y *F* es VERDADERO para *cada una de estas sustituciones*. Recibe el nombre de universal o *para todos los cuantificadores* porque cada tupla *del universo de* tuplas debe hacer que *F* sea VERDADERO para que la fórmula cuantificada también lo sea.

### 6.6.4 Ejemplos de consultas utilizando el cuantificador existencial

Vamos a utilizar algunas de las consultas de la Sección 6.5 para mostrar la forma de especificarlas tanto a través del álgebra relacional como mediante el cálculo relacional. Tenga en cuenta que será más sencillo especificar alguna de ellas mediante álgebra relacional, mientras que otras lo serán a través del cálculo relacional, y viceversa.

**Consulta 1.** Liste el nombre y la dirección de todos los empleados que trabajan para el departamento 'Investigación'.

**C1**: {*t*.Nombre, *t*.Apellido1, *t*.Dirección | EMPLEADO(*t*) **AND** ($\exists d$)
(DEPARTAMENTO(*d*) **AND** *d*.NombreDpto='Investigación' **AND** *d*.NúmeroDpto=*t*.Dno)}

Las *únicas variables de tupla libres* en una expresión de cálculo relacional deben ser aquéllas que aparecen a la izquierda de la barra (|). En C1, *t* es la única variable de tupla libre; entonces es *acotada sucesivamente* para

cada tupla. Si una tupla *satisface las condiciones* especificadas en C1, se recuperan los atributos Nombre, Apellido1 y Dirección. Las condiciones EMPLEADO($t$) y DEPARTAMENTO($d$) especifican las relaciones de rango para $t$ y $d$. La condición $d$.NombreDpto='Investigación' es una **condición de selección** y se corresponde con una operación SELECCIÓN del álgebra relacional, mientras que $d$.NúmeroDpto = $t$.Dno es una **condición de concatenación** y sirve para un propósito similar a CONCATENACIÓN (consulte la Sección 6.3).

**Consulta 2.** Por cada proyecto ubicado en 'Gijón', obtenga su número, el número del departamento que lo gestiona y los apellidos, la fecha de nacimiento y la dirección del director del mismo.

**C2**: { $p$.NumProyecto, $p$.NumDptoProyecto, $m$.Apellido1, $m$.FechaNac, $m$.Dirección | PROYECTO($p$)
**AND** EMPLEADO($m$) **AND** $p$.UbicacionProyecto='Gijón'
**AND** ((∃$d$)(DEPARTAMENTO($d$)
**AND** $p$.NumDptoProyecto=$d$.NúmeroDpto **AND** $d$.DniDirector=$m$.Dni))}

En C2 existen dos variables de tupla libres, $p$ y $m$. $d$ es de tipo acotada al cuantificador existencial. La condición de la consulta se evalúa por cada combinación de tuplas asignada a $p$ y $m$; y como expulsa todas las posibles combinaciones de tuplas en las que $p$ y $m$ son acotadas, sólo se seleccionan las combinaciones que satisfacen la condición. Distintas variables de tupla de una consulta pueden alcanzar la misma relación. Por ejemplo, para especificar C8 (por cada empleado, recuperar su nombre y primer apellido y los de su supervisor inmediato), indicamos dos variables de tupla, $e$ y $s$, que trabajan sobre la relación EMPLEADO:

**C8**: {$e$.Nombre, $e$.Apellido1, $s$.Nombre, $s$.Apellido1 | EMPLEADO($e$) **AND** EMPLEADO($s$)
**AND** $e$.SuperDni=$s$.Dni}

**Consulta 3.** Enumere el nombre de todos los empleados que trabajan en *algún* proyecto controlado por el departamento 5. Esto es una variación de la consulta 3 de la Sección 6.5, donde hablábamos de *todos* los proyectos, y no de *alguno*. En este caso necesitamos dos condiciones de conexión y dos cuantificadores existenciales.

**C3**: {$e$.Apellido1, $e$.Nombre | EMPLEADO($e$)
**AND** ((∃ $x$)(∃ $w$)(PROYECTO($x$) **AND** TRABAJA_EN($w$) **AND** $x$.NumDptoProyecto=5
**AND** $w$.DniEmpleado=$e$.Dni **AND** $x$.NumProyecto=$w$.NumProy))}

**Consulta 4.** Obtenga una lista de los números de proyecto que impliquen a cualquier empleado cuyo primer apellido sea 'Pérez', independientemente de que sean trabajadores o directores del departamento que gestiona dicho proyecto.

**C4**: {$p$.NumProyecto | PROYECTO($p$) **AND** ( ( (∃ $e$)(∃ $w$)(EMPLEADO($e$)
**AND** TRABAJA_EN($w$) **AND** $w$.NumProy=$p$.NumProyecto
**AND** $e$.Apellido1='Pérez' **AND** $e$.Dni=$w$.DniEmpleado) )
**OR**
((∃ $m$)( ∃ $d$)(EMPLEADO($m$) **AND** DEPARTAMENTO($d$)
**AND** $p$.NumDptoProyecto=$d$.NúmeroDpto **AND** $d$.DniDirector=$m$.Dni
**AND** $m$.Apellido1='Pérez')))}

Compare esto con la consulta en versión álgebra relacional de la Sección 6.5. En este caso, la operación UNIÓN puede sustituirse por un OR en los cálculos relacionales. En la sección siguiente trataremos las relaciones existentes entre los cuantificadores universal y existencial y la forma de convertir uno en otro.

### 6.6.5 Notación para consultas gráficas

En esta sección vamos a describir una notación que se ha propuesto para representar internamente consultas de cálculos relaciones. La representación más neutral de una consulta recibe el nombre de **gráfico de consulta**. La Figura 6.13 muestra el gráfico de consulta para C2. Las relaciones en la consulta están representadas por **nodos de relación**, los cuales aparecen como círculos sencillos. Las constantes, que se encuentran habitualmente en las condiciones de selección de la consulta, están representadas por **nodos constantes** que tienen la forma de círculos dobles u óvalos. Las condiciones de selección y conexión están representados por los **bordes** del gráfico (véase la Figura 6.13). Por último, los atributos a recuperar de cada relación aparecen entre corchetes sobre cada una de ellas.

La representación gráfica de una consulta no incluye el orden en el que se deben ejecutar las operaciones. Sólo existe una única correspondencia gráfica con cada consulta. Aunque algunas técnicas de optimización estaban basadas en este método, es preferible el uso de árboles de consulta porque, en la práctica, el optimizador de consultas necesita ver el orden de las operaciones para ejecutar la consulta, lo que no es posible en los gráficos de consulta.

### 6.6.6 Transformación de los cuantificadores universal y existencial

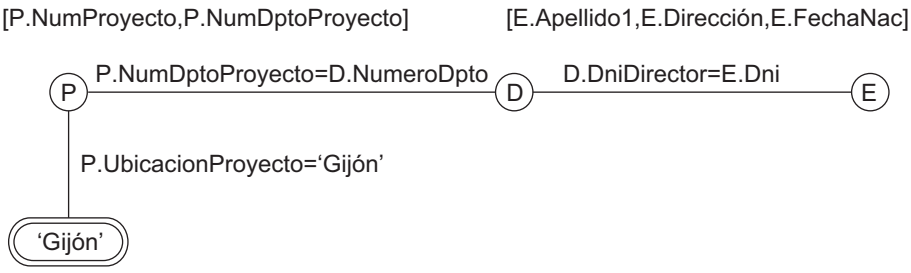
Ahora vamos a ver algunas transformaciones muy conocidas de la lógica matemática que están relacionadas con los cuantificadores universal y existencial. Es posible transformar uno en otro para obtener una expresión equivalente. De manera informal, una transformación general puede describirse como sigue: convertir un tipo de cuantificador en el otro con la negación (precedido de **NOT**); **AND** y **OR** se sustituyen uno por el otro; una fórmula negada se transforma en no-negada, y una fórmula no-negada se transforma en negada. Algunos casos especiales de esta transformación pueden declararse del siguiente modo, donde el símbolo ≡ significa "**equivalente a**":

$$(\forall x)\,(P(x)) \equiv \textbf{NOT}\ (\exists x)\ (\textbf{NOT}\ (P(x)))$$
$$(\exists x)\,(P(x)) \equiv \textbf{NOT}\ (\forall x)\ (\textbf{NOT}\ (P(x)))$$
$$(\forall x)\,(P(x))\ \textbf{AND}\ Q(x)) \equiv \textbf{NOT}\ (\exists x)\ (\textbf{NOT}\ (P(x))\ \textbf{OR NOT}\ (Q(x)))$$
$$(\forall x)\,(P(x))\ \textbf{OR}\ Q(x)) \equiv \textbf{NOT}\ (\exists x)\ (\textbf{NOT}\ (P(x))\ \textbf{AND NOT}\ (Q(x)))$$
$$(\exists x)\,(P(x))\ \textbf{OR}\ Q(x)) \equiv \textbf{NOT}\ (\forall x)\ (\textbf{NOT}\ (P(x))\ \textbf{AND NOT}\ (Q(x)))$$
$$(\exists x)\,(P(x))\ \textbf{AND}\ Q(x)) \equiv \textbf{NOT}\ (\forall x)\ (\textbf{NOT}\ (P(x))\ \textbf{OR NOT}\ (Q(x)))$$

Observe también que lo siguiente es VERDADERO, donde el símbolo ⇒ significa "**implica**":

$$(\forall x)\,(P(x)) \Rightarrow (\exists x)\,(P(x))$$
$$\textbf{NOT}\ (\exists X)\,(P(x)) \Rightarrow \textbf{NOT}\ (\forall x)\,(P(x))$$

**Figura 6.13.** Gráfico de consulta para C2.



### 6.6.7 Uso del cuantificador universal

Siempre que usemos un cuantificador universal, es bastante juicioso seguir ciertas reglas que garanticen que nuestra expresión tenga sentido. Trataremos estas reglas respecto a la C3.

**Consulta 3.** Enumere los nombres de los empleados que trabajan en *todos* los proyectos controlados por el departamento 5. Una manera de especificar esta consulta es mediante el cuantificador universal:

**C3:** {*e*.Apellido1, *e*.Nombre | EMPLEADO(*e*) **AND** ((**∀** *x*)(**NOT**(PROYECTO(*x*))
**OR NOT** (*x*.NumDptoProyecto=5) **OR** ((∃ *w*)(TRABAJA_EN(*w*) **AND** *w*.DniEmpleado= *e*.Dni
**AND** *x*.NumProyecto=*w*.NumProy))))}

Podemos dividir la C3 anterior en sus componentes básicos:

**C3:** {*e*.Apellido1, *e*.Nombre | EMPLEADO(*e*) **AND** *F* '}
*F* '= ((**∀** *x*)(**NOT**(PROYECTO(*x*)) **OR** $F_1$))
$F_1$ = **NOT**(*x*.NumDptoProyecto=5) **OR** $F_2$
$F_2$ = ((∃ *w*)(TRABAJA_EN(*w*) **AND** *w*.DniEmpleado= *e*.Dni
**AND** *x*.NumProyecto= *w*.NumProy))

Queremos asegurarnos de que un empleado *e* seleccionado trabaja en *todos los proyectos* controlados por el departamento 5, pero la definición de cuantificador universal dice que para que la fórmula cuantificada sea VERDADERA, la fórmula interna debe serlo también para *todas las tuplas del universo*. El truco consiste en excluir de la cuantificación universal aquellas tuplas en las que no estamos interesados, haciendo que la condición sea VERDADERA *para todas esas tuplas*. Esto es necesario porque una variable de tupla cuantificada universalmente, como lo es *x* en C3, debe evaluarse como VERDADERO *para cada posible tupla* asignada a ella de modo que la fórmula cuantificada también lo sea. Las primeras tuplas a excluir (haciendo que se evalúen automáticamente como VERDADERO) son aquéllas que no están en la relación *R*. En C3, el uso de la expresión **NOT**(PROYECTO(*x*)) dentro de la fórmula cuantificada universalmente evalúa como VERDADERO todas las tuplas *x* que no están en la relación PROYECTO. A continuación, excluimos de la propia *R* las tuplas que no nos interesan. En C3, la expresión **NOT**(*x*.NumDptoProyecto=5) evalúa como VERDADERO todas las tuplas *x* que están en la relación PROYECTO pero que no están controladas por el departamento 5. Para terminar, especificamos una condición $F_2$ que debe abarcar el resto de tuplas de *R*. Por consiguiente, podemos explicar C3 del siguiente modo:

1. Para que la fórmula *F*' = (∀*x*)(*F*) sea VERDADERA, la fórmula *F* debe serlo también *para todas las tuplas del universo que puedan asignarse a x*. Sin embargo, en C3 sólo nos interesa que *F* sea VERDADERO para todas las tuplas de la relación PROYECTO que están controladas por el departamento 5. Por tanto, la fórmula *F* tiene la forma (**NOT**(PROYECTO(*x*)) **OR** $F_1$). La condición '**NOT**(PROYECTO(*x*)) **OR** . . .' es VERDADERA para todas las tuplas *que no estén en la relación PROYECTO* y tiene el efecto de eliminar esas tuplas del valor de veracidad de $F_1$. Para cada tupla de la relación PROYECTO, $F_1$ debe ser VERDADERO si *F*' lo es.
2. Usando la misma línea de razonamiento, no queremos considerar las tuplas de PROYECTO que no están controladas por el departamento número 5, ya que sólo nos interesan aquéllas cuyo NumDptoProyecto=5. Por consiguiente, podemos escribir:

   **IF** (*x*.NumDptoProyecto=5) **THEN** $F_2$

   lo que es equivalente a:

   (**NOT** (*x*.NumDptoProyecto=5) **OR** $F_2$)

3. La fórmula $F_1$, por tanto, tiene la forma **NOT**(*x*.NumDptoProyecto=5) **OR** $F_2$. En el contexto de C3, esto significa que, para una tupla *x* en la relación PROYECTO, o su NumDptoProyecto ≠ 5 o debe satisfacer $F_2$.
4. Por último, $F_2$ aporta la condición que queremos mantener para una tupla EMPLEADO seleccionada: que el empleado trabaje en *cada tupla* PROYECTO *que aún no se haya excluido.* Este tipo de tuplas de empleado son las que la consulta selecciona.

Dicho de forma sencilla, C3 podría enunciarse del siguiente modo a la hora de seleccionar una tupla EMPLEADO *e*: por cada tupla *x* en la relación PROYECTO cuyo *x*.NumDptoProyecto=5, debe existir otra tupla *w* en TRABAJA_EN en la que se cumpla que *w*.DniEmpleado=*e*.Dni y *w*.NumProy=*x*.NumProyecto. Esto es equivalente a decir que el EMPLEADO *e* trabaja en cada PROYECTO *x* del DEPARTAMENTO número 5 (¡GUAU!).

Usando la transformación general de cuantificadores de universal a existencial ofrecida en la Sección 6.6.6, podemos redefinir la consulta de C3 de la forma indicada en C3A:

**C3A:** {*e*.Apellido1, *e*.Nombre | EMPLEADO(*e*) **AND** (**NOT** (∃ *x*) (PROYECTO(*x*)
**AND** (*x*.NumDptoProyecto=5) **AND** (**NOT** (∃ *w*)(TRABAJA_EN(*w*)
**AND** *w*.DniEmpleado=*e*.Dni **AND** *x*.NumProyecto=*w*.NumProy))))}

Ahora, mostraremos algunos ejemplos adicionales de consultas que usan cuantificadores.

**Consulta 6.** Liste los nombres de los empleados que no tienen subordinados.

**C6:** {*e*.Nombre, *e*.Apellido1 | EMPLEADO(e) **AND** (**NOT** (∃ *d*)(SUBORDINADO(*d*)
**AND** *e*.Dni=*d*.DniEmpleado))}

Usando la regla de transformación general, podemos redefinir C6 del siguiente modo:

**C6A:** {*e*.Nombre, *e*.Apellido1 | EMPLEADO(*e*) **AND** ((**∀** *d*)(**NOT**(SUBORDINADO(*d*))
**OR NOT**(*e*.Dni=*d*.DniEmpleado)))}

**Consulta 7.** Liste los nombres de los directores que tienen un subordinado como mínimo.

**C7:** {*e*.Nombre, *e*.Apellido1 | EMPLEADO(*e*) **AND** ((∃ *d*)(∃ ρ)(DEPARTAMENTO(*d*)
**AND** SUBORDINADO(ρ) **AND** *e*.Dni=*d*.DniDirector **AND** ρ.DniEmpleado=*e*.Dni))}

Esta consulta se manipula interpretando *directores que tienen al menos un subordinado* como *directores para los que existe algún subordinado*.

## 6.6.8 Expresiones seguras

Siempre que usemos cuantificadores universales, existenciales o negaciones de predicado en una expresión de cálculo, debemos asegurarnos de que la expresión resultante tenga sentido. Una **expresión segura** en cálculo relacional es aquélla en la que está garantizada la recuperación de un *número finito de tuplas* como resultado; en cualquier otro caso, se dice que la expresión es **insegura**.

Por ejemplo, la expresión:

{*t* | **NOT** (EMPLEADO(*t*))}

es *insegura* porque recupera todas las tuplas del universo que *no* son tuplas EMPLEADO, las cuales son infinitamente numerosas. Si seguimos las reglas dadas anteriormente para C3, obtendremos una expresión segura usando cuantificadores universales. Podemos definir de un modo más preciso las expresiones seguras introduciendo el concepto de *dominio de una expresión de cálculo relacional de tupla:* es el conjunto de todos los

valores que podrían aparecer como constantes en la expresión o existir en cualquier tupla de las relaciones referenciadas en esa expresión. El dominio de {*t* | **NOT**(EMPLEADO(*t*))} es el conjunto de todos los valores de atributo que aparecen en alguna tupla de la relación EMPLEADO (para cualquier atributo). El dominio de la expresión C3A podría incluir todos los valores que aparecen en EMPLEADO, PROYECTO y TRABAJA_EN (unidas a través del valor 5 que aparece en la propia consulta).

Se dice que una expresión es **segura** si todos los valores de su resultado son del dominio de la misma. Observe que el resultado de {*t* | **NOT**(EMPLEADO(*t*))} es inseguro ya que, en general, incluirá tuplas (y, por tanto, valores) externas a la relación EMPLEADO; valores de este tipo no pertenecen al dominio de la expresión. El resto de nuestros ejemplos son expresiones seguras.

# 6.7 Los cálculos relacionales de dominio

Existe otro tipo de cálculo relacional llamado de dominio, o simplemente **cálculo de dominio**. Mientras que SQL (consulte el Capítulo 8), un lenguaje basado en los cálculos relacionales de tuplas, estaba en fase de desarrollo en los laboratorios de IBM Research en San José (California), QBE, otro lenguaje que estaba relacionado con los cálculos de dominio, se estaba preparando casi a la vez en el Centro de Investigación T. J. Watson de IBM en Yorktown Heights (Nueva York). Las especificaciones formales del cálculo de dominio fueron propuestas después del desarrollo del sistema QBE.

Los cálculos de dominio difieren de los de tupla en el *tipo de variables* usadas en las fórmulas: en lugar de que éstas operen sobre tuplas, lo hacen sobre valores individuales de los dominios de atributos. Para componer una relación de grado *n* para el resultado de una consulta, debe disponer de *n* de estas **variables de dominio**: una por cada atributo. Una expresión de cálculo de dominio tiene la siguiente forma:

$$\{x_1, x_2, \ldots, x_n \mid \text{CONDICIÓN}(x_1, x_2, \ldots, x_n, x_{n+1}, x_{n+2}, \ldots, x_{n+m})\}$$

donde $x_1, x_2, \ldots, x_n, x_{n+1}, x_{n+2}, \ldots, x_{n+m}$ son variables de dominio que operan sobre los dominios (de atributos), y CONDICIÓN es una **condición** o **fórmula del cálculo relacional de dominio**.

Una fórmula está compuesta por **átomos**, los cuales son algo diferentes de los del cálculo de tupla, y pueden ser uno de los siguientes:

1. Un átomo de la forma $R(x_1, x_2, \ldots, x_j)$, donde *R* es el nombre de una relación de grado *j* y cada $x_i$, $1 \leq i \leq j$, es una variable de dominio. Este átomo afirma que una lista de valores de $<x_1, x_2, \ldots, x_j>$ debe ser una tupla en la relación cuyo nombre es *R*, donde $x_i$ es el *i*enésimo valor de atributo de esa tupla. Para hacer más concisa una expresión de cálculo de dominio, podemos *quitar las comas* de una lista de variables; de este modo, podemos escribir:

   $$\{x_1, x_2, \ldots, x_n \mid R(x_1\ x_2\ x_3)\ \textbf{AND}\ \ldots\}$$

   en lugar de:

   $$\{x_1, x_2, \ldots, x_n \mid R(x_1, x_2, x_3)\ \textbf{AND}\ \ldots\}$$

2. Un átomo de la forma $x_i$ **op** $x_j$, donde **op** es uno de los operadores de comparación del conjunto $\{=, <, \leq, >, \geq, \neq\}$, y $x_i$ y $x_j$ son variables de dominio.
3. Un átomo de la forma $x_i$ **op** *c* o *c* **op** $x_j$, donde **op** es uno de los operadores de comparación del conjunto $\{=, <, \leq, >, \geq, \neq\}$, $x_i$ y $x_j$ son variables de dominio y *c* es una constante.

Como ocurre en los cálculos de tupla, los átomos se evalúan como VERDADERO o FALSO para un conjunto específico de valores, llamados **valores de verdad** de esos átomos. En el caso 1, si las variables de dominio tienen asignados valores correspondientes a una tupla de la relación *R* especificada, entonces el átomo es VERDADERO. En los casos 2 y 3, si las variables de dominio están asignadas a valores que satisfacen la condición, entonces el átomo es VERDADERO.

De forma muy parecida a como ocurre con los cálculos relaciones de tupla, las fórmulas están compuestas de átomos, variables y cuantificadores, por lo que no vamos a repetir aquí de nuevo las especificaciones de las mismas.

A continuación se detallan algunos ejemplos de consultas especificadas mediante cálculos de dominio. Usaremos letras minúsculas (*l*, *m*, *n*, . . . , *x*, *y*, *z*) para especificar las variables de dominio.

**Consulta 0.** Enumere la fecha de nacimiento y la dirección de los empleados cuyo nombre sea 'José Pérez Pérez'.

**C0:** {*uv* | (∃*q*) (∃*r*) (∃*s*) (∃*t*) (∃*w*) (∃*x*) (∃*y*) (∃*z*)
(EMPLEADO(*qrstuvwxyz*) **AND** *q*='José' **AND** *r*='Pérez' **AND** *s*='Pérez')}

Necesitamos diez variables para la relación EMPLEADO, a fin de ordenar el dominio de cada atributo. De estas diez variables *q*, *r*, *s*, . . . , *z*, sólo *u* y *v* son libres. Primero especificamos los *atributos solicitados*, FechaNac y Dirección, para las variables de dominio *u* (FECHANAC) y *v* (DIRECCIÓN). Después, indicamos la condición de selección de una tupla seguida por la barra(|): a saber, que la secuencia de valores asignados a las variables *qrstuvwxyz* son una tupla de la relación EMPLEADO y que los valores para *q* (Nombre), *r* (Apellido1) y *s* (Apellido2) son 'José', 'Pérez' y 'Pérez', respectivamente. Por conveniencia, en el resto de nuestros ejemplos sólo cuantificaremos aquellas variables que *aparezcan en una condición* (*q*, *r* y *s* en C0).[13]

Una notación alternativa, usada en QBE, para escribir esta consulta es asignar las constantes 'José', 'Pérez' y 'Pérez' directamente como se muestra en C0A. Aquí, todas las variables que no aparecen a la izquierda de la barra son cuantificadas existencial e implícitamente:[14]

**C0A:** {*uv* | EMPLEADO('José','Pérez','Pérez',*t*,*u*,*v*,*w*,*x*,*y*,*z*) }

**Consulta 1.** Recupere el nombre y la dirección de todos los empleados que trabajan en el departamento de 'Investigación'.

**C1:** {*qsv* | (∃*z*) (∃*l*) (∃*m*) (EMPLEADO(*qrstuvwxyz*) **AND**
DEPARTAMENTO(*lmno*) **AND** *l*='Investigación' **AND** *m*=*z*)}

Una condición que relaciona dos variables de dominio que trabajan sobre los atributos de dos relaciones, como ocurre con *m* = *z* en C1, es una **condición de conexión**, mientras que la que relaciona una variable de dominio con una constante, como en el caso de *l* = 'Investigación', es una **condición de selección**.

**Consulta 2.** Por cada proyecto localizado en 'Gijón', liste su número, el número de departamento que lo gestiona y el nombre, los apellidos y la fecha de nacimiento de su director.

**C2:** {*iksuv* | (∃*j*)( ∃*m*)( ∃*n*)( ∃*t*)(PROYECTO(*hijk*)
**AND** EMPLEADO(*qrstuvwxyz*) **AND** DEPARTAMENTO(*lmno*)
**AND** *k*=*m* **AND** *n*=*t* **AND** *j*='Gijón')}

**Consulta 6.** Liste los nombres de los empleados que no tienen subordinados.

**C6:** {*qs* | (∃*t*)(EMPLEADO(*qrstuvwxyz*)
**AND** (**NOT**(∃*l*)(SUBORDINADO(*lmnop*) **AND** *t*=*l*)))}

---

[13] Tenga en cuenta que la notación que usamos para cuantificar sólo las variables de dominio utilizadas actualmente en las condiciones, y para mostrar un predicado como EMPLEADO(*qrstuvwxyz*) sin separar las variables de dominio con comas, es un método abreviado que usamos por conveniencia; no es la notación formal correcta.

[14] De nuevo, ésta no es una notación formalmente correcta.

C6 puede exponerse de otro modo usando cuantificadores universales en lugar de los existenciales, como puede verse en C6A:

**C6A:** {*qs* | (∃*t*)(EMPLEADO(*qrstuvwxyz*)
**AND** ((∀*l*)(**NOT**(SUBORDINADO(*lmnop*)) **OR NOT**(*t*=*l*)))))}

**Consulta 7.** Obtenga los nombres de los directores que tengan, al menos, un subordinado.

**Q7:** {*sq* | (∃*t*)( ∃*j*)( ∃*l*)(EMPLEADO(*qrstuvwxyz*) **AND** DEPARTAMENTO(*hijk*)
**AND** SUBORDINADO(*lmnop*) **AND** *t*=*j* **AND** *l*=*t*)}

Como ya mencionamos anteriormente, es posible redefinir cualquier consulta expresada mediante álgebra relacional con cálculos relaciones de dominio o tupla. Además, cualquier *expresión segura* de estos cálculos puede expresarse también mediante álgebra relacional.

El lenguaje QBE se basaba en los cálculos relacionales de dominio, aunque esto se realizó más tarde, después de que los cálculos de dominio se hubieron formalizado. QBE fue uno de los primeros lenguajes de consulta gráficos con una sintaxis mínima desarrollados para sistemas de bases de datos. Fue elaborado por IBM Research y forma parte de la opción de interfaz QMF (*Query Management Facility*) de DB2. Ha sido imitado por otros productos comerciales. Debido a su importante posición en el campo de los lenguajes relacionales, hemos incluido una panorámica de QBE en el Apéndice D.

## 6.8 Resumen

En este Capítulo hemos presentado dos lenguajes formales del modelo relacional de datos. Se usan para manipular relaciones y producir nuevas relaciones como respuestas a consultas. Tratamos el álgebra relacional y sus operaciones, las cuales se utilizan para especificar la secuencia de operaciones para definir una consulta. A continuación, mostramos dos tipos de cálculos relaciones: los de tupla y los de dominio; son declarativos porque especifican el resultado de una consulta sin especificar el modo de producir dicho resultado.

En las Secciones de la 6.1 a la 6.3 presentamos las operaciones básicas del álgebra relacional e ilustramos los tipos de consultas para las que se usan cada una de ellas. En primer lugar, abordamos los operadores relacionales unarios SELECCIÓN y PROYECCIÓN, así como la operación RENOMBRAR. Después nos centramos en el conjunto de operaciones binarias en las que es necesario que las relaciones sean de tipo compatible con la unión: UNIÓN, INTERSECCIÓN y DIFERENCIA DE CONJUNTOS. El PRODUCTO CARTESIANO es una operación de conjuntos que puede usarse para combinar tuplas de dos relaciones que produce todas las combinaciones posibles.

Aunque en la práctica es muy raro usarlo, mostramos cómo puede usarse el PRODUCTO CARTESIANO seguido de una SELECCIÓN para definir tuplas coincidentes de dos relaciones e inducir una operación CONCATENACIÓN. Presentamos varios tipos de operaciones CONCATENACIÓN llamadas AGRUPAMIENTO, EQUIJOIN y CONCATENACIÓN NATURAL. Los árboles de consultas se definieron como una representación interna de las consultas de álgebra relacional.

Tratamos algunos tipos de consultas importantes que *no* pueden obtenerse con el álgebra relacional básico, pero que son importantes en ciertas situaciones prácticas. Presentamos la PROYECCIÓN GENERALIZADA para usar funciones de atributos en la lista de proyección y la operación FUNCIÓN AGREGADA para tratar con tipos de peticiones agregadas. Comentamos las consultas recursivas, para las que no existe soporte directo en el álgebra. A continuación nos centramos en la CONCATENACIÓN EXTERNA y la UNIÓN EXTERNA, las cuales amplían la CONCATENACIÓN y la UNIÓN y permiten preservar toda la información de las relaciones originales en el resultado.

Las dos últimas secciones tratan acerca de los conceptos básicos que se esconden tras los cálculos relacionales, los cuales están basados en la rama de la lógica matemática llamada cálculo de predicado. Existen dos tipos de cálculos relacionales: (1) el de tupla, que utiliza variables de tupla que engloban las tuplas (filas) de las relaciones, y (2) los de dominio, que emplean variables de dominio que se centran en los dominios (columnas de las relaciones). En los cálculos relacionales, una consulta se especifica en una única sentencia declarativa sin especificar ningún orden o método para la recuperación del resultado de la consulta. Por consiguiente, los cálculos relacionales se consideran a menudo como un lenguaje de mayor nivel que el álgebra relacional porque una expresión de cálculo relacional declara *qué* queremos recuperar sin preocuparse del *cómo* hacerlo.

Explicamos la sintaxis de las consultas de cálculo relacional usando tanto variables de tupla como de dominio. Presentamos los gráficos de consulta como una representación interna de las consultas en los cálculos relacionales, y tratamos también los cuantificadores existencial (∃) y universal (∀). Mostramos que las variables de cálculo relacional están ligadas a estos cuantificadores. Describimos con detalle la forma de escribir la cuantificación universal, y comentamos el problema de especificar consultas seguras cuyo resultado es finito. También vimos las reglas que rigen la transformación de un cuantificador en otro. Son estos cuantificadores los que dan potencia expresiva a los cálculos relacionales, haciéndolos equivalentes al álgebra relacional. No existen en los cálculos racionales básicos una analogía para las funciones de agrupamiento y agregación, aunque algunas extensiones las han sugerido.

## Preguntas de repaso

**6.1.** Enumere las operaciones del álgebra relacional y el objetivo de cada una de ellas.

**6.2.** ¿Qué es la compatibilidad de unión? ¿Por qué es necesario que las operaciones UNIÓN, INTERSECCIÓN y DIFERENCIA requieran que las relaciones en las que se aplican sean compatibles con la unión?

**6.3.** Comente algunos de los tipos de consultas en los que sea necesario renombrar los atributos para evitar la ambigüedad en la consulta.

**6.4.** Comente los distintos tipos de operaciones de *concatenación interna*. ¿Por qué es necesaria una ASOCIACIÓN?

**6.5.** ¿Qué papel juega el concepto de *foreign key* a la hora de especificar los tipos más comunes de operaciones de concatenación?

**6.6.** ¿Qué es la operación FUNCIÓN? ¿Para qué se utiliza?

**6.7.** ¿En qué difieren las operaciones CONCATENACIÓN EXTERNA e INTERNA? ¿Y la UNIÓN EXTERNA de la UNIÓN?

**6.8.** ¿En qué difiere el cálculo relacional del álgebra relacional, y en qué se parecen?

**6.9.** ¿En qué se diferencian los cálculos relacionales de tupla y de dominio?

**6.10.** Comente el significado del cuantificador existencial (∃) y del universal (∀).

**6.11.** Defina los siguientes términos respecto a los cálculos de tupla: *variable de tupla*, *relación de rango*, *átomo*, *fórmula* y *expresión*.

**6.12.** Defina los siguientes términos respecto a los cálculos de dominio: *variable de dominio*, *relación de rango*, *átomo*, *fórmula* y *expresión*.

**6.13.** ¿Cuál es el significado de una *expresión segura* en los cálculos relacionales?

**6.14.** ¿Cuándo se dice que un lenguaje de consulta es relacionalmente completo?

## Ejercicios

**6.15.** Muestre el resultado de las consultas de ejemplo de la Sección 6.5 aplicadas a la base de datos de la Figura 5.6.

**6.16.** Especifique las siguientes consultas del esquema de base de datos EMPRESA de la Figura 5.5 usando los operadores relacionales comentados en este capítulo. Muestre también el resultado de cada consulta aplicada a la base de datos de la Figura 5.6.

- **a.** Recupere los nombres de todos los empleados del departamento 5 que trabajan más de 10 horas por semana en el proyecto ProductoX.
- **b.** Enumere los nombres de todos los empleados que tienen un subordinado con el mismo nombre.
- **c.** Localice los nombres de todos los empleados que están supervisados directamente por 'Alberto Campos'.
- **d.** Por cada proyecto, enumere su nombre y el número total de horas semanales (de todos los empleados) dedicadas al mismo.
- **e.** Recupere los nombres de todos los empleados que trabajan en cada proyecto.
- **f.** Recupere los nombres de todos los empleados que no trabajan en ningún proyecto.
- **g.** Por cada departamento, recupere su nombre y el salario medio de todos los empleados que trabajan en él.
- **h.** Recupere el salario medio de todas las empleadas.
- **i.** Busque los nombres y las direcciones de todos los empleados que trabajan en, al menos, un proyecto localizado en Madrid, pero cuyo departamento no lo esté.
- **j.** Liste los apellidos de todos los directores de departamento que no tengan subordinados.

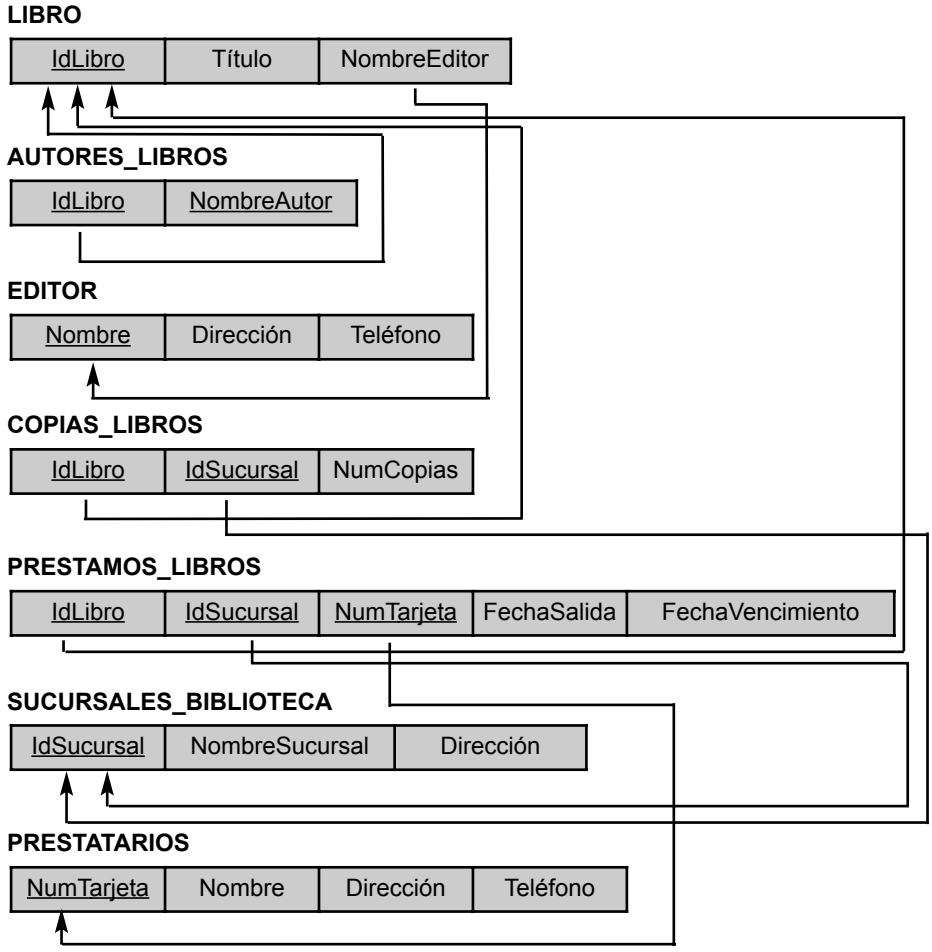
**6.17.** Considere el esquema de la base de datos LINEA_AEREA de la Figura 5.8, el cual se describió en el Ejercicio 5.12. Especifique las siguientes consultas en forma de álgebra relacional:

- **a.** Por cada vuelo, liste el número del mismo, el aeropuerto de partida del primer plan de vuelo y el aeropuerto de llegada del último.
- **b.** Enumere los números de vuelo y los días de la semana de todos los vuelos, o planes de vuelo, que salgan del Aeropuerto internacional de Houston (código de aeropuerto 'IAH') y lleguen al Aeropuerto internacional de Los Ángeles (código de aeropuerto 'LAX').
- **c.** Obtenga el número de vuelo, el código del aeropuerto de salida, la hora programada para el despegue, el código del aeropuerto de llegada, la hora programada para el aterrizaje y los días de la semana de todos los vuelos, o planes de vuelo, que salgan de algún aeropuerto de la ciudad de Houston y lleguen a alguno de Los Ángeles.
- **d.** Liste todas las tarifas del número de vuelo 'CO197'.
- **e.** Recupere el número de asientos disponibles del vuelo 'CO197' el día '09-10-1999'.

**6.18.** Considere el esquema de base de datos BIBLIOTECA de la Figura 6.14, la cual se emplea para controlar los libros, los solicitantes de los mismos y los préstamos. Las restricciones de integridad referencial aparecen como arcos dirigidos en la Figura 6.14, del mismo modo que en la notación de la Figura 5.7. Escriba expresiones relacionales para las siguientes consultas:

- **a.** ¿Cuántas copias del libro *La cabaña del tío Tom* son propiedad de la delegación cuyo nombre es 'Sharpstown'?
- **b.** ¿Cuántas copias del libro *La cabaña del tío Tom* pertenecen a cada delegación de la biblioteca?
- **c.** Recupere los nombres de todos los prestatarios que no tienen ningún libro prestado.
- **d.** Por cada libro que está prestado fuera de la delegación de Sharpstown y cuya FechaPrestamo sea hoy, recuperar el título de la obra y el nombre y la dirección del prestatario.
- **e.** Por cada delegación, recuperar el nombre de la misma y el número total de libros prestados fuera de esa delegación.
- **f.** Recuperar los nombres, direcciones y número de libros prestados de todos los prestatarios que tengan más de cinco obras prestadas.

**Figura 6.14.** Esquema de base de datos relacional para una base de datos BIBLIOTECA.




**g.** Por cada libro escrito (o coescrito) por Stephen King, recuperar el título y el número de copias de la delegación cuyo nombre es Central.

**6.19.** Especificar las siguientes consultas en álgebra relacional para el esquema de base de datos ofrecido en el Ejercicio 5.14:

**a.** Listar el NumeroPedido y la FechaSalida de todos los pedidos procedentes del NumeroAlmacen W2.

**b.** Listar la información de ALMACEN que abastece al CLIENTE José López. En el listado debe aparecer el NumeroEntrega y el NumeroAlmacen.

**c.** Obtener un listado con los datos relativos al NombreCliente, NumeroDePedidios y MediaPedidos, donde la columna central es el número total de pedidos del cliente y la última es el importe medio de ese cliente.

**d.** Enumere los pedidos que no fueron expedidos a los 30 días de su petición.

**e.** Recupere el NumeroPedido de todos los pedidos que fueron expedidos de *todos* los almacenes que la empresa tiene en Nueva York.

**6.20.** Especifique las siguientes consultas en álgebra relacional del esquema de base de datos del Ejercicio 5.15:

**a.** Obtenga todos los detalles de los viajes que sobrepasen los 2.000 euros de gastos.

**b.** Imprima el Dni del vendedor que realiza viajes a Honolulu.

**c.** Imprima los gastos totales de viajes del vendedor con DNI='234567890'.

**6.21.** Especifique las siguientes consultas en álgebra relacional en la base de datos del Ejercicio 5.16:

**a.** Liste el número de cursos recibidos por todos los estudiantes llamados Juan Pérez en el verano de 1999 (esto es, trimestre=V99).

**b.** Obtener una lista de los libros de texto (incluyendo NumeroCurso, ISBNLibro, TituloLibro) de los cursos ofrecidos por el departamento 'CC' que han usando más de dos libros.

**c.** Muestre cualquier departamento que haya adoptado libros publicados por 'AWL Publishing'.

**6.22.** Considere las dos tablas *T*1 y *T*2 mostradas en la Figura 6.15. Muestre los resultados de las siguientes operaciones:

**a.** $T1 \bowtie_{T1.P = T2.A} T2$

**b.** $T1 \bowtie_{T1.Q = T2.B} T2$

**c.** $T1 \;⟕_{T1.P = T2.A}\; T2$

d. $T1 \;⟖_{T1.Q = T2.B}\; T2$

**e.** $T1 \cup T2$

**f.** $T1 \bowtie_{(T1.P = T2.A \textbf{ AND } T1.R = T2.C)} T2$

**6.23.** Especifique las siguientes consultas en álgebra relacional sobre la base de datos del Ejercicio 5.17:

**a.** Para la vendedora 'Manuela Pris', obtenga la siguiente información de todos los coches que vendió: NumeroBastidor, Fabricante y PrecioVenta.

**b.** Liste el NumeroBastidor y el Modelo de los coches que no tengan extras.

**c.** Considere una operación de CONCATENACIÓN_ NATURAL entre VENDEDOR y VENTA. ¿Cuál es el significado de una concatenación externa izquierda para estas tablas (no cambie el orden de las relaciones)? Explíquelo con un ejemplo.

**d.** Escriba una consulta en álgebra relacional que comporte una selección y un conjunto de operación y exprese con palabras lo que dicha consulta hace.

**6.24.** Especifique las consultas a, b, c, e, f, i y j del Ejercicio 6.16 en forma de cálculo relacional de dominio y de tupla.

**6.25.** Especifique las consultas a, b, c y d del Ejercicio 6.17 en forma de cálculo relacional de dominio y de tupla.

**6.26.** Especifique las consultas c, d y f del Ejercicio 6.18 en forma de cálculo relacional de dominio y de tupla.

**6.27.** En una consulta de cálculo relacional de tupla con *n* variables de tupla, ¿cuál sería el número mínimo de condiciones de conexión típico? ¿Por qué? ¿Cuál sería el efecto de tener un número menor de condiciones de conexión?

**Figura 6.15.** Estado de una base de datos para las relaciones *T*1 y *T*2.

**TABLA T1**

| P | Q | R |
|---|---|---|
| 10 | a | 5 |
| 15 | b | 8 |
| 25 | a | 6 |

**TABLA T2**

| A | B | C |
|---|---|---|
| 10 | b | 6 |
| 25 | c | 3 |
| 10 | b | 5 |

**6.28.** Reescriba las consultas de cálculo relacional de dominio que siguen a la C0 en la Sección 6.7 en el estilo abreviado mostrado en la C0A, donde el objetivo es minimizar el número de variables de dominio escribiendo constantes donde sea posible.

**6.29.** Considere esta consulta: recupere los Dni de los empleados que trabajan en, al menos, los mismos proyectos que los que tienen el Dni=123456789. Esto puede declararse como (**FORALL** *x*) (**IF** *P* **THEN** *Q*), donde:

*x* es una variable de tupla que abarca la relación PROYECTO.

*P* ≡ EMPLEADO con Dni=123456789 trabaja en el PROYECTO *x*.

*Q* ≡ EMPLEADO *e* trabaja en el PROYECTO *x*.

Exprese la consulta en forma de cálculos relacionales de tupla usando estas reglas:

(∀ *x*)(*P*(*x*)) ≡ **NOT**(∃*x*)(**NOT**(*P*(*x*))).

(**IF** *P* **THEN** *Q*) ≡ (**NOT**(*P*) **OR** *Q*).

**6.30.** Indique cómo podría especificar las siguientes operaciones de álgebra relacional en forma de cálculos relacionales de tupla y de dominio.

**a.** $\sigma_{A=C}(R(A, B, C))$

**b.** $\pi_{<A, B>}(R(A, B, C))$

**c.** $R(A, B, C) * S(C, D, E)$

**d.** $R(A, B, C) \cup S(A, B, C)$

**e.** $R(A, B, C) \cap S(A, B, C)$

**f.** $R(A, B, C) - S(A, B, C)$

**g.** $R(A, B, C) \times S(D, E, F)$

**h.** $R(A, B) \div S(A)$

**6.31.** Sugiera las extensiones necesarias de los cálculos relacionales que permitan expresar los siguientes tipos de operaciones que fueron tratados en la Sección 6.4: (a) funciones agregadas y de agrupamiento; (b) operaciones de CONCATENACIÓN EXTERNA; (c) consultas de finalización recursivas.

**6.32.** Una consulta anidada es una consulta dentro de otra. De forma más específica puede decirse que una consulta anidada es una consulta con paréntesis cuyo resultado puede usarse como valor en muchos otros lugares, como en lugar de una relación. Especifique las siguientes consultas de la base de datos de la Figura 5.5 usando el concepto de consulta anidada y los operadores relacionales estudiados en este capítulo. Muestre también el resultado de cada consulta aplicada a la base de datos de la Figura 5.6.

**a.** Liste los nombres de todos los empleados que trabajan en el departamento que cuenta con el empleado de mayor salario de todos los trabajadores.

**b.** Obtenga los nombres de todos los empleados cuyo supervisor de su supervisor tenga como Dni el valor '888665555'.

**c.** Recupere los nombres de todos los empleados que ganen, al menos, 10.000 euros más que el empleado con el sueldo más bajo de toda la compañía.

**6.33.** Indique si las siguientes conclusiones son verdaderas o falsas:

**a.** **NOT** (*P*(*x*) **OR** *Q*(*x*)) → (**NOT** (*P*(*x*)) **AND** (**NOT** (*Q*(*x*)))

**b.** **NOT** (∃ x) (*P*(*x*)) → ∀*x* (**NOT** (*P*(*x*))

**c.** (∃ *x*) (*P*(*x*)) → ∀*x* ((*P*(*x*))

## Ejercicios de práctica

**6.34.** Especifique y ejecute las siguientes consultas en álgebra relacional usando el intérprete RA del esquema de base de datos relacional EMPRESA.

**a.** Liste los nombres de todos los empleados del departamento 5 que trabajen más de 10 horas a la semana en el proyecto ProductoX.

**b.** Obtenga los nombres de todos los empleados que tengan un subordinado con su mismo nombre.

**c.** Recupere los nombres de los empleados que están supervisados directamente por Alberto Campos.

**d.** Enumere los nombres de los empleados que trabajan en cada proyecto.

**e.** Muestre los nombres de todos los empleados que no trabajan en ningún proyecto.

**f.** Indique los nombres y las direcciones de los empleados que trabajan en, al menos, un proyecto localizado en Madrid, pero cuyo departamento no esté localizado en esa ciudad.

**g.** Muestre los nombres de los directores de departamento que no tengan subordinados.

**6.35.** Considere el siguiente esquema relacional PEDIDOS_CORREO que describe los datos de una compañía de envío de pedidos por correo.

REPUESTOS(NumeroRep, NombreRep, ACuenta, Precio, ONivel)

CLIENTES (NumeroCliente, Nombre, Direccion, CP, Telefono)

EMPLEADOS(NumeroEmpleado, NombreEmpleado, CP, Hdate)

CODIGOS_POSTALES(CP, Ciudad)

PEDIDOS(NumeroPedido, NumeroCliente, NumeroEmpleado, Recibido, Enviado)

DETALLE_PEDIDO(NumeroPedido, NumeroProyecto, Cantidad)

Los nombres de los atributos son auto explicativos: ACuenta mantiene la *cantidad entregada a cuenta*. Especifique y ejecute las siguientes consultas usando el intérprete RA del esquema PEDIDOS_CORREO.

**a.** Recupere los nombres de los repuestos que cuestan menos de 20 euros.

**b.** Recupere los nombres y las ciudades de los empleados que han efectuado pedidos de repuestos que cuestan más de 50 euros.

**c.** Recupere los números de cliente de aquéllos que vivan en el mismo código postal.

**d.** Recupere los nombres de los clientes que tengan pedidos de repuestos a representantes que vivan en Valencia.

**e.** Recupere los nombres de los clientes que hayan solicitado repuestos que cuesten menos de 20 euros.

**f.** Recupere los nombres de los clientes que no hayan hecho pedidos.

**g.** Recupere los nombres de los clientes que hayan hecho dos pedidos.

**6.36.** Considere el siguiente esquema relacional PLAN_ESTUDIOS que describe los datos del plan de estudios de un profesor concreto. (*Nota*. Los atributos A, B, C y D de CURSOS almacenan las abreviaturas de las notas).

CATALOGO(Cno, TituloCatalogo)

ESTUDIANTES(NumeroEstudiante, Nombre, Apellidos)

CURSOS(Term, NumSec, Cno, A, B, C, D)

MATRICULAS(NumeroEstudiante, Term, NumSec)

Especifique y ejecute las siguientes consultas usando el intérprete RA sobre el esquema PLAN_ESTUDIOS.

**a.** Recupere los nombres de los estudiantes matriculados en la clase Autómatas durante el tercer trimestre de 1996.

**b.** Recupere los valores de NumeroEstudiante de los matriculados en CSc226 y CSc227.

**c.** Recupere los valores de NumeroEstudiante de los matriculados en CSc226 o CSc227.

**d.** Recupere los nombres de los estudiantes que no están matriculados en ninguna clase.

**e.** Recupere los nombres de los estudiantes matriculados en todos los cursos de la tabla CATALOGO.

**6.37.** Considere una base de datos compuesta por las siguientes relaciones.

PROVEEDOR(NumeroProveedor, NombreProveedor)

REPUESTOS(NumeroRep, NombreRep)

PROYECTO(NumeroProyecto, NombreProyecto)

SUMINISTRO(NumeroProveedor, NumeroRep, NumeroProyecto)

La base de datos registra información acerca de los proveedores, repuestos y proyectos e incluye una relación ternaria entre ellos. Esta relación es de tipo muchos-muchos-muchos. Especifique y ejecute las siguientes consultas usando el intérprete RA.

**a.** Obtenga los números de repuesto que son suministrados a exactamente dos proyectos.

**b.** Obtenga los nombres de los proveedores que suministran más de dos repuestos al proyecto 'J1'.

**c.** Obtenga los números de repuesto suministrados por cada proveedor.

**d.** Obtenga los nombres de los proyectos suministrados sólo por el proveedor 'S1'.

**e.** Obtenga los nombres de los proveedores que suministran, al menos, dos repuestos diferentes a dos proyectos distintos.

**6.38.** Especifique y ejecute las siguientes consultas para la base de datos del Ejercicio 5.16 usando el intérprete RA.

**a.** Recupere los nombres de los estudiantes que están matriculados en un curso que utiliza libros de texto de la editorial Addison Wesley.

**b.** Recupere los nombres de los cursos en los que los libros se han cambiado una vez al menos.

**c.** Recupere los nombres de los departamentos que adoptan libros publicados sólo por Addison Wesley.

**d.** Recupere los nombres de los departamentos que adoptan libros escritos por Navathe y publicados por Addison Wesley.

**e.** Recupere los nombres de los estudiantes que nunca han usado un libro (en un curso) escrito por Navathe y publicado por Addison Wesley.

**6.39.** Repita los Ejercicios de prácticas del 6.34 al 6.38 en DRC (Cálculo relacional de dominio, *Domain Relational Calculus*) usando el intérprete DRC.

## Bibliografía seleccionada

Codd (1970) definió el álgebra relacional básica. Date (1983a) abordó las concatenaciones externas. El trabajo para la extensión de las operaciones relacionales fue tratado por Carlis (1986) y Ozsoyoglu y otros (1985). Cammarata y otros (1989) expande las restricciones de integridad del modelo relacional y las concatenaciones.

Codd (1971) presentó el lenguaje Alpha, que está basado en los conceptos de los cálculos relacionales de tupla. Alpha incluye también la noción de función agregada, la cual va más allá de los cálculos relacionales. La definición formal original de los cálculos relacionales fue ofrecida por Codd (1972), el cual proporcionó

también un algoritmo que transforma cualquier expresión de cálculo relacional de tupla en álgebra relacional. El QUEL (Stonebraker y otros, 1976) está basado en los cálculos relacionales de tupla, con cuantificadores existenciales implícitos, pero no cuantificadores universales, y fue implementado en el sistema Ingres como un lenguaje comercial. Ullman (1988) ofrece una prueba de la equivalencia del álgebra relacional con las expresiones seguras del cálculo relacional de dominio y de tupla. Abiteboul y otros (1995), junto con Atzeni y de Antonellis (1993), ofrecen un tratamiento detallado de los lenguajes relacionales formales.

Aunque las ideas de los cálculos relacionales de dominio fueron propuestas inicialmente en el lenguaje QBE (Zloof 1975), el concepto fue definido formalmente por Lacroix y Pirotte (1977). La versión experimental del sistema QBE está descrita en Zloof (1977). El ILL (Lacroix y Pirotte 1977a) está basado en los cálculos relacionales de dominio. Whang y otros (1990) amplía el QBE con los cuantificadores universales. Los lenguajes de consulta visual, de los que QBE es un ejemplo, se propusieron como un medio de consultar las bases de datos; varios grupos como el Visual Database Systems Workshop (por ejemplo, Arisawa y Catarci [2000] o Zhou y Pu [2002]) ofrecen una gran cantidad de propuestas para lenguajes de este tipo.

CAPÍTULO 7

# Diseño de bases de datos relacionales por mapeado ER- y EER-a-relacional

Este capítulo se centra en cómo **diseñar el esquema de una base de datos relacional** basándose en un diseño de esquema conceptual. Esto corresponde al diseño lógico de la base de datos o mapeado del modelo de datos que explicamos en la Sección 3.1 (véase la Figura 3.1). Presentamos los procedimientos para crear un esquema relacional a partir de un esquema Entidad-Relación (ER) o un esquema ER mejorado (EER). Nuestra explicación relaciona las construcciones de los modelos ER y EER, presentadas en los Capítulos 3 y 4, con las construcciones del modelo relacional, presentadas en los Capítulos 5 y 6. Muchas herramientas de ingeniería de software asistidas por computador (CASE) están basadas en los modelos ER o EER, o en otros modelos similares, como hemos explicado en los Capítulos 3 y 4. Los diseñadores de bases de datos utilizan interactivamente estas herramientas computerizadas para desarrollar un esquema ER o EER para una aplicación de base de datos. Muchas herramientas utilizan los diagramas ER o EER, o variaciones de ellos, para desarrollar gráficamente el esquema, y después lo convierten automáticamente en un esquema de base de datos relacional en el DDL de un DBMS relacional específico, empleando algoritmos parecidos a los presentados en este capítulo.

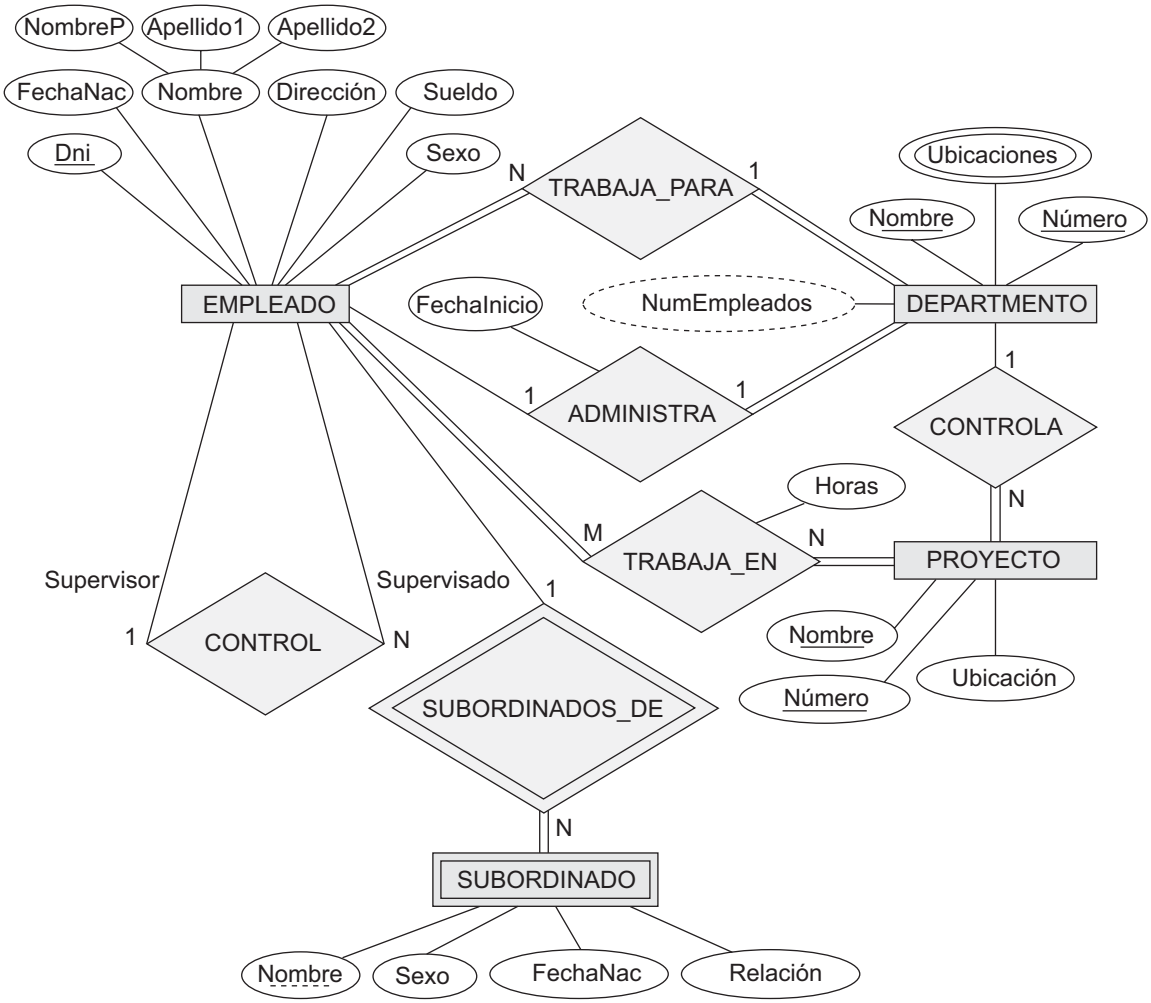
En la Sección 7.1 esbozamos un algoritmo en siete pasos para convertir las estructuras de un modelo ER básico (tipos de entidades [fuertes y débiles], relaciones binarias [con distintas restricciones estructurales], relaciones *n*-ary y atributos [simples, compuestos y multivalor]) en relaciones. Después, en la Sección 7.2, continuamos el algoritmo de mapeado describiendo cómo mapear construcciones del modelo EER (especialización/generalización y tipos de unión [categorías]) a relaciones.

## 7.1 Diseño de una base de datos relacional utilizando el mapeado ER-a-relacional

### 7.1.1 Algoritmo de mapeado ER-a-relacional

A continuación describimos los pasos de un algoritmo para el mapeado ER-a-relacional, para lo que utilizaremos la base de datos EMPRESA. El esquema ER de esta base de datos se muestra de nuevo en la Figura 7.1, mientras que en la Figura 7.2 mostramos el esquema de la base de datos relacional EMPRESA correspondiente para ilustrar los pasos del mapeado.

**Figura 7.1.** Diagrama del esquema conceptual ER para la base de datos EMPRESA.

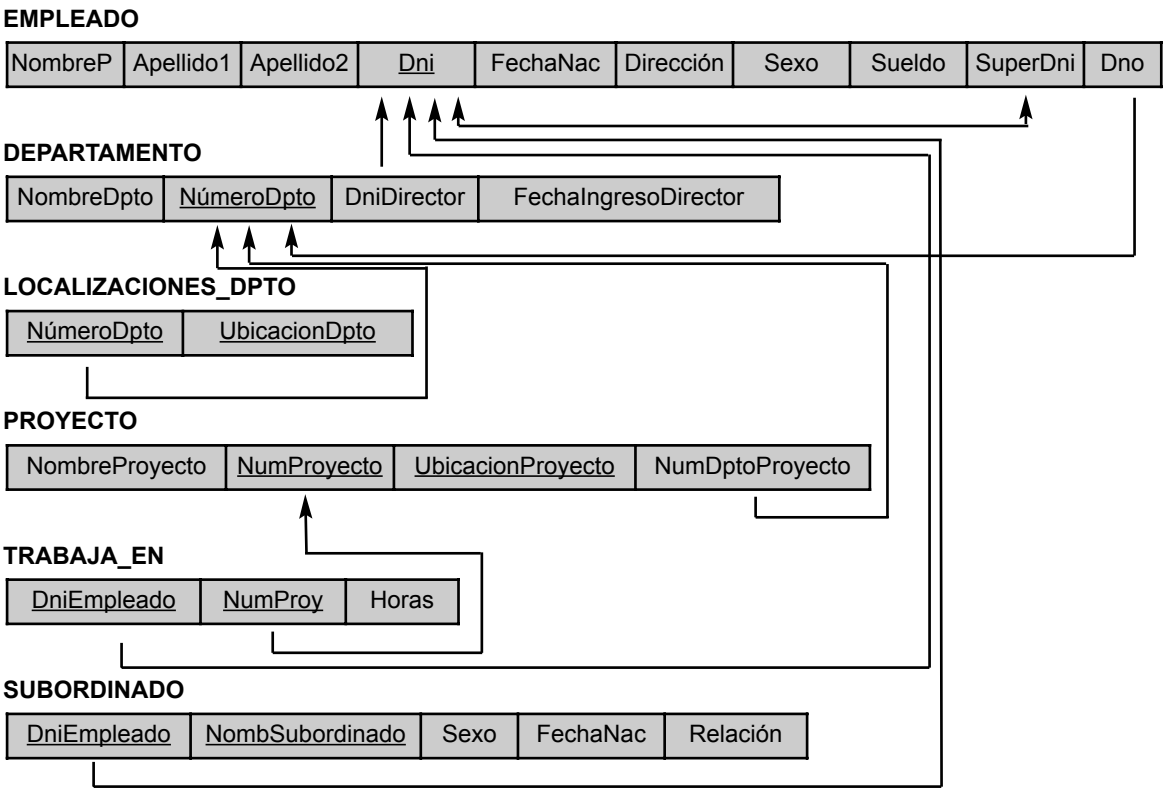



**Paso 1: Mapeado de los tipos de entidad regulares.** Por cada entidad (fuerte) regular *E* del esquema ER, cree una relación *R* que incluya todos los atributos simples de *E*. Incluya únicamente los atributos simples que conforman un atributo compuesto. Seleccione uno de los atributos clave de *E* como clave principal para *R*. Si la clave elegida de *E* es compuesta, entonces el conjunto de los atributos simples que la forman constituirán la clave principal de *R*.

Si durante el diseño conceptual se identificaron varias claves para *E*, la información que describe los atributos que forman cada clave adicional conserva su orden para especificar las claves (únicas) secundarias de la relación *R*. El conocimiento sobre las claves también es necesario para la indexación y otros tipos de análisis.

En nuestro ejemplo, creamos las relaciones EMPLEADO, DEPARTAMENTO y PROYECTO en la Figura 7.2 como correspondientes a los tipos de entidad regulares EMPLEADO, DEPARTAMENTO y PROYECTO de la Figura 7.1. La *foreign key* y los atributos de relación, si los hay, no se incluyen aún; se añadirán durante los pasos posteriores. Nos referimos a los atributos SuperDni y Dno de EMPLEADO, DniDirector y FechaIngresoDirector de DEPARTAMENTO, y NumDptoProyecto de PROYECTO. En nuestro ejemplo, seleccionamos Dni, NúmeroDpto y NumProyecto como claves principales para las relaciones EMPLEADO, DEPARTAMENTO y

**Figura 7.2.** Resultado de mapear el esquema ER de EMPRESA dentro de un esquema de base de datos relacional.




PROYECTO, respectivamente. Recuerde que NombreDpto de DEPARTAMENTO y NombreProyecto de PROYECTO son claves secundarias, pues es posible que las utilicemos más tarde en el diseño.

Las relaciones que se crean a partir del mapeado de tipos de entidad se conocen a veces como **relaciones de entidad** porque cada tupla representa una instancia de entidad. En la Figura 7.3(a) se muestra el resultado tras este paso del mapeado.

**Paso 2: Mapeado de los tipos de entidad débiles.** Por cada tipo de entidad débil *W* del esquema ER con el tipo de entidad propietario *E*, cree una relación *R* e incluya todos los atributos simples (o componentes simples de los atributos compuestos) de *W* como atributos de *R*. Además, incluya como atributos de la *foreign key* de *R*, el(los) atributo(s) de la o las relaciones que correspondan al o los tipos de entidad propietarios; esto se encarga de identificar el tipo de relación de *W*. La clave principal de *R* es la combinación de la(s) clave(s) principal(es) del o de los propietarios y la clave parcial del tipo de entidad débil *W*, si la hubiera.

Si hay un tipo de entidad débil $E_2$ cuyo propietario también es un tipo de entidad débil $E_1$, entonces $E_1$ debe asignarse antes que $E_2$ para determinar primero su clave principal.

En nuestro ejemplo, creamos la relación SUBORDINADO en este paso como correspondencia con el tipo de entidad SUBORDINADO (véase la Figura 7.3[b]). Incluimos la clave principal Dni de la relación EMPLEADO (que corresponde al tipo de entidad propietario) como un atributo *foreign key* de SUBORDINADO; lo renombramos como DniEmpleado, aunque no es necesario. La clave principal de la relación SUBORDINADO es la combinación {DniEmpleado, NombreSubordinado} porque NombreSubordinado (también renombrado a partir de Nombre en la Figura 7.1) es la clave parcial de SUBORDINADO.

**Figura 7.3.** Ilustración de algunos pasos del mapeado. (a) Relaciones de entidad después del paso 1. (b) Relación de entidad débil adicional después del paso 2. (c) Relación de relación después del paso 5. (d) Relación que representa el atributo multivalor después del paso 6.

**(a)** **EMPLEADO**

| NombreP | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo |
|---|---|---|---|---|---|---|---|

**DEPARTAMENTO**

| NombreDpto | NúmeroDpto |
|---|---|

**PROYECTO**

| NombreProyecto | NumProyecto | UbicaciónProyecto |
|---|---|---|

**(b)** **SUBORDINADO**

| DniEmpleado | NombSubordinado | Sexo | FechaNac | Relación |
|---|---|---|---|---|

**(c)** **TRABAJA_EN**

| DniEmpleado | NumProy | Horas |
|---|---|---|

**(d)** **LOCALIZACIONES_DPTO**

| NúmeroDpto | UbicaciónDpto |
|---|---|

Es normal elegir la opción de propagación (CASCADA) para la acción de activación referencial (consulte la Sección 8.2) en la *foreign key* en la relación correspondiente al tipo de entidad débil, pues la existencia de una entidad débil depende de su entidad propietaria. Esto se puede utilizar tanto para ON UPDATE como para ON DELETE.

**Paso 3: Mapeado de los tipos de relación 1:1 binaria.** Por cada tipo de relación 1:1 binaria *R* del esquema ER, identifique las relaciones *S* y *T* que corresponden a los tipos de entidad que participan en *R*. Hay tres metodologías posibles: (1) la metodología de la *foreign key*, (2) la metodología de la relación mezclada y (3) la metodología de referencia cruzada o relación de relación. La primera metodología es la más útil y la que debe seguirse salvo que se den ciertas condiciones especiales, como las que explicamos a continuación:

1. **Metodología de la *foreign key*.** Seleccione una de las relaciones (por ejemplo, *S*) e incluya como *foreign key* en *S* la clave principal de *T*. Lo mejor es elegir un tipo de entidad con *participación total* en *R* en el papel de *S*. Incluya todos los tributos simples (o los componentes simples de los atributos compuestos) del tipo de relación 1:1 *R* como atributos de *S*.

   En nuestro ejemplo, mapeamos el tipo de relación 1:1 ADMINISTRA de la Figura 7.1 eligiendo el tipo de entidad participante DEPARTAMENTO para que desempeñe el papel de *S* porque su participación en el tipo de relación ADMINISTRA es total (cada departamento tiene un director). Incluimos la clave principal de la relación EMPLEADO como *foreign key* en la relación DEPARTAMENTO y la renombramos como DniDirector. También incluimos el atributo simple FechaInicio del tipo de relación ADMINISTRA en la relación DEPARTAMENTO y lo renombramos como FechaIngresoDirector (véase la Figura 7.2).

   Es posible incluir, en lugar de esto, la clave principal de *S* como una *foreign key* en *T*. En nuestro ejemplo, esto equivale a tener un atributo de *foreign key*, digamos DepartamentoAdministrado en la relación EMPLEADO, pero tendrá un valor NULL para las tuplas empleado que no dirijan un departamento. Si

sólo el 10 por ciento de los empleados administra un departamento, entonces en este caso el 90 por ciento de las *foreign keys* serían NULL. Otra posibilidad es recurrir a la redundancia al tener *foreign keys* en las relaciones *S* y *T*, pero esto malogra el mantenimiento de la coherencia.

2. **Metodología de la relación mezclada.** Una asignación alternativa de un tipo de relación 1:1 es posible al mezclar los dos tipos de entidad y la relación en una sola relación. Esto puede ser apropiado cuando *las dos participaciones son totales.*
3. **Metodología de referencia cruzada o relación de relación.** La tercera opción consiste en configurar una tercera relación *R* con el propósito de crear una referencia cruzada de las claves principales de las relaciones *S* y *T* que representan los tipos de entidad. Como veremos, esta metodología es necesaria para las relaciones M:N binarias. La relación *R* se denomina **relación de relación** (y, en algunas ocasiones, **tabla de búsqueda**), porque cada tupla de *R* representa una instancia de relación que relaciona una tupla de *S* con otra de *T*.

**Paso 4: Mapeado de tipos de relaciones 1:N binarias.** Por cada relación 1:N binaria regular *R*, identifique la relación *S* que representa el tipo de entidad participante en el lado *N* del tipo de relación. Incluya como *foreign key* en *S* la clave principal de la relación *T* que representa el otro tipo de entidad participante en *R*; hacemos esto porque cada instancia de entidad en el lado N está relacionada, a lo sumo, con una instancia de entidad del lado 1 del tipo de relación. Incluya cualesquiera atributos simples (o componentes simples de los atributos compuestos) del tipo de relación 1:N como atributos de *S*.

En nuestro ejemplo, vamos a asignar ahora los tipos de relación 1:N TRABAJA_PARA, CONTROLA y CONTROL de la Figura 7.1. Para TRABAJA_PARA incluimos la clave principal NúmeroDpto de la relación DEPARTAMENTO como *foreign key* en la relación EMPLEADO y la denominamos Dno. En CONTROL, incluimos la clave principal de la relación EMPLEADO como *foreign key* en la propia relación EMPLEADO (porque la relación es recursiva) y la denominamos SuperDni. La relación CONTROLA está asignada al atributo de *foreign key* NumDptoProyecto de PROYECTO, que hace referencia a la clave principal NúmeroDpto de la relación DEPARTAMENTO. Estas *foreign keys* se muestran en la Figura 7.2.

De nuevo, una metodología alternativa es la opción de relación de relación (referencia cruzada), como en el caso de las relaciones 1:1 binarias. Creamos una relación *R* separada cuyos atributos son las claves de *S* y *T*, y cuya clave principal es la misma que la clave de *S*. Esta opción puede utilizarse si pocas tuplas de *S* participan en la relación para evitar excesivos valores NULL en la *foreign key*.

**Paso 5: Mapeado de tipos de relaciones M:N binarias.** Por cada tipo de relación M:N binaria *R*, cree una nueva relación *S* para representar a *R*. Incluya como atributos de la *foreign key* en *S* las claves principales de las relaciones que representan los tipos de entidad participantes; su combinación formará la clave principal de *S*. Incluya también cualesquiera atributos simples del tipo de relación M:N (o los componentes simples de los atributos compuestos) como atributos de *S*. No podemos representar un tipo de relación M:N con un atributo de *foreign key* en una de las relaciones participantes (como hicimos para los tipos de relación 1:1 o 1:N) debido a la razón de cardinalidad M:N; debemos crear una *relación de relación S* separada.

En nuestro ejemplo, mapeamos el tipo de relación M:N TRABAJA_EN de la Figura 7.1 creando la relación TRABAJA_EN de la Figura 7.2. Incluimos las claves principales de las relaciones PROYECTO y EMPLEADO como *foreign keys* en TRABAJA_EN y las renombramos como NumProy y DniEmpleado, respectivamente. También incluimos un atributo Horas en TRABAJA_EN para representar el atributo Horas del tipo de relación. La clave principal de la relación TRABAJA_EN es la combinación de los atributos de la *foreign key* {DniEmpleado, NumProy}. Esta relación de *relación* se muestra en la Figura 7.3(c).

La opción (CASCADA) de propagación para la acción de activación referencial (consulte la Sección 8.2) debe especificarse en las *foreign keys* de la relación correspondiente a la relación *R*, puesto que la existencia de cada instancia de relación depende de cada una de las entidades que relaciona. Esto se puede utilizar tanto para ON UPDATE como para ON DELETE.

Siempre podemos asignar las relaciones 1:1 o 1:N de un modo similar a las relaciones M:N utilizando la metodología de la referencia cruzada (relación de relación), como explicamos anteriormente. Esta alternativa es particularmente útil cuando existen pocas instancias de relación, a fin de evitar valores NULL en las *foreign keys*. En este caso, la clave principal de la relación de relación *sólo será una* de las *foreign keys* que hace referencia a las relaciones de entidad participantes. Para una relación 1:N, la clave principal de la relación de relación será la *foreign key* que hace referencia a la relación de la entidad en el lado N. En una relación 1:1, cada *foreign key* se puede utilizar como la clave principal de la relación de relación, siempre y cuando no haya entradas NULL en la relación.

**Paso 6: Mapeado de atributos multivalor.** Por cada atributo multivalor *A*, cree una nueva relación *R*. Esta relación incluirá un atributo correspondiente a *A*, más el atributo clave principal *K* (como *foreign key* en *R*) de la relación que representa el tipo de entidad o tipo de relación que tiene *A* como un atributo. La clave principal de *R* es la combinación de *A* y *K*. Si el atributo multivalor es compuesto, incluimos sus componentes simples.

En nuestro ejemplo, creamos una relación LOCALIZACIONES_DPTO (véase la Figura 7.3[d]). El atributo UbicaciónDpto representa el atributo multivalor UBICACIONES de DEPARTAMENTO, mientras que NúmeroDpto (como *foreign key*) representa la clave principal de la relación DEPARTAMENTO. La clave principal de LOCALIZACIONES_DPTO es la combinación de {NúmeroDpto, UbicacionDpto}. En LOCALIZACIONES_DPTO existirá una tupla separada para cada ubicación que tenga un departamento.

La opción (CASCADA) de propagación para la acción de activación referencial (consulte la Sección 8.2) debe especificarse en la *foreign key* en la relación *R* correspondiente al atributo multivalor para ON UPDATE y ON DELETE. La clave de *R*, cuando se mapea un atributo compuesto multivalor, requiere cierto análisis del significado de los atributos simples. En algunos casos, cuando un atributo multivalor es compueto, sólo son necesarios algunos de los atributos simples para que formen parte de la clave de *R*; estos atributos son parecidos a una clave parcial de un tipo de entidad débil que corresponda al atributo multivalor (consulte la Sección 3.5).

La Figura 7.2 muestra el esquema de la base de datos relacional EMPRESA obtenido con los pasos 1 a 6, y la Figura 5.6 muestra un ejemplo del estado de la base de datos. Todavía no hemos explicado el mapeado de los tipos de relación *n*-ary ($n > 2$) porque en la Figura 7.1 no existe ninguno; se asignan de una forma parecida a los tipos de relación M:N añadiendo el siguiente paso (adicional) al algoritmo de asignación.

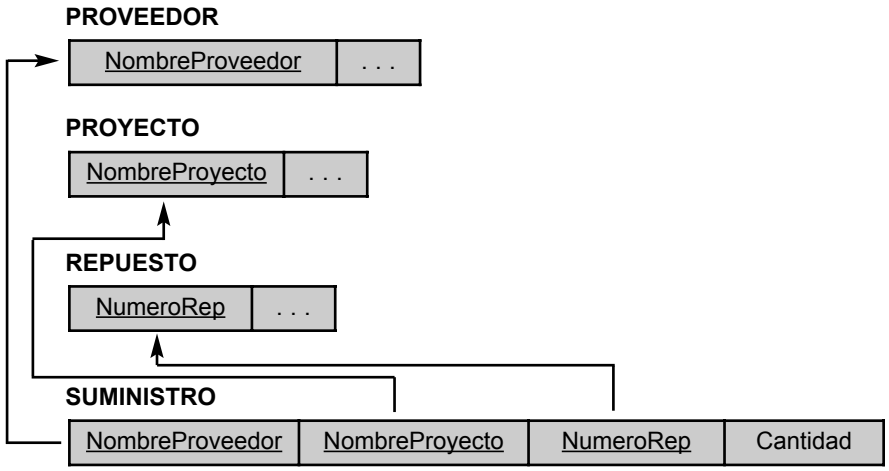
**Paso 7: Mapeado de los tipos de relación *n*-ary.** Por cada tipo de relación *n*-ary *R*, donde $n > 2$, cree una nueva relación *S* para representar *R*. Incluya como atributos de la *foreign key* en *S* las claves principales de las relaciones que representan los tipos de entidad participantes. Incluya también cualesquiera atributos simples del tipo de relación *n*-ary (o los componentes simples de los atributos compuestos) como atributos de *S*. Normalmente, la clave principal de *S* es una combinación de todas las *foreign keys* que hacen referencia a las relaciones que representan los tipos de entidad participantes. No obstante, si las restricciones de cardinalidad en cualquiera de los tipos de entidad *E* que participan en *R* es 1, entonces la clave principal de *S* no debe incluir el atributo de la *foreign key* que hace referencia a la relación *E*' correspondiente a *E* (consulte la Sección 3.9.2).

Por ejemplo, considere el tipo de relación SUMINISTRO de la Figura 3.17: se puede asignar a la relación SUMINISTRO de la Figura 7.4, cuya clave principal es la combinación de las tres *foreign keys* {NombreProveedor, NumeroRep, NombreProyecto}.

## 7.1.2 Explicación y resumen del mapeado para las construcciones del modelo ER

La Tabla 7.1 resume las correspondencias entre las construcciones y las restricciones de los modelos ER y relacional.

**Figura 7.4.** Asignación del tipo de relación *n*-ary SUMINISTRO de la Figura 3.17(a).




Uno de los puntos principales del esquema relacional, en contraste con un esquema ER, es que los tipos de relación no están explícitamente representados; en su lugar, están representados con dos atributos *A* y *B*, uno como clave principal y otro como *foreign key* (en el mismo dominio) incluidos en las dos relaciones *S* y *T*. Dos tuplas de *S* y *T* están relacionadas cuando tienen el mismo valor para *A* y *B*. Al utilizar la operación EQUIJOIN (o concatenación natural si los dos atributos de concatenación tienen el mismo nombre) sobre *S.A* y *T.B*, podemos combinar todas las parejas de tuplas relacionadas de *S* y *T* y materializar la relación. Cuando se ve implicado un tipo de relación 1:1 o 1:N binaria, normalmente sólo se necesita una operación de concatenación. En el caso de un tipo de relación M:N binaria, se necesitan dos operaciones de concatenación, mientras que para los tipos de relación *n*-ary, se necesitan *n* concatenaciones para materializar completamente las instancias de relación.

Por ejemplo, para formar una relación que incluya el nombre del empleado, el nombre del proyecto y las horas que el empleado trabaja en cada proyecto, tenemos que conectar cada tupla EMPLEADO con las tuplas PROYECTO relacionadas a través de la relación TRABAJA_EN de la Figura 7.2. Por tanto, debemos aplicar la operación EQUIJOIN a las relaciones EMPLEADO y TRABAJA_EN con la condición de concatenación Dni

**Tabla 7.1.** Correspondencia entre los modelos ER y relacional.

| Modelo ER | Modelo relacional |
|---|---|
| Tipo de entidad | Relación de *entidad* |
| Tipo de relación 1:1 o 1:N | *Foreign key* (o relación de *relación*) |
| Tipo de relación M:N | Relación de *relación* y *dos foreign keys* |
| Tipo de relación *n*-ary | Relación de *relación* y *n foreign keys* |
| Atributo simple | Atributo |
| Atributo compuesto | Conjunto de atributos simples |
| Atributo multivalor | Relación y *foreign key* |
| Conjunto de valores | Dominio |
| Atributo clave | Clave principal (o secundaria) |

= DniEmpleado, y después aplicar otra operación EQUIJOIN a la relación resultante y a la relación PROYECTO con la condición de concatenación NumProy = NumProyecto. En general, cuando hay que atravesar varias relaciones, hay que especificar numerosas operaciones de concatenación. El usuario de una base de datos relacional siempre debe tener cuidado con los atributos de la *foreign key* a fin de utilizarlos correctamente al combinar las tuplas relacionadas de dos o más relaciones. En ocasiones, esto se considera un inconveniente del modelo de datos relacional, porque las correspondencias *foreign key*/clave principal no siempre resultan obvias al inspeccionar los esquemas relacionales. Si se lleva a cabo una EQUIJOIN entre los atributos de dos relaciones que no representan una relación *foreign key*/clave principal, el resultado a menudo puede carecer de sentido y conducir a datos falsos. Por ejemplo, el lector puede intentar concatenar las relaciones PROYECTO y LOCALIZACIONES_DPTO con la condición UbicacionDpto = UbicacionProyecto y examinar el resultado (consulte el Capítulo 10).

En el esquema relacional creamos una relación separada por *cada* atributo multivalor. Para una entidad en particular con un conjunto de valores para el atributo multivalor, el valor del atributo clave de la entidad se repite una vez por cada valor del atributo multivalor en una tupla separada, porque el modelo relacional básico *no permite* valores múltiples (una lista, o un conjunto de valores) para un atributo en una sola tupla. Por ejemplo, como el departamento 5 tiene tres ubicaciones, hay tres tuplas en la relación LOCALIZACIONES_DPTO de la Figura 5.6; cada tupla especifica una de las ubicaciones. En nuestro ejemplo, aplicamos EQUIJOIN a LOCALIZACIONES_DPTO y DEPARTAMENTO en el atributo NúmeroDpto para obtener los valores de todas las ubicaciones junto con otros atributos DEPARTAMENTO. En la relación resultante, los valores de los otros atributos DEPARTAMENTO están repetidos en tuplas separadas por cada una de las ubicaciones que tiene un departamento.

El álgebra relacional básica no tiene una operación ANIDAR (NEST) o COMPRIMIR (COMPRESS) que pudiera producir un conjunto de tuplas de la forma {<‘1’, ‘Madrid’>, <‘4’, ‘Gijón’>, <‘5’, {‘Valencia’, ‘Sevilla’, ‘Madrid’}>} a partir de la relación LOCALIZACIONES_DPTO de la Figura 5.6. Esto es un serio inconveniente de la versión normalizada o *plana* del modelo relacional. En este sentido, el modelo orientado a objetos y los modelos de herencia jerárquica y de red ofrecen más posibilidades que el modelo relacional. El modelo relacional anidado y los sistemas de objetos relacionales (consulte el Capítulo 22) intentan remediar esto.

# 7.2 Mapeado de construcciones del modelo EER a las relaciones

A continuación explicamos el mapeado de las construcciones del modelo EER a las relaciones, extendiendo el algoritmo de asignación ER-a-relacional que presentamos en la Sección 7.1.1.

## 7.2.1 Mapeado de la especialización o generalización

Ha varias opciones para mapear una cierta cantidad de subclases que juntas forman una especialización (o, alternativamente, que están generalizadas en una subclase), como las subclases {SECRETARIA, TÉCNICO, INGENIERO} de EMPLEADO de la Figura 4.4. Podemos añadir un paso más a nuestro algoritmo de mapeado ER-a-relacional de la Sección 7.1.1, que tiene siete pasos, para manipular el mapeado de la especialización. El paso 8, que se detalla a continuación, ofrece las opciones más comunes; también son posibles otros mapeados. Explicamos las condiciones bajo las que debe utilizarse cada opción. Utilizamos Atrs($R$) para denotar *los atributos de la relación R*, y CPr(R) para referirnos a la *clave principal de R*. En primer lugar, describimos formalmente el mapeado, y después lo ilustramos con unos ejemplos.

**Paso 8: Opciones para mapear la especialización o generalización.** Convierta cada especialización con $m$ subclases $\{S_1, S_2, \ldots, S_m\}$ y la superclase (generalizada) $C$, donde los atributos de $C$ son $\{k, a_1, \ldots a_n\}$ y $k$ es la clave (principal), en esquemas de relación utilizando alguna de las siguientes opciones:

- **Opción 8A: Varias relaciones (superclase y subclase).** Cree una relación $L$ para $C$ con los atributos $\text{Atrs}(L) = \{k, a_1, \ldots, a_n\}$ y la $\text{CPr}(L) = k$. Cree una relación $L_i$ para cada subclase $S_i$, $1 \leq i \leq m$, con los atributos $\text{Atrs}(L_i) = \{k\} \cup \{\text{atributos de } S_i\}$ y $\text{CPr}(L_i) = k$. Esta opción funciona para cualquier especialización (total o parcial, disjunta o solapada).
- **Opción 8B: Varias relaciones (sólo relaciones de subclase).** Cree una relación $L_i$ por cada subclase $S_i$, $1 \leq i \leq m$, con los atributos $\text{Atrs}(L_i) = \{\text{atributos de } S_i\} \cup \{k, a_1, \ldots, a_n\}$ y la $\text{CPr}(L_i) = k$. Esta opción sólo funciona para una especialización cuyas subclases sean *totales* (cada entidad de la superclase debe pertenecer [al menos] a una de las subclases). Si la especialización es *solapada*, una entidad se puede duplicar en varias relaciones.
- **Opción 8C: Una sola relación con un atributo de tipo.** Cree una sola relación $L$ con los atributos $\text{Atrs}(L) = \{k, a_1, \ldots, a_n\} \cup \{\text{atributos de } S_1\} \cup \ldots \cup \{\text{atributos de } S_m\} \cup \{t\}$ y $\text{CPr}(L) = k$. El atributo $t$ se denomina atributo de **tipo** (o **discriminatorio**) que indica la subclase a la que pertenece cada tupla, si la hay. Esta opción sólo funciona para una especialización cuyas subclases son *disjuntas,* y tiene el potencial de generar muchos valores NULL si en las subclases existen muchos atributos específicos.
- **Opción 8D: Una sola relación con varios atributos de tipo.** Cree un solo esquema de relación $L$ con los atributos $\text{Atrs}(L) = \{k, a_1, \ldots, a_n\} \cup \{\text{atributos de } S_1\} \cup \ldots \cup \{\text{atributos de } S_m\} \cup \{t_1, t_2, \ldots, t_m\}$ y la $\text{CPr}(L) = k$. Cada $t_i$, $1 \leq i \leq m$, es un **atributo de tipo booleano** que indica si una tupla pertenece a la subclase $S_i$. Esta opción funciona para la especialización cuyas subclases sean *solapadas* (pero también funcionará para una especialización disjunta).

Las opciones 8A y 8B se pueden denominar **opciones de relación múltiple**, mientras que las opciones 8C y 8D se pueden denominar **opciones de una sola relación**. La opción 8A crea una relación $L$ para la superclase $C$ y sus atributos, más una relación $L_i$ por cada subclase $S_i$; cada $L_i$ incluye los atributos específicos (o locales) de $S_i$, más la clave principal de la superclase $C$, que se propaga a $L_i$ y se convierte en su clave principal. También se convierte en una *foreign key* a la relación de la superclase. Una operación EQUIJOIN en la clave principal entre cualquier $L_i$ y $L$ produce todos los atributos específicos y heredados de las entidades de $S_i$.

Esta opción se ilustra en la Figura 7.5(a) para el esquema EER de la Figura 4.4. La opción 8A funciona para cualesquiera restricciones de la especialización: disjunta o solapada, total o parcial. Observe que la restricción

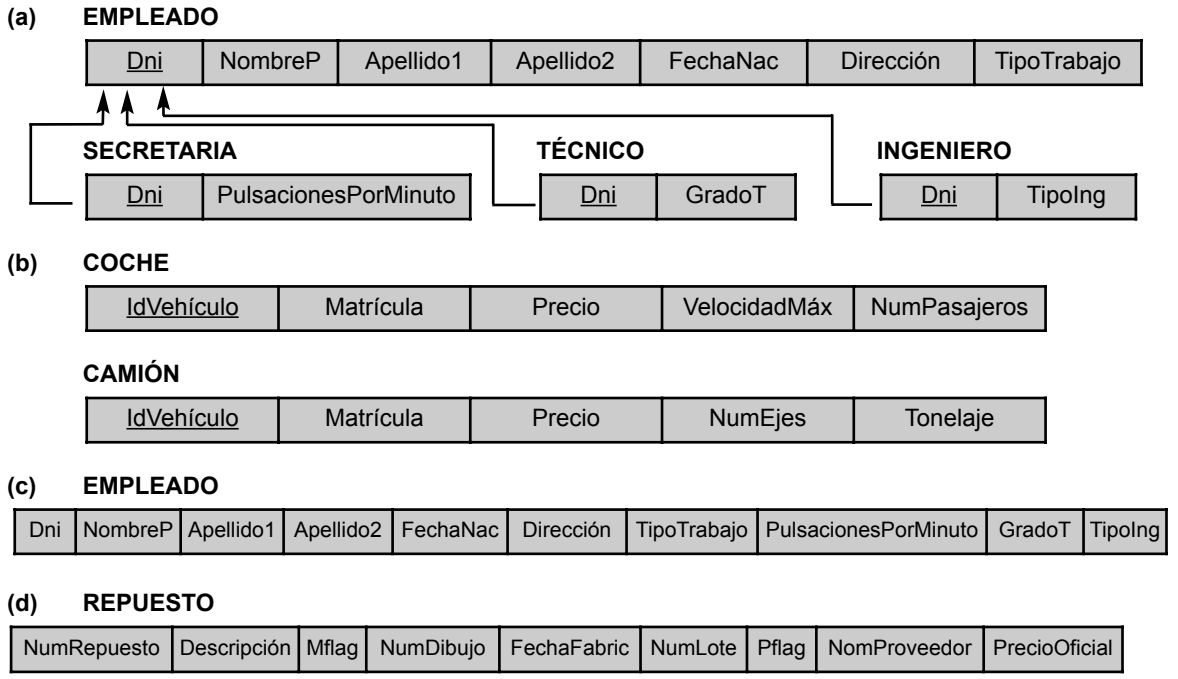
$$\pi_{<k>}(L_i) \subseteq \pi_{<k>}(L)$$

se debe mantener por cada $L_i$. Esto especifica una *foreign key* de cada $L_i$ a $L$, así como una *dependencia de inclusión* $L_i.k < L.k$ (consulte la Sección 11.5).

En la opción 8B, la operación EQUIJOIN se *crea sobre* el esquema y se elimina la relación $L$, como se ilustra en la Figura 7.5(b) para la especialización EER de la Figura 4.3(b). Esta opción sólo funciona bien cuando se mantienen las dos restricciones (disjunta y total). Si la especialización no es total, una entidad que no pertenece a cualquiera de las subclases $S_i$ se pierde. Si la especialización no es disjunta, una entidad que pertenece a más de una subclase tendrá sus atributos heredados de la superclase $C$ almacenados redundantemente en más de una $L_i$. Con la opción 8B, ninguna relación mantiene todas las entidades de la superclase $C$; en consecuencia, debemos aplicar una operación de concatenación externa (OUTER UNION, o FULL OUTER JOIN) a las relaciones $L_i$ para recuperar todas las entidades de $C$. El resultado de la unión externa será parecido a las relaciones de las opciones 8C y 8D, salvo que desaparecerán los campos de tipo. Siempre que busquemos una entidad arbitraria en $C$, tenemos que buscar todas las $m$ relaciones $L_i$.

Las opciones 8C y 8D crean una sola relación para representar la superclase $C$ y todas sus subclases. Una entidad que no pertenece a alguna de las subclases tendrá valores NULL para los atributos específicos de esas subclases. Estas opciones no son recomendables si se definen muchos atributos específicos para las subclases. No obstante, si sólo existen unos pocos atributos de subclase, estos mapeados son preferibles a las opciones 8A

**Figura 7.5.** Opciones para el mapeado de la especialización y la generalización. (a) Mapeado del esquema EER de la Figura 4.4 utilizando la opción 8A. (b) Mapeado del esquema EER de la Figura 4.3(b) utilizando la opción 8B. (c) Mapeado del esquema EER de la Figura 4.4 utilizando la opción 8C. (d) Mapeado de la Figura 4.5 utilizando la opción 8D con los campos de tipo booleano Mflag y Pflag.




y 8B, porque eliminan la necesidad de especificar operaciones EQUIJOIN y OUTER UNION; por consiguiente, pueden ofrecer una implementación más eficaz.

La opción 8C se utiliza para manipular las subclases disjuntas, incluyendo un solo **atributo** (o **imagen** o **discriminatorio**) **de tipo** *t* para indicar la subclase a la que pertenece cada tupla; por tanto, el dominio de *t* podría ser {1, 2, . . . , *m*}. Si la especialización es parcial, *t* puede tener valores NULL en las tuplas que no pertenecen a cualquier subclase. Si la especialización está definida por atributo, sirve con el propósito de *t* y ya no se necesita *t*; en la Figura 7.5(c) se ilustra esta opción para la especialización EER de la Figura 4.4.

La opción 8D está diseñada para manipular el solapamiento de subclases incluyendo *m* campos de tipo *booleano*, uno por cada subclase. También se puede utilizar para las subclases disjuntas. Cada campo de tipo $t_i$ puede tener un dominio {sí, no}, donde un valor de "sí" indica que la tupla es miembro de la subclase $S_i$. Si utilizamos esta opción para la especialización EER de la Figura 4.4, incluiríamos tres atributos de tipo (EsSecretaria, EsIngeniero y EsTécnico) en lugar del atributo TipoTrabajo de la Figura 7.5(c). También es posible crear un solo atributo de tipo de *m bits* en lugar de *m* campos de tipo.

Cuando tenemos una jerarquía de especialización (o generalización) multinivel o entramado, no tenemos que seguir la misma opción de mapeado para todas las especializaciones. En su lugar, podemos utilizar una opción de mapeado para parte de la jerarquía o entramado y otras opciones para otras partes. La Figura 7.6 muestra un posible mapeado en las relaciones para el entramado EER de la Figura 4.6. Aquí hemos utilizado la opción 8A para PERSONA/{EMPLEADO, EX_ALUMNO, ESTUDIANTE}, la opción 8C para EMPLEADO/{ADMINISTRATIVO, DOCENTE, ADJUNTO}, y la opción 8D para ADJUNTO/{ADJUNTO_INVESTIGACIÓN, ADJUNTO_ENSEÑANZA}, ESTUDIANTE/ADJUNTO (en ESTUDIANTE), y ESTUDIANTE/{ESTUDIANTE_DIPLOMADO, ESTUDIANTE_NO_DIPLOMADO}. En la Figura 7.6, todos los atributos cuyos nombres terminan con *tipo* o *flag* son campos de tipo.

**Figura 7.6.** Mapeado del entramado de especialización EER de la Figura 4.8 utilizando varias opciones.

## 7.2.2 Mapeado de subclases compartidas (herencia múltiple)

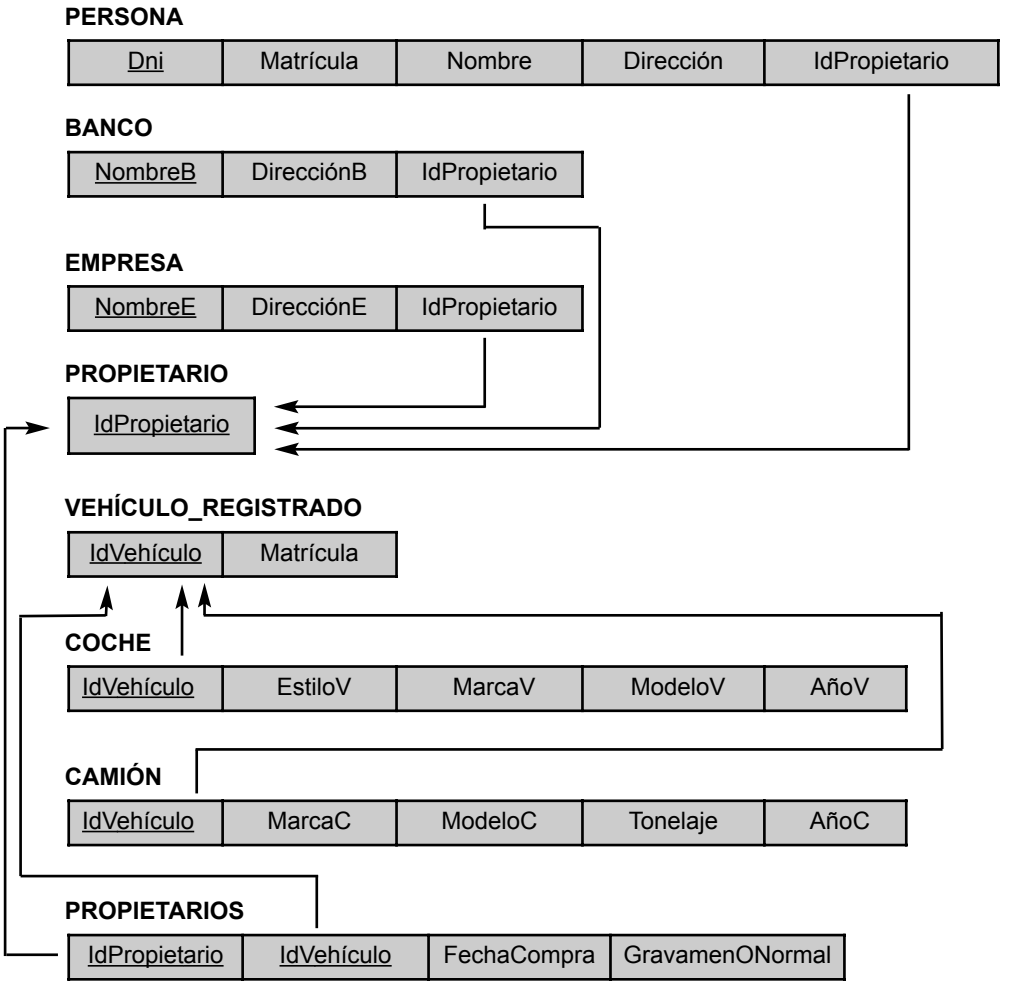
Una subclase compartida, como INGENIERO_JEFE de la Figura 4.7, es una subclase de varias superclases, indicando la herencia múltiple. Estas clases deben tener todas el mismo atributo clave; en caso contrario, la subclase compartida se modelaría como una categoría. Podemos aplicar cualquiera de las opciones explicadas en el paso 8 para una subclase compartida, sujeta a las restricciones explicadas en el paso 8 del algoritmo de mapeado. En la Figura 7.6 se utilizan las opciones 8C y 8D para la subclase compartida ADJUNTO. La opción 8C se utiliza en la relación EMPLEADO (atributo TipoEmpleado) y la opción 8D para la relación ESTUDIANTE (atributo Adjunto_flag).

## 7.2.3 Mapeado de categorías (tipos de unión)

Añadimos otros paso al procedimiento de mapeado (paso 9) para manipular las categorías. Una categoría (o tipo de unión) es una subclase de la *unión* de dos o más superclases que puede tener diferentes claves, porque pueden ser de distintos tipos de entidad. Un ejemplo es la categoría PROPIETARIO de la Figura 4.8, que es un subconjunto de la unión de tres tipos de entidad PERSONA, BANCO y EMPRESA. La otra categoría de esta figura, VEHÍCULO_REGISTRADO, tiene dos superclases que tienen el mismo atributo clave.

**Paso 9: Mapeado de tipos de unión (categorías).** Para mapear una categoría cuyas superclases definitorias tienen claves diferentes, es costumbre especificar un nuevo atributo de clave, denominado **clave sustituta**, al crear una relación correspondiente a la categoría. Las claves de las clases de definición son diferentes, por lo que no podemos utilizar una de ellas exclusivamente para identificar todas las entidades de la categoría. En nuestro ejemplo de la Figura 4.8, podemos crear una relación PROPIETARIO para corresponderse con la categoría PROPIETARIO, como se ilustra en la Figura 7.7, e incluir cualquier atributo de la categoría en esta relación. La clave principal de la relación PROPIETARIO es la clave sustituta, que denominamos IdPropietario. También incluimos el atributo de clave sustituta IdPropietario como *foreign key* en cada relación correspondiente a una superclase de la categoría, para especificar la correspondencia en los valores entre la clave sustituta y la clave de cada superclase. Observe que si una entidad PERSONA (o BANCO o EMPRESA) en particular no es miembro de PROPIETARIO, tendría un valor NULL para su atributo IdPropietario en su correspondiente tupla de la relación PERSONA (o BANCO o EMPRESA), y no tendría una tupla en la relación PROPIETARIO.

**Figura 7.7.** Mapeado de las categorías EER (tipos de unión) de la Figura 4.8 en relaciones.




En el caso de una categoría cuyas superclases tengan la misma clave, como VEHÍCULO en la Figura 4.8, no es necesaria una clave sustituta. El mapeado de la categoría VEHÍCULO_REGISTRADO, que ilustra este caso, también se muestra en la Figura 7.7.

# 7.3 Resumen

En la Sección 7.1 vimos cómo el diseño de un esquema conceptual en el modelo ER se puede mapear en un esquema de base de datos relacional. Hemos ofrecido e ilustrado con ejemplos de la base de datos EMPRESA un algoritmo para el mapeado ER-a-relacional. La Tabla 7.1 resume las correspondencias entre los modelos ER y relacional. A continuación, en la Sección 7.2 añadimos algunos pasos al algoritmo para mapear las estructuras del modelo ER al modelo relacional. En las herramientas gráficas de diseño de bases de datos se incorporan algoritmos parecidos para crear automáticamente un esquema relacional a partir del diseño de un esquema conceptual.

## Preguntas de repaso

**7.1.** Explique las correspondencias entre los modelos ER y relacional. Muestre cómo cada construcción del modelo ER se puede mapear al modelo relacional y explique los mapeados alternativos.

**7.2.** Explique las opciones que hay para mapear las construcciones del modelo EER en relaciones.

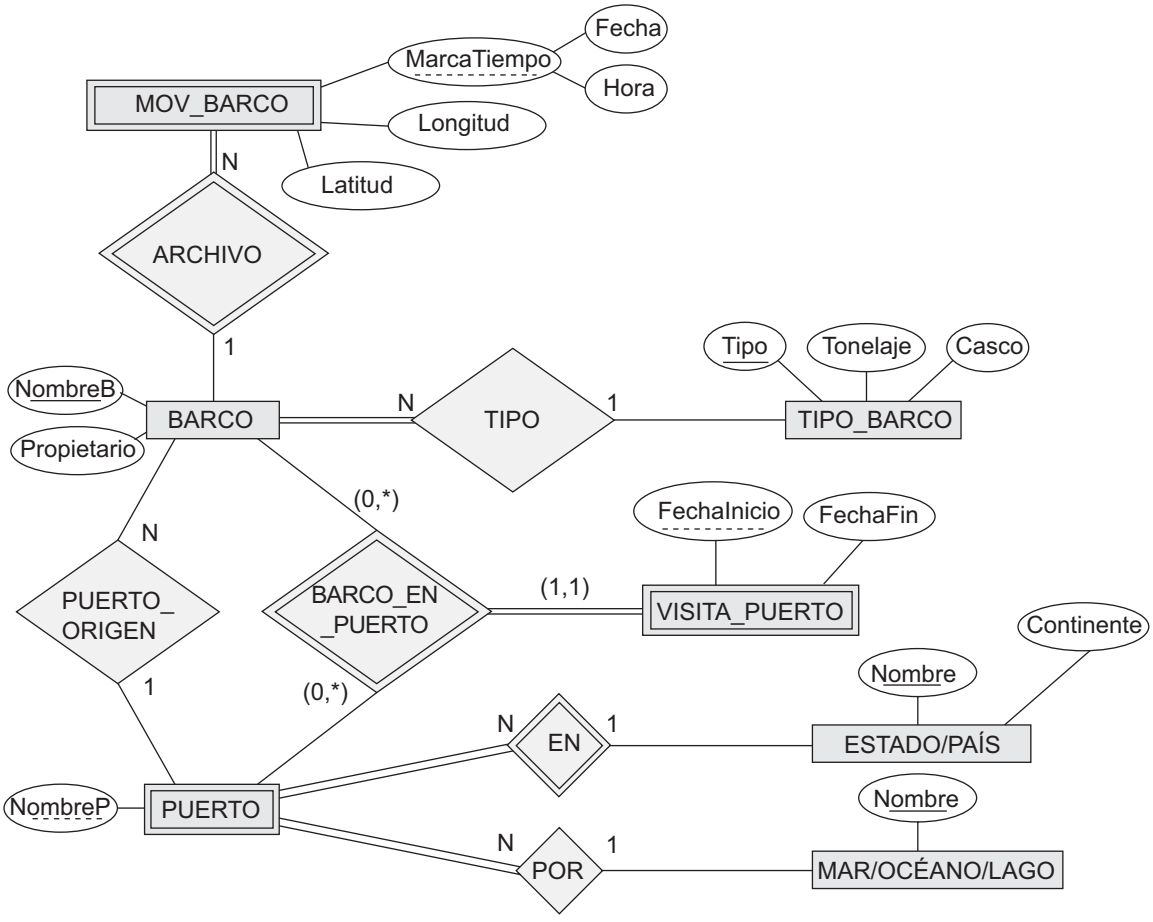
## Ejercicios

**7.3.** Intente mapear el esquema relacional de la Figura 6.14 a un esquema ER. Esto es parte de un proceso conocido como *ingeniería inversa*, donde se crea el esquema conceptual para una base de datos implementada ya existente. Detalle las suposiciones que haga.

**7.4.** La Figura 7.8 muestra un esquema ER para una base de datos que se puede utilizar para que las autoridades marítimas puedan hacer el seguimiento de los buques mercantes y sus ubicaciones. Mapee este esquema a un esquema relacional y especifique todas las claves principales y *foreign keys*.

**7.5.** Mapee el esquema ER de BANCO del Ejercicio 3.23 (y mostrado en la Figura 3.21) a un esquema relacional. Especifique todas las claves principales y *foreign keys*. Repítalo para el esquema de AEROLÍNEA (véase la Figura 3.20) del Ejercicio 3.19 y para los otros esquemas de los Ejercicios 3.16 a 3.24.

**Figura 7.8.** Un esquema ER para una base de datos SEGUIMIENTO_BARCOS.



**7.6.** Mapee los diagramas EER de las Figuras 4.9 y 4.12 a esquemas relacionales. Justifique sus elecciones.

**7.7.** ¿Es posible mapear satisfactoriamente un tipo de relación M:N sin necesidad de una nueva relación? ¿Por qué, o por qué no?

**7.8.** Utilizando los atributos que proporcionó para el diagrama EER en el Ejercicio 4.27 del Capítulo 4, mapee el esquema completo a un conjunto de relaciones. Elija una de las opciones (8A a 8D) de la Sección 7.2.1 para realizar el mapeado de las generalizaciones y defienda su elección.

## Ejercicios de práctica

**7.9.** Considere el diseño ER para la base de datos UNIVERSIDAD que se modeló utilizando una herramienta como ERWin o Rational Rose en el Ejercicio de práctica 3.31. Con la función de generación del esquema SQL de la herramienta de modelado, genere el esquema SQL para una base de datos Oracle.

**7.10.** Considere el diseño ER para la base de datos PEDIDOS_CORREO que se modeló utilizando una herramienta como ERWin o Rational Rose en el Ejercicio de práctica 3.32. Con la función de generación del esquema SQL de la herramienta de modelado, genere el esquema SQL para una base de datos Oracle.

**7.11.** Considere el diseño ER para la base de datos AEROLÍNEA que se modeló utilizando una herramienta como ERWin o Rational Rose en el Ejercicio de práctica 3.34. Con la función de generación del esquema SQL de la herramienta de modelado, genere el esquema SQL para una base de datos Oracle.

**7.12.** Considere el diseño EER para la base de datos UNIVERSIDAD que se modeló utilizando una herramienta como ERWin o Rational Rose en el Ejercicio de práctica 4.28. Con la función de generación del esquema SQL de la herramienta de modelado, genere el esquema SQL para una base de datos Oracle.

**7.13.** Considere el diseño EER para la base de datos PEQUEÑO_AEROPUERTO que se modeló utilizando una herramienta como ERWin o Rational Rose en el Ejercicio de práctica 4.29. Con la función de generación del esquema SQL de la herramienta de modelado, genere el esquema SQL para una base de datos Oracle.

## Bibliografía seleccionada

El algoritmo de mapeado ER-a-relacional se describió en el ensayo clásico de Chen (Chen 1976) que presentó el modelo ER original. Batini y otros (1992) explica varios algoritmos de mapeado de los modelos ER y EER a modelos heredados, y viceversa.

CAPÍTULO 8

# SQL-99: definición del esquema, restricciones, consultas y vistas

El lenguaje SQL se puede considerar como una de las principales razones del éxito comercial de las bases de datos relacionales. Como se convirtió en un estándar para estas últimas, los usuarios perdieron el miedo a migrar sus aplicaciones de base de datos desde otros tipos de sistemas de bases de datos (por ejemplo, sistemas de red o jerárquicos) a los sistemas relacionales, porque aunque estuvieran satisfechos con el producto DBMS relacional que estaban utilizando, no esperaban que la conversión a otro producto DBMS relacional fuera caro y consumiera mucho tiempo, ya que ambos sistemas seguían los mismos estándares en cuanto al lenguaje. En la práctica, por supuesto, hay muchas diferencias entre los distintos paquetes DBMS relacionales comerciales. Sin embargo, si el usuario sólo utiliza las funciones que forman parte del estándar, y si ambos sistemas relacionales soportan fielmente el estándar, la conversión entre los dos sistemas es mucho más sencilla. Otra ventaja de disponer de un estándar es que los usuarios pueden escribir sentencias en una aplicación de base de datos para acceder a los datos almacenados en dos o más DBMSs relacionales sin tener que cambiar el sublenguaje de base de datos (SQL), siempre y cuando esos DBMS soporten el SQL estándar.

Este capítulo presenta las principales características del estándar SQL para los DBMSs relacionales *comerciales*, mientras que el Capítulo 5 presentó los conceptos fundamentales del modelo de datos relacional *formal*. En el Capítulo 6 (Secciones 6.1 a 6.5) explicamos las operaciones del *álgebra relacional*, que es muy importante para entender los tipos de solicitudes que se pueden especificar en una base de datos relacional. También son muy importantes para el procesamiento y la optimización de consultas en un DBMS relacional, como también veremos en los Capítulos 15 y 16. No obstante, las operaciones del álgebra relacional están consideradas como muy técnicas por la mayoría de los usuarios de los DBMSs comerciales, porque una consulta en el álgebra relacional se escribe como una secuencia de operaciones que, cuando se ejecutan, producen el resultado requerido. Por tanto, el usuario debe especificar cómo (es decir, *en qué orden*) hay que ejecutar las operaciones de la consulta. Por otro lado, el lenguaje SQL proporciona una interfaz de lenguaje *declarativo* del más alto nivel, por lo que el usuario sólo especifica *lo que debe ser* el resultado, dejando para el DBMS la optimización y las decisiones de cómo ejecutar la consulta. Aunque SQL incluye algunas características del álgebra relacional, está basado en gran medida en el *cálculo relacional de tuplas* (consulte la Sección 6.6). Sin embargo, la sintaxis de SQL es mucho más amigable para el usuario que cualquiera de los otros dos lenguajes formales.

El nombre **SQL** significa Lenguaje de consulta estructurado (*Structured Query Language*). Originalmente, SQL se denominaba SEQUEL (*Structured English QUEry Language*) y fue diseñado e implementado por IBM Research a modo de interfaz para un sistema de base de datos relacional conocido como SYSTEM R. SQL es ahora el lenguaje estándar de los DBMSs relacionales comerciales. Un esfuerzo conjunto llevado a cabo por el Instituto nacional americano de normalización (ANSI, *American National Standards Institute*) y la Organizacón internacional para la normalización (ISO, *International Standards Organization*) llevó a una versión estándar de SQL (ANSI 1986), denominada SQL-86 o SQL1. A continuación se desarrolló un estándar revisado y mucho más amplio, SQL2 (también conocido como SQL-92). El siguiente estándar fue SQL-99. Se han propuesto otros estándares, como SQL3, pero no han gozado de suficiente respaldo por parte de la industria. Intentaremos cubrir en lo posible la última versión de SQL.

SQL es un lenguaje de bases de datos global: cuenta con sentencias para definir datos, consultas y actualizaciones. Por tanto, se comporta como DDL y como DML. Además, dispone de características para definir vistas en la base de datos, especificar temas de seguridad y autorización, definir restricciones de integridad, y especificar controles de transacciones. También tiene reglas para incrustar sentencias de SQL en un lenguaje de programación de propósito general, como Java, COBOL o C/C++.[1]

Como la especificación del estándar SQL sigue creciendo, con más funciones en cada nueva versión del estándar, el último estándar, **SQL-99**, está dividido en una especificación **central** (o núcleo) más unos **paquetes** especilizados opcionales. Se supone que todos los desarrolladores de DBMSs compatibles con SQL-99 implementan dicho núcleo. Los paquetes pueden implementarse como módulos opcionales que pueden adquirirse independientemente para determinadas aplicaciones de bases de datos; por ejemplo, para el minado de datos, datos espaciales, datos meteorólogicos, datos de almacenamiento, procesamiento analítico online (OLAP), datos multimedia, etcétera. Ofreceremos un resumen de algunos de estos paquetes (y dónde se explican en este libro) al final del presente capítulo.

Como SQL es muy importante (y muy extenso) dedicaremos dos capítulos a sus características básicas. En este capítulo, la Sección 8.1 describe los comandos DDL de SQL para crear esquemas y tablas, y ofrece una panorámica de los tipos básicos de datos. La Sección 8.2 explica cómo se especifican las restricciones básicas, como la integridas de clave y referencial. La Sección 8.3 explica las sentencias para modificar esquemas, tablas y restricciones. La Sección 8.4 describe las construcciones SQL básicas destinadas a especificar las consultas de recuperación, mientras que la Sección 8.5 explora las funciones más complejas de las consultas SQL, como las funciones de agregación y agrupamiento. La Sección 8.6 describe los comandos SQL para la inserción, eliminación y actualización de datos.

En la Sección 8.7 describimos la sentencia CREATE ASSERTION, que permite especificar las restricciones más generales de la base de datos. También introducimos el concepto de *triggers*, que se presentan más en profundidad en el Capítulo 24. A continuación, la Sección 8.8 describe los servicios de SQL para definir vistas en la base de datos. Las vistas también se denominan *tablas virtuales o derivadas* porque presentan al usuario lo que parece haber en unas tablas; sin embargo, la información de dichas tablas deriva de otras tablas previamente definidas.

La Sección 8.9 enumera algunas de las características de SQL que se presentan en otros capítulos del libro, como el control de las transacciones en el Capítulo 17, los temas de seguridad y autorización en el Capítulo 23, las bases de datos activas (*triggers*) en el Capítulo 24, las características orientadas a objetos en el Capítulo 22, y las características de procesamiento analítico online (OLAP) en el Capítulo 28. La Sección 8.10 resume el capítulo. El Capítulo 9 explica varias técnicas de programación de bases de datos destinadas a programar con SQL.

El lector que desee una introducción menos global a SQL, puede omitir partes de las Secciones 8.2 y 8.5, así como las Secciones 8.7 y 8.8.

---

[1] Originalmente, SQL tenía sentencias para crear y eliminar índices en los ficheros que representaban las relaciones, pero esto ha desaparecido por un tiempo del estándar SQL.

# 8.1 Definición de datos y tipos de datos de SQL

SQL utiliza los términos **tabla**, **fila** y **columna** para los términos *relación, tupla* y *atributo* del modelo relacional formal, respectivamente. Utilizaremos todos estos términos indistintamente. El principal comando de SQL para definir datos es la sentencia CREATE, que se utiliza para crear esquemas, tablas (relaciones) y dominios (así como otras estructuras, como vistas, aserciones y *triggers*). Antes de describir las sentencias CREATE relevantes, explicamos los conceptos de esquema y catálogo en la Sección 8.1.1 para centrar nuestra explicación. La Sección 8.1.2 describe la creación de las tablas y la Sección 8.1.3 describe los tipos de datos más importantes disponibles para la especificación de atributos. Como la especificación SQL es muy larga, ofrecemos una descripción de las características más importantes. Los detalles más avanzados se pueden encontrar en los distintos documentos de los estándares SQL (consulte las notas bibliográficas).

## 8.1.1 Conceptos de esquema y catálogo en SQL

Las versiones anteriores de SQL no incluían el concepto de esquema de base de datos relacional; todas las tablas (relaciones) estaban consideradas como parte del mismo esquema. El concepto de esquema SQL se incorporó por primera vez en SQL2 para agrupar las tablas y otras estructuras pertenecientes a la misma aplicación de base de datos. Un **esquema SQL** se identifica con un **nombre de esquema** e incluye un **identificador de autorización** para indicar el usuario o la cuenta propietaria del esquema, así como unos **descriptores** para *cada elemento*. Los **elementos** del esquema son las tablas, las restricciones, las vistas, los dominios y otras estructuras (como la concesión de autorización), que describen el esquema. El esquema se crea con la sentencia CREATE SCHEMA, que puede incluir las definiciones de todos sus elementos. Como alternativa, puede asignarse un nombre y un identificador de autorización al esquema, y definir los elementos más tarde. Por ejemplo, la siguiente sentencia crea el esquema EMPRESA, propiedad del usuario cuyo identificador de autorización es 'Jperez'.

```
CREATE SCHEMA EMPRESA AUTHORIZATION Jperez;
```

En general, no todos los usuarios están autorizados a crear esquemas y elementos de esquema. El privilegio de crear esquemas, tablas y otras estructuras debe ser otorgado explícitamente por el administradr del sistema o DBA a las cuentas de usuario pertinentes.

Además del concepto de esquema, SQL2 utiliza el concepto de **catálogo**, que es una colección de esquemas bajo un nombre, en un entorno SQL. Un **entorno** SQL es básicamente una instalación de un RDBMS compatible con SQL en un computador.[2] Un catálogo siempre contiene un esquema especial denominado INFORMATION_SCHEMA, que proporciona información sobre todos los esquemas del catálogo y todos los descriptores de elemento de esos esquemas. Las restricciones de integridad, como la integridad referencial, se pueden definir entre las relaciones sólo si existen en los esquemas del mismo catálogo. Los esquemas del mismo catálogo también pueden compartir ciertos elementos, como las definiciones de dominio.

## 8.1.2 El comando CREATE TABLE de SQL

El comando **CREATE TABLE** se utiliza para especificar una nueva relación, asignándole a esta última un nombre y sus atributos y restricciones iniciales. Primero se especifican los atributos, a cada uno de los cuales se le asigna un nombre, un tipo de datos para especificar su dominio de valores, y cualesquiera restricciones de atributo, como NOT NULL. Las restricciones de clave, integridad de entidad e integridad referencial, pueden especificarse con la sentencia CREATE TABLE después de haber declarado los atributos, o pueden añadirse más tarde con el comando ALTER TABLE (consulte la Sección 8.3). La Figura 8.1 muestra unos ejemplos de sentencias de SQL para definir los datos del esquema de la base de datos relacional de la Figura 5.7.

[2] SQL también incluye el concepto de *grupo* (*cluster*) de catálogos dentro de un entorno, pero no está claro si se requieren muchos niveles de anidamiento en la mayoría de las aplicaciones.

**Figura 8.1.** Sentencias de definición de datos CREATE TABLE para definir el esquema EMPRESA de la Figura 5.7.

```
CREATE TABLE EMPLEADO
        ( Nombre                VARCHAR(15)     NOT NULL,
          Apellido1             CHAR,
          Apellido2             VARCHAR(15)     NOT NULL,
          Dni                   CHAR(9)         NOT NULL,
          FechaNac              DATE,
          Dirección             VARCHAR(30),
          Sexo                  CHAR,
          Sueldo                DECIMAL(10,2),
          SuperDni              CHAR(9),
          Dno                   INT             NOT NULL,
        PRIMARY KEY (Dni),
        FOREIGN KEY(SuperDni) REFERENCES EMPLEADO(Dni),
        FOREIGN KEY(Dno) REFERENCES DEPARTAMENTO(NúmeroDpto) );
CREATE TABLE DEPARTAMENTO
        ( NombreDpto            VARCHAR(15)     NOT NULL,
          NúmeroDpto            INT             NOT NULL,
          DniDirector           CHAR(9)         NOT NULL,
          FechaIngresoDirector  DATE,
        PRIMARY KEY(NúmeroDpto),
        UNIQUE(NombreDpto),
        FOREIGN KEY(DniDirector) REFERENCES EMPLEADO(Dni) );
CREATE TABLE LOCALIZACIONES_DPTO
        ( NúmeroDpto            INT             NOT NULL,
          UbicaciónDpto         VARCHAR(15)     NOT NULL,
        PRIMARY KEY(NúmeroDpto, UbicaciónDpto),
        FOREIGN KEY(NúmeroDpto) REFERENCES DEPARTAMENTO(NúmeroDpto) );
CREATE TABLE PROYECTO
        ( NombreProyecto        VARCHAR(15)     NOT NULL,
          NumProyecto           INT             NOT NULL,
          UbicaciónProyecto     VARCHAR(15),
          NumDptoProyecto       INT             NOT NULL,
        PRIMARY KEY(NumProyecto),
        UNIQUE(NombreProyecto),
        FOREIGN KEY(NumDptoProyecto) REFERENCES DEPARTAMENTO(NúmeroDpto) );
CREATE TABLE TRABAJA_EN
        (  DniEmpleado          CHAR(9)         NOT NULL,
           NumProy              INT             NOT NULL,
           Horas                DECIMAL(3,1)    NOT NULL,
        PRIMARY KEY(DniEmpleado, NumProy),
        FOREIGN KEY(DniEmpleado) REFERENCES EMPLEADO(Dni),
        FOREIGN KEY(NumProy) REFERENCES PROYECTO(NumProyecto) );
CREATE TABLE SUBORDINADO
        (  DniEmpleado          CHAR(9)         NOT NULL,
```

**Figura 8.1.** (*Continuación*).

```
    NombSubordinado      VARCHAR(15)    NOT NULL,
    Sexo                 CHAR,
    FechaNac             DATE,
    Relación             VARCHAR(8),
  PRIMARY KEY(DniEmpleado, NombSubordinado),
  FOREIGN KEY(DniEmpleado) REFERENCES EMPLEADO(Dni) );
```

Normalmente, el esquema SQL en el que se declaran las relaciones se especifica implícitamente en el entorno en el que se ejecuta la sentencia CREATE TABLE. De forma alternativa, podemos adjuntar explícitamente el nombre del esquema al nombre de la relación, separándolos con un punto. Por ejemplo, al escribir

```
CREATE TABLE EMPRESA.EMPLEADO . . .
```

en lugar de

```
CREATE TABLE EMPLEADO . . .
```

como en la Figura 8.1, podemos conseguir explícitamente (y no implícitamente) que la tabla EMPLEADO forme parte del esquema EMPRESA.

Las relaciones declaradas mediante sentencias CREATE TABLE se denominan **tablas base** (o relaciones base); esto significa que el DBMS crea y almacena como un fichero la relación y sus tuplas. Las relaciones base se distinguen de las **relaciones virtuales**, que se crean con CREATE VIEW (consulte la Sección 8.8), en que pueden o no corresponder a un fichero físico real. En SQL, los atributos de una tabla base están considerados como *ordenados en la secuencia en que se especificaron* en la setencia CREATE TABLE. No obstante, no se considera que las filas (tuplas) estén ordenadas dentro de una relación.

En la Figura 8.1 hay algunas claves externas (*foreign keys*) que pueden provocar errores porque o bien se especifican a través de referencias circulares, o porque se refieren a una tabla que todavía no se ha creado. Por ejemplo, la *foreign key* SuperDni de la tabla EMPLEADO es una referencia cruzada porque se refiere a la propia tabla. La *foreign key* Dno de la tabla EMPLEADO se refiere a la tabla DEPARTAMENTO, que todavía no se ha creado. Para tratar con este tipo de problema, estas restricciones pueden omitirse de la sentencia CREATE TABLE inicial, y añadirse más tarde con la sentencia ALTER TABLE (consulte la Sección 8.3.2).

### 8.1.3 Tipos de datos y dominios en SQL

Los **tipos de datos** básicos disponibles para los atributos son numérico, cadena de caracteres, cadena de bits, booleano, fecha y hora.

- El tipo de datos **numéricos** incluye los números enteros de varios tamaños (INTEGER o INT, y SMALLINT) así como los números en coma flotante (reales) de distintas precisiones (FLOAT o REAL, y DOUBLE PRECISION). Los números se pueden declarar utilizando DECIMAL(*i,j*) [o DEC(*i,j*) o NUMERIC(*i,j*)], donde *i,* la *precisión,* es el número total de dígitos decimales y *j,* la *escala*, es el número de dígitos después del punto decimal. El valor predeterminado para la escala es cero, mientras que la precisión predeterminada se define en la implementación.
- El tipo de datos **cadena de caracteres** puede ser de longitud fija [CHAR(*n*) o CHARACTER(*n*), donde *n* es la cantidad de caracteres] o de longitud variable [VARCHAR(*n*) o CHAR VARYING(*n*) o CHARACTER VARYING(*n*), donde *n* es la cantidad máxima de caracteres]. Al especificar un valor de cadena literal, se coloca entre comillas simples (apóstrofes) y distingue entre minúsculas y mayúsculas.[3] En el

[3] No es el caso con las palabras clave de SQL, como CREATE o CHAR. SQL no hace distinción entre mayúsculas y minúsculas con ellas, de modo que trata todas las letras por igual, sean mayúsculas o minúsculas.

caso de las cadenas de longitud fija, las cadenas más cortas se rellenan con caractereres en blanco por la derecha. Por ejemplo, si el valor 'Pérez' corresponde a un atributo de tipo CHAR(10), se rellena con cinco caracteres en blanco para convertirse en 'Pérez ', si es necesario. Los blancos de relleno normalmente se ignoran cuando se comparan cadenas. En las comparaciones, las cadenas se consideran ordenadas alfabéticamente (o lexicográficamente); si una cadena *cad1* aparece antes que otra cadena *cad2* en orden alfabético, entonces se considera que *cad1* es menor que *cad2*.[4] También existe un operador de concatenación representado por || (doble barra vertical) que permite concatenar dos cadenas en SQL. Por ejemplo, 'abc' || 'XYZ' da como resultado una sola cadena, 'abcXYZ'. En SQL-99 hay otro tipo de datos denominado CHARACTER LARGE OBJECT o CLOB, destinado a especificar columnas que tienen valores de texto más largos, como los documentos.

- El tipo de datos **cadena de bits** es de longitud fija *n* [BIT(*n*)] o de longitud variable [BIT VARYING(*n*)], donde *n* es la cantidad máxima de bits. El valor predeterminado para *n,* que es la longitud de una cadena de caracteres o de una cadena de bits, es 1. Las cadenas de bits literales se escriben entre comillas simples pero precedidas por una B para distinguirlas de las cadenas de caracteres; por ejemplo, B'10101'.[5] En SQL-99 existe otro tipo de datos denominado BINARY LARGE OBJECT o BLOB destinado a especificar las columnas que tienen valores binarios más grandes, como las imágenes.
- Un tipo de datos **booleano** tiene los valores tradicionales TRUE o FALSE. En SQL, debido a la presencia de los valores NULL, se utiliza una lógica de tres valores, que permite un tercer valor: UNKNOWN. En la Sección 8.5.1 explicaremos la necesidad de UNKNOWN y de la lógica de tres valores.
- En SQL2 se añadieron dos nuevos tipos de datos, **fecha** y **hora**. El tipo de datos DATE tiene diez posiciones y sus componentes son AÑO, MES y DÍA según la forma AAAA-MM-DD. El tipo de datos TIME tiene al menos ocho posiciones, con los componentes HORAS, MINUTOS y SEGUNDOS en la forma HH:MM:SS. La implementación de SQL sólo debe permitir las fechas y las horas válidas. La comparación < (menor que) se puede utilizar con fechas y horas (una fecha *anterior* se considera que es más pequeña que una fecha posterior, y lo mismo pasa con las horas). Los valores literales se representan mediante cadenas entre comillas simples precedidas por la palabra clave DATE o TIME; por ejemplo, DATE '2002-09-27' o TIME '09:12:47'. Además, un tipo de datos TIME(*i*), especifica *i* + 1 posiciones adicionales para TIME (una posición para un carácter separador adicional, e *i* posiciones para especificar las fracciones decimales de un segundo). Un tipo de datos TIME WITH TIME ZONE incluye seis posiciones adicionales para especificar el *desplazamiento* respecto a la zona horaria universal estándar, y que puede variar desde +13:00 a –12:59 en unidades de HORAS:MINUTOS. Si no se incluye WITH TIME ZONE, el valor predeterminado es la zona horaria local para la sesión de SQL.
- Un tipo de datos **marca de tiempo** (TIMESTAMP) incluye los campos DATE y TIME, más un mínimo de seis posiciones para las fracciones decimales de segundos y un calificador WITH TIME ZONE opcional. Los valores literales se representan como cadenas entre comillas simples precedidas por la palabra clave TIMESTAMP, con un espacio en blanco entre la fecha y la hora; por ejemplo, TIMESTAMP '2002-09-27 09:12:47 648302'.
- Otro tipo de datos relacionado con DATE, TIME y TIMESTAMP es INTERVAL, que permite especificar un **intervalo** (un *valor relativo* que puede utilizarse para incrementar o reducir el valor absoluto de una fecha, una hora o una marca de tiempo). Los intervalos están cualificados para ser intervalos AÑO/MES o intervalos DÍA/HORA.
- El formato de DATE, TIME y TIMESTAMP se puede considerar como un tipo especial de cadena. Por tanto, se pueden utilizar normalmente en comparaciones de cadena si se convierten en las cadenas equivalentes.

[4] En el caso de caracteres no alfabéticos hay un orden definido.

[5] Las cadenas de bits cuya longitud es un múltiplo de 4 se pueden especificar en notación *hexadecimal*, donde la cadena literal va precedida por una X y cada carácter hexadecimal representa 4 bits.

Es posible especificar directamente el tipo de datos de cada atributo, como en la Figura 8.1; alternativamente, se puede declarar un dominio, y el nombre del dominio se puede utilizar con la especificación del atributo. Esto hace más fácil cambiar el tipo de datos de un dominio que es utilizado por numerosos atributos de un esquema, y mejorar la legibilidad de este último. Por ejemplo, podemos crear un dominio TIPO_DNI con la siguiente sentencia:

```
CREATE DOMAIN TIPO_DNI AS CHAR(9);
```

En la Figura 8.1 podemos utilizar TIPO_DNI en lugar de CHAR(9) para los atributos Dni y SuperDni de Empleado, DniDirector de Departamento, DniEmpleado de TRABAJA_EN y DniEmpleado de SUBORDINADO. Un dominio también puede tener una especificación predeterminada opcional a través de una cláusula DEFAULT, como explicamos más adelante para los atributos. Los dominios no están disponibles en muchas implementaciones de SQL.

# 8.2 Especificación de restricciones en SQL

Esta sección describe las restricciones básicas que se pueden especificar en SQL como parte de la creación de una tabla. Entre ellas podemos citar las restricciones de clave y de integridad referencial, así como las restricciones en los dominios de atributo y NULLs y en las tuplas individuales dentro de una relación. En la Sección 8.7 explicaremos la especificación de las restricciones más generales, denominadas aserciones.

## 8.2.1 Especificación de restricciones de atributo y valores predeterminados de atributo

Como SQL permite NULL como valor de atributo, es posible especificar una *restricción* NOT NULL si no se permite NULL para un atributo en particular. Esto siempre se especifica implícitamente para los atributos que forman parte de la *clave principal* de cada relación, pero puede especificarse para cualquier otro atributo para cuyo valor se exige que no sea NULL (véase la Figura 8.1).

También es posible definir un *valor predeterminado* para un atributo añadiendo la cláusula **DEFAULT** <valor> a su definición. El valor predeterminado se incluye en cualquier tupla nueva si no se proporciona un valor explícito para ese atributo. La Figura 8.2 ilustra un ejemplo de cómo especificar un director predeteminado para un departamento nuevo y un departamento predeterminado para un empleado nuevo. Si no se especifica una cláusula predeterminada, el *valor predeterminado* es NULL para los atributos *que no tienen* la restricción NOT NULL.

Otro tipo de restricción puede ser restringir los valores de atributo o dominio con la cláusula **CHECK** a continuación de la definición de un atributo o dominio.[6] Por ejemplo, suponga que los números de departamento están restringidos a número enteros entre 1 y 20; entonces, podemos cambiar la declaración de atributo de NumeroDpto en la tabla DEPARTAMENTO (véase la Figura 8.1) a lo siguiente:

```
NumeroDpto INT NOT NULL CHECK (NumeroDpto > 0 AND NumeroDpto < 21);
```

La cláusula CHECK también se puede utilizar en combinación con la sentencia CREATE DOMAIN. Por ejemplo, podemos escribir la siguiente sentencia:

```
CREATE DOMAIN NUM_D AS INTEGER CHECK
    (NUM_D > 0 AND NUM_D < 21);
```

[6] La cláusula CHECK también se puede utilizar con otros fines, como veremos.

**Figura 8.2.** Ejemplo de cómo especificar en SQL los valores de atributo predeterminados y las acciones de activación de la integridad referencial.

```
CREATE TABLE EMPLEADO
        ( . . .
          Dno          INT          NOT NULL      DEFAULT 1,
        CONSTRAINT EMPPK
          PRIMARY KEY(Dni),
        CONSTRAINT SUPERFKEMP
          FOREIGN KEY(SuperDni) REFERENCES EMPLEADO(Dni)
            ON DELETE SET NULL      ON UPDATE CASCADE
        CONSTRAINT EMPDEPTFK
          FOREIGN KEY(Dno) REFERENCES DEPARTAMENTO(NumeroDpto)
            ON DELETE SET DEFAULT ON UPDATE CASCADE );
CREATE TABLE DEPARTAMENTO
        ( . . . ,
          DniDirector    CHAR(9)       NOT NULL      DEFAULT '888665555',
          . . . ,
        CONSTRAINT DEPTPK
          PRIMARY KEY(NumeroDpto),
        CONSTRAINT DEPTSK
          UNIQUE(NombreDpto),
        CONSTRAINT DEPTMGRFK
          FOREIGN KEY(DniDirector) REFERENCES EMPLEADO(Dni)
                ON DELETE SET DEFAULT    ON UPDATE CASCADE );
CREATE TABLE LOCALIZACIONES_DPTO
        ( . . . ,
        PRIMARY KEY(NumeroDpto, UbicaciónDpto),
        FOREIGN KEY(NumeroDpto) REFERENCES DEPARTAMENTO(NúmeroDpto)
                ON DELETE CASCADE        ON UPDATE CASCADE );
```

Podemos utilizar entonces el dominio NUM_D creado como tipo de atributo para todos los atributos que se refieran a los números de departamento de la Figura 8.1, como NumeroDpto de Departamento, NumDptoProyecto de PROYECTO, Dno de EMPLEADO, etcétera.

### 8.2.2 Especificación de las restricciones de clave y de integridad referencial

Como las restricciones de clave e integridad referencial son muy importantes, hay cláusulas especiales para la sentencia CREATE TABLE. En la Figura 8.1 se muestran algunos ejemplos para ilustrar la especificación de claves y la integridad referencial.[7] La cláusula **PRIMARY KEY** especifica uno o más atributos que constituyen la clave principal de una relación. Si una clave principal sólo tiene un atributo, la cláusula puede seguir al atributo directamente. Por ejemplo, la clave principal de DEPARTAMENTO se puede especificar como sigue (en lugar de como se muestra en la Figura 8.1):

```
NumeroDpto INT PRIMARY KEY;
```

[7] Las restricciones de clave y de integridad referencial no se incluían en las versiones anteriores de SQL. En algunas de esas implementaciones, las claves se especificaban implícitamente en el nivel interno mediante el comando CREATE INDEX.

La cláusula **UNIQUE** especifica claves (secundarias) alternativas, como se ilustraba en la declaración de las tablas DEPARTAMENTO y PROYECTO de la Figura 8.1.

La integridad referencial se especifica mediante la cláusula **FOREIGN KEY** (véase la Figura 8.1). Como se explicó en la Sección 5.2.4, una restricción de integridad referencial se puede violar cuando se insertan o eliminan tuplas o cuando se modifica el valor de un atributo de la *foreign key* o de la clave principal. La acción predeterminada que SQL toma en caso de una violación de la integridad es **rechazar** la operación de actualización que provocaría tal violación. Sin embargo, el diseñador del esquema puede especificar una acción alternativa si se viola la integridad referencial añadiendo una cláusula de **acción de activación referencial** a cualquier restricción de *foreign key*. Las opciones son SET NULL, CASCADE y SET DEFAULT. Una opción debe cualificarse con ON DELETE u ON UPDATE. Ilustramos esto con los ejemplos de la Figura 8.2. Aquí, el diseñador de la base de datos eligió SET NULL ON DELETE y CASCADE ON UPDATE para la *foreign key* SuperDni de EMPLEADO. Esto significa que si se *elimina* la tupla de un empleado supervisor, el valor de SuperDni se establece automáticamente a NULL en todas las tuplas de empleado que hacían referencia a la tupla de empleado borrada. Por el contrario, si se *actualiza* el valor Dni de un empleado supervisor (por ejemplo, porque se introdujo incorrectamente), entonces el valor nuevo se *actualiza en cascada* para el SuperDni de todas las tuplas de empleado que hacen referencia a la tupla de empleado actualizada.[8]

En general, la acción tomada por el DBMS para SET NULL o SET DEFAULT es la misma tanto para ON DELETE como para ON UPDATE: el valor de los atributos de referencia afectados se cambia a NULL para SET NULL y al valor predeterminado especificado para SET DEFAULT. La acción para CASCADE ON DELETE es eliminar todas las tuplas referenciadas, mientras que la acción para CASCADE ON UPDATE es cambiar el valor de la *foreign key* por el valor de la clave principal (nueva) actualizada en todas las tuplas referenciadas. Es responsabilidad del diseñador de la base de datos elegir la acción apropiada y especificarla en el esquema de la base de datos. Como regla general, la opción CASCADE es adecuada para las relaciones "de relación" (consulte la Sección 7.1), como TRABAJA_EN; para las relaciones que representan atributos multivalor, como LOCALIZACIONES_DPTO; y para las relaciones que representan tipos de entidad débiles, como SUBORDINADO.

### 8.2.3 Asignación de nombres a las restricciones

La Figura 8.2 también ilustra cómo puede asignarse un **nombre de restricción** a una restricción, con la palabra clave **CONSTRAINT**. Los nombres de todas las restricciones dentro de un esquema particular deben ser únicos. El nombre de una restricción se utiliza para identificar una restricción particular en caso de que la restricción tenga que eliminarse más tarde y sustituirse por otra restricción, como se explica en la Sección 8.3. La asignación de nombres a las restricciones es opcional.

### 8.2.4 Especificación de restricciones en las tuplas utilizando CHECK

Además de las restricciones de clave y de integridad referencial, que se especifican mediante palabras claves especiales, se pueden indicar otras *restricciones de tabla* mediante cláusulas CHECK adicionales al final de una sentencia CREATE TABLE. Estas restricciones se pueden denominar **basadas en tuplas** porque se aplican *individualmente* a cada tupla y se comprueban siempre que se inserta o modifica una tupla. Por ejemplo, suponga que la tabla DEPARTAMENTO de la Figura 8.1 tiene un atributo adicional FechaCreaciónDpto, que almacena la fecha en que se creó el departamento. Después, podríamos añadir la siguiente cláusula CHECK al final de la sentencia CREATE TABLE para la tabla DEPARTAMENTO para garantizar que la fecha de inicio de un director es posterior a la fecha de creación del departamento.

[8] La *foreign key* SuperDni de la tabla EMPLEADO es una referencia circular y, por tanto, es posible que tenga que añadirse más tarde como una restricción con nombre utilizando la sentencia ALTER TABLE, como explicamos al final de la Sección 8.1.2.

```
CHECK (FechaCreaciónDpto <= FechaIngresoDirector);
```

La cláusula CHECK también se puede utilizar para especificar restricciones más generales utilizando la sentencia CREATE ASSERTION de SQL. Lo explicamos en la Sección 8.7 porque requiere toda la potencia de las consultas, que se explican en las Secciones 8.4 y 8.5.

# 8.3 Sentencias de SQL para cambiar el esquema

En esta sección ofrecemos una panorámica de los **comandos de evolución del esquema** de SQL, que se pueden utilizar para alterar un esquema añadiendo o eliminando tablas, atributos, restricciones y otros elementos del esquema.

## 8.3.1 Comando DROP

El comando DROP se puede utilizar para eliminar los elementos con nombre del esquema, como tablas, dominios o restricciones. También puede eliminar un esquema. Por ejemplo, si ya no se necesita un esquema entero, se puede utilizar el comando DROP SCHEMA. Hay dos opciones de comportamiento para estas eliminaciones: CASCADE y RESTRICT. Por ejemplo, para eliminar el esquema de la base de datos EMPRESA y todas sus tablas, dominios y otros elementos, se utiliza la opción CASCADE de este modo:

```
DROP SCHEMA EMPRESA CASCADE;
```

Si se opta por RESTRICT en lugar de CASCADE, el esquema sólo se elimina si *no contiene elementos*; en caso contrario, el comando DROP no se ejecutará.

Si ya no se necesita una relación base dentro de un esquema, la relación y su definición se pueden eliminar con el comando DROP TABLE. Por ejemplo, si ya no queremos hacer un seguimiento de los subordinados de los empleados en la base de datos EMPRESA de la Figura 8.1, podemos librarnos de la relación SUBORDINADO ejecutando el siguiente comando:

```
DROP TABLE SUBORDINADO CASCADE;
```

Si se elige la opción RESTRICT en lugar de CASCADE, la tabla sólo se elimina si no se hace referencia a ella en ninguna restricción (por ejemplo, desde las definiciones de *foreign key* de otra relación) o vista (consulte la Sección 8.8). Con la opción CASCADE, todas estas restricciones y vistas que hacen referencia a la tabla se eliminan automáticamente del esquema, junto con la propia tabla.

El comando DROP TABLE no sólo elimina todos los registros de la tabla, sino también la definición de la tabla del catálogo. Si se desea eliminar los registros, pero manteniendo la definición de la tabla para un uso futuro, entonces hay que utilizar el comando DELETE (consulte la Sección 8.6.2) en lugar de DROP TABLE.

El comando DROP también se puede utilizar para eliminar otros tipos de elementos con nombre del esquema, como las restricciones o los dominios.

## 8.3.2 Comando ALTER

La definición de una tabla base o de otros elementos con nombre del esquema se puede cambiar con el comando ALTER. Para las tablas base, las posibles acciones de alteración incluyen la adición o eliminación de una columna (atributo), el cambio de la definición de una columna, y la adición o eliminación de restricciones. Por ejemplo, para añadir a las relaciones base EMPLEADO del esquema EMPRESA un atributo que sirva para hacer un seguimiento de los trabajos de los empleados, podemos usar este comando:

```
ALTER TABLE EMPRESA.EMPLEADO ADD COLUMN Trabajo VARCHAR(12);
```

Todavía debemos introducir un valor para el atributo nuevo, Trabajo, por cada tupla EMPLEADO. Lo podemos hacer especificando una cláusula predeterminada o utilizando el comando UPDATE (consulte la Sección 8.6). Si no especificamos una cláusula predeterminada, el atributo nuevo tendrá NULL en todas las tuplas de la relación inmediatamente después de haberse ejecutado el comando; por tanto, la restricción NOT NULL *no está permitida* en este caso.

Para eliminar una columna, debemos elegir CASCADE o RESTRICT como comportamiento de eliminación. En el caso de CASCADE, todas las restricciones y vistas que hacen referencia a la columna se eliminarán automáticamente del esquema, junto con la columna. Si optamos por RESTRICT, el comando es satisfactorio sólo si no hay vistas o restricciones (u otros elementos) que hagan referencia a la columna. Por ejemplo, el siguiente comando elimina el atributo Dirección de la tabla base EMPLEADO:

```
ALTER TABLE EMPRESA.EMPLEADO DROP COLUMN Dirección CASCADE;
```

También es posible alterar la definición de una columna eliminando una cláusula predeterminada existente o definiendo una nueva cláusula predeterminada. Los siguientes ejemplos sirven como explicación:

```
ALTER TABLE EMPRESA.DEPARTAMENTO ALTER COLUMN DniDirector
    DROP DEFAULT;
ALTER TABLE EMPRESA.DEPARTAMENTO ALTER COLUMN DniDirector
    SET DEFAULT '333445555';
```

También se pueden cambiar las restricciones especificadas en una tabla añadiendo o eliminando una restricción. Para ser eliminada, la restricción debe contar con un nombre asignado durante su definición. Por ejemplo, para eliminar la restricción SUPERFKEMP de la relación EMPLEADO (véase la Figura 8.2) escribimos:

```
ALTER TABLE EMPRESA.EMPLEADO DROP CONSTRAINT SUPERFKEMP CASCADE;
```

Una vez hecho esto, podemos redefinir una restricción de sustitución añadiendo, si es necesario, una restricción nueva a la relación. Esto se consigue utilizando la palabra clave **ADD** en la sentencia ALTER TABLE seguida por la restricción nueva, que puede o no tener nombre, y que puede ser de cualquiera de los tipos de restricción de tabla explicados.

Las subsecciones anteriores han ofrecido una panorámica de los comandos de evolución del esquema de SQL. Hay otros muchos detalles y opciones, por lo que instamos al lector a consultar los documentos citados en la sección "Bibliografía seleccionada", al final de este capítulo. Las dos secciones siguientes explican las capacidades de consulta de SQL.

# 8.4 Consultas básicas en SQL

SQL tiene una sentencia básica para recuperar información de una base de datos: **SELECT**. Esta sentencia *no tiene relación* con la operación SELECCIÓN del álgebra relacional, que explicamos en el Capítulo 6. En SQL hay muchas opciones y versiones de la sentencia SELECT, por lo que introduciremos sus características gradualmente. Utilizaremos las consultas de ejemplo especificadas en el esquema de la Figura 5.5 y nos referiremos al estado de la base de datos de la Figura 5.6 para mostrar los resultados de algunas consultas.

Antes de continuar, debemos hacer una *distinción importante* entre SQL y el modelo relacional formal explicado en el Capítulo 5. SQL permite que una tabla (relación) tenga dos o más tuplas idénticas en todos sus valores de atributo. Por tanto, en general, una tabla SQL no es un *conjunto de tuplas*, porque un conjunto no permite dos miembros idénticos; en su lugar, es un **multiconjunto** (a veces conocido como *bolsa*) de tuplas. Algunas relaciones SQL están *restringidas a ser conjuntos* porque se ha declarado una restricción de clave o porque se ha utilizado la opción DISTINCT con la sentencia SELECT (se explica más adelante en esta sección). Debemos ser conscientes de esta distinción durante la explicación de los ejemplos.

### 8.4.1 Estructura SELECT-FROM-WHERE de las consultas básicas de SQL

Las consultas en SQL pueden ser muy complejas. Empezaremos con las sencillas e iremos progresando, paso a paso, hasta las más complejas. La forma básica de la sentencia SELECT, denominada en ocasiones **mapeado** o **bloque select-from-where**, está formada por las cláusulas SELECT, FROM y WHERE y tiene la siguiente forma:[9]

```
SELECT   <lista de atributos>
FROM     <lista de tablas>
WHERE    <condición>;
```

donde:

- <lista de atributos> es una lista de los atributos cuyos valores serán recuperados por la consulta.
- <lista de tablas> es una lista de las relaciones necesarias para procesar la consulta.
- <condición> es una expresión condicional (booleana) que identifica las tuplas que la consulta recuperará.

En SQL, los operadores básicos para comparar lógicamente los valores de los atributos entre sí y con constantes literales son =, <, <=, >, >= y <>, que se corresponden con los operadores =, <, ?, >, ? y ? del álgebra relacional, respectivamente, y con los operadores =, <, <=, >, >= y != del lenguaje de programación C/C++. La diferencia principal es el operador *no igual*. SQL tiene muchos operadores de comparación adicionales que iremos presentando a medida que los necesitemos.

Vamos a ilustrar la sentencia SELECT básica con algunas consultas de ejemplo. Las consultas están etiquetadas con los mismos números que en el Capítulo 6.

**Consulta 0.** Recuperar la fecha de nacimiento y la dirección del empleado (o empleados) cuyo nombre sea José Pérez Pérez.

```
C0:  SELECT   FechaNac, Dirección
     FROM     EMPLEADO
     WHERE    Nombre='José' AND Apellido1='Pérez' AND Apellido2='Pérez';
```

Esta consulta sólo implica a la relación EMPLEADO que se especifica con la cláusula FROM. La consulta *selecciona* las tuplas de EMPLEADO que satisfacen la condición de la cláusula WHERE; después, *proyecta* el resultado en los atributos FechaNac y Dirección enumerados en la cláusula SELECT. C0 es parecida a la siguiente expresión escrita en álgebra relacional, excepto que *no* se eliminan los duplicados, si los hay:

$$\pi_{\text{FechaNac,Dirección}}(\sigma_{\text{Nombre='José' AND Apellido1='Pérez' AND Apellido2='Pérez'}}\ (\text{EMPLEADO}))$$

Por tanto, una consulta SQL sencilla con una sola relación en la cláusula FROM es similar a una pareja de operaciones SELECCIÓN-PROYECTO del álgebra relacional. La cláusula SELECT de SQL especifica los *atributos de proyección* y la cláusula WHERE especifica la *condición de selección*. La única diferencia es que en la consulta SQL podemos obtener tuplas duplicadas en el resultado porque no se implementa la restricción de que una relación es un conjunto. La Figura 8.3(a) muestra el resultado de la consulta C0 sobre la base de datos de la Figura 5.6.

La consulta C0 también es parecida a la siguiente expresión de cálculo relacional de tuplas, excepto que los duplicados, si los hay, tampoco serían eliminados en la consulta SQL:

```
C0:  { t.FechaNac, t.Dirección | EMPLEADO(t) AND t.Nombre='José' AND t.Apellido1='Pérez'
       AND t.Apellido2='Pérez' }
```

[9] Las cláusulas SELECT y FROM son necesarias en todas las consultas SQL. WHERE es opcional (consulte la Sección 8.4.3).

**Figura 8.3.** Resultado de las consultas SQL aplicadas sobre el estado de la base de datos EMPRESA de la Figura 5.6. (a) C0. (b) C1. (c) C2. (d) C8. (e) C9. (f) C10. (g) C1C.

**(a)**

| FechaNac | Dirección |
|---|---|
| 01-09-1965 | Eloy I, 98 |

**(b)**

| Nombre | Apellido1 | Dirección |
|---|---|---|
| José | Pérez | Eloy I, 98 |
| Alberto | Campos | Avda. Ríos, 9 |
| Fernando | Ojeda | Portillo, s/n |
| Aurora | Oliva | Antón, 6 |

**(c)**

| NumProyecto | NumDptoProyecto | Apellido1 | Dirección | FechaNac |
|---|---|---|---|---|
| 10 | 4 | Sainz | Cerquillas, 67 | 20-06-1941 |
| 30 | 4 | Sainz | Cerquillas, 67 | 20-06-1941 |

**(d)**

| E.Nombre | E.Apellido1 | S.Nombre | S.Apellido1 |
|---|---|---|---|
| José | Pérez | Alberto | Campos |
| Alberto | Campos | Eduardo | Ochoa |
| Alicia | Jiménez | Juana | Sainz |
| Juana | Sainz | Eduardo | Ochoa |
| Fernando | Ojeda | Alberto | Campos |
| Aurora | Oliva | Alberto | Campos |
| Luis | Pajares | Juana | Sainz |

**(e)**

| E.Nombre |
|---|
| 123456789 |
| 333445555 |
| 999887777 |
| 987654321 |
| 666884444 |
| 453453453 |
| 987987987 |
| 888665555 |

**(f)**

| Dni | NombreDpto |
|---|---|
| 123456789 | Investigación |
| 333445555 | Investigación |
| 999887777 | Investigación |
| 987654321 | Investigación |
| 666884444 | Investigación |
| 453453453 | Investigación |
| 987987987 | Investigación |
| 888665555 | Investigación |
| 123456789 | Administración |
| 333445555 | Administración |
| 999887777 | Administración |
| 987654321 | Administración |
| 666884444 | Administración |
| 453453453 | Administración |
| 987987987 | Administración |
| 888665555 | Administración |
| 123456789 | Sede central |
| 333445555 | Sede central |
| 999887777 | Sede central |
| 987654321 | Sede central |
| 666884444 | Sede central |
| 453453453 | Sede central |
| 987987987 | Sede central |
| 888665555 | Sede central |

**(g)**

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|
| José | Pérez | Pérez | 123456789 | 01-09-1965 | Eloy I, 98 | H | 30000 | 333445555 | 5 |
| Alberto | Campos | Sastre | 333445555 | 08-12-1955 | Avda. Ríos, 9 | H | 40000 | 888665555 | 5 |
| Fernando | Ojeda | Ordóñez | 666884444 | 15-09-1962 | Portillo, s/n | H | 38000 | 333445555 | 5 |
| Aurora | Oliva | Avezuela | 453453453 | 31-07-1972 | Antón, 6 | M | 25000 | 333445555 | 5 |

Por tanto, podemos pensar en una variable de tupla implícita en la consulta SQL pasando por cada tupla de la tabla EMPLEADO y evaluando la condición de la cláusula WHERE. Sólo se seleccionan las tuplas que satisfacen la condición (es decir, las tuplas para las que la condición se evalúa como TRUE después de sustituir sus correspondientes valores de atributo).

**Consulta 1.** Recuperar el nombre y la dirección de todos los empleados que trabajan en el departamento 'Investigación'.

```
C1:  SELECT   Nombre, Apellido1, Dirección
     FROM     EMPLEADO, DEPARTAMENTO
     WHERE    NombreDpto='Investigación' AND NumeroDpto=Dno;
```

La consulta C1 es parecida a una secuencia SELECCIÓN-PROYECCIÓN-CONCATENACIÓN de operaciones del álgebra relacional. Dichas consultas se denominan a veces **consultas selección-proyección-concatenación**. En la cláusula WHERE de C1, la condición NombreDpto='Investigación' es una **condición de selección** y se corresponde con una operación SELECCIÓN del álgebra relacional. La condición NumeroDpto=Dno es una **condición de concatenación**, que corresponde a una condición CONCATENACIÓN del álgebra relacional. El resultado de la consulta C1 se muestra en la Figura 8.3(b). En general, en una consulta SQL sencilla puede especificarse cualquier cantidad de condiciones de selección y concatenación. El siguiente ejemplo es una consulta selección-proyección-concatenación con *dos* condiciones de concatenación.

**Consulta 2.** Por cada proyecto ubicado en 'Gijón', mostrar su número, el número del departamento que lo gestiona y el primer apellido, dirección y fecha de nacimiento del director del mismo.

```
C2:  SELECT   NumProyecto, NumDptoProyecto, Apellido1, Dirección, FechaNac
     FROM     PROYECTO, DEPARTAMENTO, EMPLEADO
     WHERE    NumDptoProyecto=NumeroDpto AND DniDirector=Dni AND
              UbicaciónProyecto='Gijón';
```

La condición de concatenación NumDptoProyecto = NumeroDpto relaciona un proyecto con el departamento que lo controla, mientras que la condición de concatenación DniDirector=Dni relaciona el departamento de control con el empleado que lo administra. El resultado de la consulta C2 se muestra en la Figura 8.3(c).

## 8.4.2 Nombres de atributo ambiguos, alias y variables de tupla

En SQL el mismo nombre se puede utilizar para dos (o más) atributos, siempre y cuando los atributos se encuentren en *relaciones diferentes*. Si es el caso, y una consulta se refiere a dos o más atributos que tienen el mismo nombre, debemos **calificar** el nombre del atributo con el nombre de la relación a fin de evitar la ambigüedad. Esto se consigue colocando como *prefijo* el nombre de la relación al nombre del atributo, y separando los dos nombres con un punto. A modo de ilustración, suponga que en las Figuras 5.5 y 5.6, los atributos Dno y Apellido1 de la relación EMPLEADO se llamaran NumeroDpto y Nombre, y que el atributo NombreDpto de DEPARTAMENTO también se llamara Nombre; entonces, para evitar la ambigüedad, la consulta C1 tendría que reformarse como aparece en la consulta C1A. Hemos añadido un prefijo a los atributos Nombre y NumeroDpto para especificar a cuáles nos estamos refiriendo en realidad, ya que estos nombres de atributo se utilizan en dos relaciones:

```
C1A: SELECT   Nombre, EMPLEADO.Nombre, Dirección
     FROM     EMPLEADO, DEPARTAMENTO
     WHERE    DEPARTAMENTO.Nombre='Investigación' AND
              DEPARTAMENTO.NumeroDpto=EMPLEADO.NumeroDpto;
```

La ambigüedad también aparece en el caso de las consultas que se refieren dos veces a la misma relación, como en el siguiente ejemplo:

**Consulta 8.** Por cada empleado, recuperar el nombre y el primer apellido del empleado, y el nombre y el primer apellido de su supervisor inmediato.

```
C8:  SELECT  E.Nombre, E.Apellido1, S.Nombre, S.Apellido1
     FROM    EMPLEADO AS E, EMPLEADO AS S
     WHERE   E.SuperDni=S.Dni;
```

En este caso, nos permite declarar nombres de relación alternativos, E y S, denominados **alias** o **variables de tupla**, para la relación EMPLEADO. Un alias puede seguir a la palabra clave **AS**, como se muestra en C8, o puede seguir directamente al nombre de la relación (por ejemplo, escribiendo EMPLEADO E, EMPLEADO S en la cláusula FROM de C8). También es posible renombrar los atributos de la relación dentro de la consulta SQL, asignándoles unos alias. Por ejemplo, si escribimos:

```
EMPLEADO AS E(Np, A1, A2, Dni, Fn, Dir, Sex, Sal, Sdni, Dno)
```

en la cláusula FROM, Np será el alias de Nombre, A1 de Apellido1, A2 de Apellido2, etcétera.

En C8, podemos pensar que E y S son dos *copias diferentes* de la relación EMPLEADO; la primera, E, representa a los empleados en el papel de supervisados; la segunda, S, representa a los empleados en el papel de supervisores. Ahora podemos concatenar las dos copias. Por supuesto, en la realidad *sólo hay una* relación EMPLEADO, y la condición de concatenación significa unir la relación consigo misma haciendo coincidir las tuplas que satisfagan la condición de concatenación E.SuperDni=S.Dni. Esto es un ejemplo de consulta recursiva de un solo nivel, como explicamos en la Sección 6.4.2. En versiones anteriores de SQL, como en el álgebra relacional, no era posible especificar una consulta recursiva general, con un número desconocido de niveles, en una sola sentencia SQL. En SQL-99 se ha incorporado una estructura para especificar consultas recursivas, como se describe en el Capítulo 22.

El resultado de la consulta C8 se muestra en la Figura 8.3(d). Siempre que se otorguen uno o más alias a una relación, podemos utilizar esos nombres para representar diferentes referencias a esa relación. Esto permite múltiples referencias a la misma relación dentro de una consulta. Si queremos, podemos utilizar este mecanismo alias-denominación en cualquier consulta SQL para especificar en la cláusula WHERE variables de tupla para cada tabla, necesite o no la misma relación ser referenciada más de una vez. De hecho, esta práctica es recomendable porque da lugar a consultas más fáciles de comprender. Por ejemplo, podríamos especificar la consulta C1A como en C1B:

```
C1B: SELECT  E.Nombre, E.NombreC, E.Dirección
     FROM    EMPLEADO E, DEPARTAMENTO D
     WHERE   D.Nombre='Investigación' AND D.NumeroDpto=E.NumeroDpto;
```

Si especificamos variables de tupla para cada tabla de la cláusula WHERE, una consulta sección-proyección-concatenación en SQL se parece mucho a la correspondiente expresión de cálculo relacional de tuplas (excepto por la eliminación de los duplicados). Por ejemplo, compare C1B con la siguiente expresión de cálculo relacional de tuplas sobre las tablas originales:

```
C1: {e.Nombre, e.Apellido1, e.Dirección | EMPLEADO(e) AND (∃d)
     (DEPARTAMENTO(d) AND d.NombreDpto='Investigación' AND d.NumeroDpto=e.Dno)}
```

La principal diferencia (aparte de la sintaxis) es que en la consulta SQL el cuantificador existencial no se especifica explícitamente.

## 8.4.3 Cláusula WHERE no especificada y uso del asterisco

Vamos a ver dos características más de SQL. La ausencia de una cláusula WHERE indica que no hay una condición en la selección de tuplas; por tanto, *todas las tuplas* de la relación especificada en la cláusula FROM

se califican y seleccionan para la consulta resultante. Si en la cláusula FROM se especifica más de una relación y no hay una cláusula WHERE, entonces se selecciona el PRODUCTO CRUZADO (*todas las posibles combinaciones de tuplas*) de esas relaciones. Por ejemplo, la Consulta 9 selecciona todos los DNIs de EMPLEADO (véase la Figura 8.3[e]), y la Consulta 10 selecciona todas las combinaciones de un Dni de EMPLEADO y el NombreDpto de un DEPARTAMENTO (véase la Figura 8.3[f]).

**Consultas 9 y 10.** Seleccione todos los Dni de EMPLEADO (C9) y todas las combinaciones de Dni de EMPLEADO y NombreDpto de DEPARTAMENTO (C10) en la base de datos.

```
C9:   SELECT   Dni
      FROM     EMPLEADO;

C10:  SELECT   Dni, NombreDpto
      FROM     EMPLEADO, DEPARTAMENTO;
```

Es extremadamente importante especificar todas las selecciones y condiciones de concatenación en la cláusula WHERE; si se omite cualquiera de esas condiciones, pueden obtenerse relaciones incorrectas y muy grandes. La C10 es parecida a una operación PRODUCTO CRUZADO seguida por una operación PROYECCIÓN en el álgebra relacional. Si especificamos todos los atributos de EMPLEADO y DEPARTAMENTO en C10, obtenemos el PRODUCTO CRUZADO (excepto por la eliminación de duplicados, si los hay).

Para recuperar todos los valores de atributo de las tuplas seleccionadas, no tenemos que listar explícitamente los nombres de los atributos en SQL; podemos escribir un *asterisco* (*), que tiene el significado de *todos los atributos*. Por ejemplo, la consulta C1C recupera todos los valores de atributo de cualquier EMPLEADO que trabaje en el DEPARTAMENTO número 5 (véase la Figura 8.3[g]), la consulta C1D recupera todos los atributos de un EMPLEADO y los atributos del DEPARTAMENTO en el que trabaja por cada empleado del departamento 'Investigación', y C10A especifica el PRODUCTO CRUZADO de las relaciones EMPLEADO y DEPARTAMENTO.

```
C1C:  SELECT   *
      FROM     EMPLEADO
      WHERE    Dno=5;

C1D:  SELECT   *
      FROM     EMPLEADO, DEPARTAMENTO
      WHERE    NombreDpto='Investigación' AND Dno=NumeroDpto;

C10A: SELECT   *
      FROM     EMPLEADO, DEPARTAMENTO;
```

## 8.4.4 Tablas como conjuntos en SQL

Como mencionamos anteriormente, SQL trata normalmente a una tabla no como un conjunto, sino como un **multiconjunto**; *las tuplas duplicadas pueden aparecer más de una vez* en una tabla, y en el resultado de una consulta. SQL no elimina automáticamente las tuplas duplicadas en los resultados de las consultas, por las siguientes razones:

- La eliminación de duplicados es una operación muy costosa. Una forma de implementarla consiste en ordenar primero las tuplas y después eliminar los duplicados.
- El usuario puede querer ver las tuplas duplicadas en el resultado de una consulta.
- Cuando se aplica una función de agregación (consulte la Sección 8.5.7) a las tuplas, en la mayoría de los casos no queremos eliminar los duplicados.

**Figura 8.4.** Resultados de distintas consultas SQL aplicadas sobre el estado de la base de datos EMPRESA de la Figura 5.6. (a) C11. (b) C11A. (c) C16. (d) C18.

**(a)**

| Sueldo |
|---|
| 30000 |
| 40000 |
| 25000 |
| 43000 |
| 38000 |
| 25000 |
| 25000 |
| 55000 |

**(b)**

| Sueldo |
|---|
| 30000 |
| 40000 |
| 25000 |
| 43000 |
| 38000 |
| 55000 |

**(c)**

| Nombre | Apellido1 |
|---|---|

**(d)**

| Nombre | Apellido1 |
|---|---|
| Eduardo | Ochoa |

Una tabla SQL con una clave está restringida a ser un conjunto, ya que el valor de la clave debe ser distinto en cada tupla.[10] Si *queremos* eliminar las tuplas duplicadas del resultado de una consulta SQL, utilizamos la palabra clave **DISTINCT** en la cláusula SELECT, lo que significa que sólo las tuplas distintas deben permanecer en el resultado. En general, una consulta con SELECT DISTINCT elimina los duplicados, mientras que una consulta con SELECT ALL no lo hace. Especificar SELECT sin ALL ni DISTINCT (como en nuestros anteriores ejemplos) es equivalente a SELECT ALL. Por ejemplo, C11 recupera el sueldo de los empleados; si varios empleados tienen el mismo sueldo, ese valor aparecerá varias veces en el resultado de la consulta (véase la Figura 8.4[a]). Si sólo estamos interesados en los valores de sueldo diferentes, será deseable que cada valor aparezca una sola vez, independientemente del número de empleados que ganen ese sueldo. Con la palabra clave **DISTINCT** como en C11A, conseguimos lo mencionado (véase la Figura 8.4[b]).

**Consulta 11.** Recuperar el sueldo de todos los empleados (C11) y todos los valores de sueldo que son distintos (C11A).

```
C11:  SELECT   ALL Sueldo
      FROM     EMPLEADO;

C11A: SELECT   DISTINCT Sueldo
      FROM     EMPLEADO;
```

SQL ha incorporado directamente algunas de las operaciones del álgebra relacional: unión de conjuntos (**UNION**), diferencia de conjuntos (**EXCEPT**)[11] e intersección de conjuntos (**INTERSECT**). Las relaciones resultantes de estas operaciones son conjuntos de tuplas; es decir, *las tuplas duplicadas son eliminadas del resultado*. Como estas operaciones con conjuntos sólo se aplican a las *relaciones compatibles con la unión*, debemos asegurarnos de que las dos relaciones sobre las que apliquemos la operación tengan los mismos atributos y que éstos aparezcan en el mismo orden en las dos relaciones. El siguiente ejemplo ilustra el uso de UNION.

**Consulta 4.** Crear una lista con el número de todos los proyectos en los que esté implicado un empleado cuyo primer apellido sea 'Pérez', sea un trabajador o sea director del departamento que controla el proyecto.

[10] En general, no es necesario que una tabla SQL tenga una clave, aunque en la mayoría de los casos contarán con una.

[11] En algunos sistemas, se utiliza la palabra clave MINUS para la diferencia de conjuntos, en lugar de EXCEPT.

```
C4:  ( SELECT   DISTINCT NumProyecto
     FROM       PROYECTO, DEPARTAMENTO, EMPLEADO
     WHERE      NumDptoProyecto=NumeroDpto AND DniDirector=Dni
                AND Apellido1='Pérez' )
     UNION
     ( SELECT   DISTINCT NumProyecto
     FROM       PROYECTO, TRABAJA_EN, EMPLEADO
     WHERE      NumProyecto=NumProy AND DniEmpleado=Dni
                AND Apellido1='Pérez' );
```

La primera consulta SELECT recupera los proyectos que implican a 'Pérez' como director del departamento encargado de controlar el proyecto, y la segunda recupera los proyectos en los que trabaja 'Pérez'. Si varios empleados tienen como primer apellido 'Pérez', se recuperará el nombre de los proyectos en los que cualquiera de ellos esté implicado. La aplicación de la operación UNION a las dos consultas SELECT proporciona el resultado deseado.

SQL también dispone de las operaciones multiconjunto correspondientes, que van seguidas por la palabra clave **ALL** (UNION ALL, EXCEPT ALL, INTERSECT ALL). Sus resultados son multiconjuntos (los duplicados no se eliminan). El comportamiento de estas operaciones se ilustra en los ejemplos de la Figura 8.5. Básicamente, cuando se aplican estas operaciones a cada tupla (sea duplicada o no), ésta se considera una tupla diferente.

## 8.4.5 Comparación de subcadenas y operadores aritméticos

En esta sección explicamos algunas características más de SQL. En primer lugar hablaremos de las condiciones de comparación de partes de una cadena de caracteres, mediante el operador de comparación **LIKE**. Esto se puede utilizar para la **comparación de patrones**. Las cadenas parciales se especifican mediante dos caracteres reservados: % sustituye una cantidad arbitraria de caracteres (de cero o más caracteres), y el guión de subrayado (_) reemplaza un solo carácter. Por ejemplo, considere la siguiente consulta.

**Consulta 12.** Recuperar todos los empleados cuya dirección se encuentra en Madrid.

```
C12: SELECT   Nombre, Apellido1
     FROM     EMPLEADO
     WHERE    Dirección LIKE '%Madrid%';
```

**Figura 8.5.** Resultado de las operaciones de multiconjunto de SQL. (a) Dos tablas, R(A) y S(A). (b) R(A) UNION ALL S(A). (c) R(A) EXCEPT ALL S(A). (d) R(A) INTERSECT ALL S(A).

**(a)** **R**

| A |
|---|
| a1 |
| a2 |
| a2 |
| a3 |

**S**

| A |
|---|
| a1 |
| a2 |
| a4 |
| a5 |

**(b)** **T**

| A |
|---|
| a1 |
| a1 |
| a2 |
| a2 |
| a2 |
| a3 |
| a4 |
| a5 |

**(c)** **T**

| A |
|---|
| a2 |
| a3 |

**(d)** **T**

| A |
|---|
| a2 |
| a3 |

Para recuperar todos los empleados que hayan nacido en la década de 1950 podemos utilizar la Consulta 12A. Aquí, el '5' debe ser el noveno carácter de la cadena (de acuerdo con nuestro formato de fecha), por lo que utilizamos el valor '_ _ _ _ _ _ _ _ 5 _', en el que cada guión de subrayado sirve como marcador de un carácter.

**Consulta 12A.** Encontrar todos los empleados que hayan nacido durante la década de 1950.

```
C12A: SELECT   Nombre, Apellido1
      FROM     EMPLEADO
      WHERE    FechaNac LIKE '_ _ _ _ _ _ _ _ 5 _';
```

Si hay necesidad de utilizar el carácter % o el guión bajo como carácter literal en la cadena, hay que precederlo por un *carácter de escape*, que se especifica después de la cadena mediante la palabra clave ESCAPE. Por ejemplo, 'AB\ _CD\ %EF' ESCAPE '\ ' representa la cadena literal 'AB_CD%EF' porque se ha especificado el carácter \ como carácter de escape. Como carácter de escape se puede utilizar cualquier carácter que no se utilice en la cadena. Además, necesitamos una regla para especificar los apóstrofes o comillas simples (' ') en caso de tener que incluirlas en una cadena porque se utilicen para empezar o terminar cadenas. Si necesitamos un apóstrofe ('), hay que representarlo como dos apóstrofes consecutivos (''), para que no sea interpretado como final de la cadena.

Otra característica permite utilizar la aritmética en las consultas. Los operadores aritméticos estándar para la suma (+), la diferencia (−), la multiplicación (∗) y la división (/) se pueden aplicar a valores o atributos numéricos con dominios numéricos. Por ejemplo, suponga que queremos ver el efecto de subir un 10% el sueldo de todos los empleados que trabajan en el proyecto 'ProductoX'; podemos ejecutar la Consulta 13 para ver sus nuevos salarios. Este ejemplo también muestra cómo podemos renombrar un atributo en el resultado de la consulta utilizando AS en la cláusula SELECT.

**Consulta 13.** Mostrar los salarios aumentados un 10% de todos los empleados que trabajan en el proyecto 'ProductoX'.

```
C13: SELECT   Nombre, Apellido1, 1.1∗Sueldo AS SueldoAumentado
     FROM     EMPLEADO, TRABAJA_EN, PROYECTO
     WHERE    Dni=DniEmpleado AND NumProy=NumProyecto AND
              NombreProyecto='ProductoX';
```

En las cadenas es posible utilizar el operador de concatenación || para añadir dos valores de cadena. En el caso de fechas, horas, marcas de tiempo e intervalos, los operadores suponen incrementar (+) o decrementar (-) una fecha, una hora o una marca de tiempo según un intervalo. Además, un valor de intervalo es el resultado de la diferencia entre dos valores de fecha, hora o marca de tiempo. Otro operador de comparación que se puede utilizar es **BETWEEN**, que se ilustra en la Consulta 14.

**Consulta 14.** Recuperar todos los empleados del departamento 5 cuyo salario esté entre 30.000 y 40.000.

```
C14: SELECT   ∗
     FROM     EMPLEADO
     WHERE    (Sueldo BETWEEN 30000 AND 40000) AND Dno = 5;
```

La condición (Sueldo **BETWEEN** 30000 **AND** 40000) de la C14 es equivalente a la condición ((Sueldo >= 30000) **AND** (Sueldo <= 40000)).

## 8.4.6 Ordenación del resultado de una consulta

SQL permite ordenar las tuplas del resultado de una consulta por los valores de uno o más atributos, utilizando la cláusula ORDER BY. Es lo que se ilustra en la Consulta 15.

**Consulta 15.** Recuperar una lista de empleados y de los proyectos en los que trabajan, ordenada por el departamento. Dentro de cada departamento, ordenar alfabéticamente los empleados por su primer apellido y su nombre.

```
C15: SELECT    NombreDpto, Apellido1, Nombre, NombreProyecto
     FROM      DEPARTAMENTO, EMPLEADO, TRABAJA_EN, PROYECTO
     WHERE     NumeroDpto=Dno AND Dni=DniEmpleado AND NumProy=NumProyecto
     ORDER BY  NombreDpto, Apellido1, Nombre;
```

El orden predeterminado es el ascendente. Con la palabra clave DESC podemos ver el resultado ordenado descendentemente. La palabra clave ASC permite especificar explícitamente el orden ascendente. Por ejemplo, si deseamos el orden descendente para NombreDpto y el orden ascendente para Apellido1, Nombre, la cláusula ORDER BY de C15 se puede escribir de este modo:

```
ORDER BY NombreDpto DESC, Apellido1 ASC, Nombre ASC
```

# 8.5 Consultas SQL más complejas

En la sección anterior describimos algunos tipos básicos de consultas SQL. Debido a la generalidad y la potencia expresiva del lenguaje, hay muchas otras características que permiten consultas más complejas. En esta sección explicaremos algunas de estas características.

## 8.5.1 Comparaciones con valores NULL y lógica de tres valores

SQL tiene varias reglas para tratar con los valores NULL. Si recuerda de la Sección 5.1.2, NULL se utiliza para representar la ausencia de un valor, aunque normalmente tiene una de tres interpretaciones diferentes: valor desconocido (existe, pero no se conoce), valor no disponible (existe, pero no se especifica a propósito), o atributo no aplicable (no definido para esta tupla). Considere los siguientes ejemplos para ilustrar cada uno de los significados de NULL.

1. **Valor desconocido**. Una persona en particular tiene una fecha de nacimiento, pero no la conocemos, por lo que la representamos con NULL en la base de datos.
2. **Valor no disponible o no especificado**. Una persona tiene un teléfono en casa, pero no quiere que aparezca listado, por lo que se impide su visualización y se representa como NULL en la base de datos.
3. **Atributo no aplicable**. Un atributo ÚltimoGrado sería NULL para una persona que no tiene una licenciatura, algo que no es aplicable para esa persona.

A veces no es posible determinar el significado que se pretende; por ejemplo, NULL para el teléfono de casa de una persona puede tener cualquiera de los tres significados. Por tanto, SQL no distingue entre los diferentes significados de NULL.

En general, cada NULL es considerado diferente a cualquier otro NULL de la base de datos. Cuando en una comparación se ve implicado un NULL, se considera que el resultado es UNKNOWN, o desconocido (podría ser TRUE o podría ser FALSE). Por tanto, SQL utiliza una lógica de tres valores con los valores TRUE, FALSE y UNKNOWN, en lugar de la lógica estándar de dos valores con TRUE o FALSE. Por consiguiente, es necesario definir los resultados de las expresiones lógicas de tres valores cuando se utilizan los conectores lógicos AND, OR y NOT. La Tabla 8.1 muestra los valores resultantes.

En las Tablas 8.1(a) y 8.1(b), las filas y las columnas representan los valores de los resultados de las expresiones booleanas (condiciones de comparación) de tres valores, que normalmente aparecen en la cláusula WHERE de una consulta SQL. Cada resultado de una expresión tendría un valor de TRUE, FALSE o UNKNOWN. La Tabla 8.1(a) muestra el resultado de combinar los dos mediante el conector lógico AND, y la

**Tabla 8.1.** Conexiones lógicas en la lógica de tres valores.

**(a)**

| **AND** | TRUE | FALSE | UNKNOWN |
|---|---|---|---|
| TRUE | TRUE | FALSE | UNKNOWN |
| FALSE | FALSE | FALSE | FALSE |
| UNKNOWN | UNKNOWN | FALSE | UNKNOWN |

**(b)**

| **OR** | TRUE | FALSE | UNKNOWN |
|---|---|---|---|
| TRUE | TRUE | TRUE | TRUE |
| FALSE | TRUE | FALSE | UNKNOWN |
| UNKNOWN | TRUE | UNKNOWN | UNKNOWN |

**(c)**

| **NOT** | |
|---|---|
| TRUE | FALSE |
| FALSE | TRUE |
| UNKNOWN | UNKNOWN |

Tabla 8.1(b) muestra el resultado de utilizar OR. Por jemplo, el resultado de (FALSE AND UNKNOWN) es FALSE, mientras que el resultado de (FALSE OR UNKNOWN) es UNKNOWN. La Tabla 8.1(c) muestra el resultado de la operación lógica NOT. En la lógica booleana estándar, sólo están permitidos los valores TRUE o FALSE; no existe el valor UNKNOWN.

En las consultas selección-proyección-concatenación, la regla general es que sólo se seleccionan las combinaciones de tuplas que evalúan la expresión lógica de la consulta como TRUE. Las combinaciones de tuplas que hacen una evaluación de FALSE o UNKNOWN no son seleccionadas. Sin embargo, hay excepciones a esta regla con ciertas operaciones, como las concatenaciones externas, como veremos.

SQL permite consultas que comprueban si el valor de un atributo es **NULL**. En lugar de utilizar = o <> para comparar el valor de un atributo con NULL, SQL utiliza **IS** o **IS NOT**. Esto es así porque SQL considera cada valor NULL distinto a cualquier otro valor NULL, por lo que la comparación de igualdad no es apropiada. Resulta que cuando se especifica una condición de concatenación, las tuplas con valores NULL en los atributos de concatenación no se incluyen en el resultado (a menos que se trate de una OUTER JOIN; consulte la Sección 8.5.6). La Consulta 18 ilustra esto, y su resultado se muestra en la Figura 8.4(d).

**Consulta 18.** Recuperar el nombre de todos los empleados que no tienen supervisores.

```
C18: SELECT  Nombre, Apellido1
     FROM    EMPLEADO
     WHERE   SuperDni IS NULL;
```

## 8.5.2 Consultas anidadas, tuplas y comparaciones conjunto/multiconjunto

Algunas consultas requieren obtener valores existentes en la base de datos para usarlos después en una condición de comparación. Dichas consultas se pueden formular convenientemente mediante **consultas anidadas**, que son bloques select-from-where completos dentro de la cláusula WHERE de otra consulta. Esa otra consulta es la que se conoce como **consulta externa**. La Consulta 4 está formulada en C4 sin consulta anida-

da, pero se puede reformar para que haga uso de consultas anidadas, como se muestra en C4A, donde se introduce el operador de comparación **IN**, que compara un valor *v* con un conjunto (o multiconjunto) de valores *V* y se evalúa como **TRUE** si *v* es uno de los elementos de *V*.

```
C4A:  SELECT    DISTINCT NumProyecto
      FROM      PROYECTO
      WHERE     NumProyecto IN
      ( SELECT  NumProyecto
      FROM      PROYECTO, DEPARTAMENTO, EMPLEADO
      WHERE     NumDptoProyecto=NumeroDpto AND
                DniDirector=Dni AND Apellido1='Pérez' )
OR
      NumProyecto IN
      ( SELECT  NumProy
      FROM      TRABAJA_EN, EMPLEADO
      WHERE     DniEmpleado=Dni AND Apellido1='Pérez' );
```

La primera consulta anidada selecciona los números de proyecto de los proyectos que tienen a 'Pérez' como director, mientras que la segunda selecciona los números de proyecto de los proyectos que tienen a 'Pérez' como trabajador. En la consulta externa, utilizamos **OR** para recuperar una tupla PROYECTO si el valor NUMPROYECTO de esa tupla se encuentra en el resultado de cualquier consulta anidada.

Si una consulta anidada devuelve un solo atributo *y* una sola tupla, el resultado de la consulta será un solo valor (escalar). En estos casos, está permitido utilizar = en lugar de IN como operador de comparación. En general, la consulta anidada devolverá una **tabla** (relación), que es un conjunto o un multiconjunto de tuplas.

SQL permite el uso de **tuplas** de valores en las comparaciones colocándolas entre paréntesis. Observe esta consulta:

```
SELECT    DISTINCT DniEmpleado
FROM      TRABAJA_EN
WHERE     (NumProy, Horas) IN ( SELECT NumProy, Horas
FROM      TRABAJA_EN
WHERE     Dni='123456789' );
```

Esta consulta seleccionará los DNIs de todos los empleados que trabajan la misma combinación (proyecto, horas) en alguno de los proyectos en los que trabaja el empleado 'José Pérez' (cuyo Dni='123456789'). En este ejemplo, el operador IN compara la subtupla de valores entre paréntesis (NumProy, Horas) de cada tupla TRABAJA_EN con el conjunto de tuplas compatibles con la unión producidas por la consulta anidada.

Además del operador IN, es possible utilizar otros operadores de comparación para comparar un solo valor *v* (normalmente el nombre de un atributo) con un conjunto o multiconjunto *V* (normalmente, una consulta anidada). El operador = ANY (o = SOME) devuelve TRUE si el valor *v* es igual a *algún valor* del conjunto *V* y, por tanto, es equivalente a IN. Las palabras clave ANY y SOME tienen el mismo significado. Otros operadores que se pueden combinar con ANY (o SOME) son >, >=, <, <= y <>. La palabra clave ALL también se puede combinar con cada uno de estos operadores. Por ejemplo, la condicón de comparación (*v* > ALL *V*) devuelve TRUE si el valor *v* es mayor que *todos* los valores del conjunto (o multiconjunto) *V*. En la siguiente consulta tiene un ejemplo, que devuelve el nombre de los empleados cuyo salario es mayor que el salario de todos los empleados del departamento 5:

```
SELECT    Apellido1, Nombre
FROM      EMPLEADO
WHERE     Sueldo > ALL ( SELECT      Sueldo
```

```
        FROM    EMPLEADO
        WHERE   Dno=5 );
```

En general, podemos tener varios niveles de consultas anidadas. Una vez más nos podemos enfrentar a una posible ambigüedad con el nombre de los atributos si existen varios atributos con el mismo nombre: uno en una relación de la cláusula FROM de la *consulta externa,* y otro en una relación de la cláusula FROM de la *consulta anidada*. La norma es que una referencia a un *atributo sin calificar* se refiere a la relación declarada en la **consulta anidada más interna**. Por ejemplo, en las cláusulas SELECT y WHERE de la primera consulta anidada de C4A, una referencia a cualquier atributo no calificado de la relación PROYECTO se refiere a la relación PROYECTO especificada en la cláusula FROM de la consulta anidada. Para referirse a un atributo de la relación PROYECTO especificada en la consulta externa, podemos especificar y referirnos a un *alias* (variable de tupla) para esa relación. Estas reglas son parecidas a las reglas de ámbito de las variables de programación en la mayoría de los lenguajes de programación que permiten procedimientos y funciones anidados. Para ilustrar el potencial de ambigüedad en los nombres de los atributos en las consultas anidadas, veamos la Consulta 16, cuyo resultado se muestra en la Figura 8.4(c).

**Consulta 16.** Recuperar el nombre de cada empleado que tiene un subordinado con el mismo nombre de pila y el mismo sexo que dicho empleado.

```
C16:  SELECT   E.Nombre, E.Apellido1
      FROM     EMPLEADO AS E
      WHERE    E.Dni IN ( SELECT     DniEmpleado
      FROM     SUBORDINADO
      WHERE    E.Nombre=NombSubordinado
      AND E.Sexo=Sexo );
```

En la consulta anidada de C16 debemos calificar E.Sexo porque se refiere al atributo Sexo de EMPLEADO de la consulta externa, y SUBORDINADO también tiene un atributo denominado Sexo. Todas las referencias a Sexo sin calificar de la consulta anidada se refieren a Sexo de SUBORDINADO. No obstante, no tenemos que calificar Nombre y Dni porque la relación SUBORDINADO no tiene unos atributos denominados Nombre y Dni, por lo que no hay ambigüedad.

Es generalmente aconsejable crear variables de tupla (alias) para *todas las tablas referenciadas en una consulta SQL* a fin de evitar errores y ambigüedades potenciales.

### 8.5.3 Consultas anidadas correlacionadas

Siempre que una condición de la cláusula WHERE de una consulta anidada se refiera a algún atributo de una relación declarada en la consulta exterior, se dice que las dos consultas son **correlacionadas**. Podemos entender mejor una consulta correlacionada teniendo en cuenta que la *consulta anidada se evalúa una vez por cada tupla (o combinación de tuplas) en la consulta exterior*. Por ejemplo, podemos imaginar C16 de este modo: por *cada* tupla EMPLEADO, evaluar la consulta anidada, que recupera los valores DniEmpleado para todas las tuplas SUBORDINADO que tienen el mismo sexo y nombre que la tupla de EMPLEADO; si el valor de Dni de la tupla EMPLEADO se encuentra *en* el resultado de la consulta anidada, entonces seleccionar esa tupla EMPLEADO.

En general, una consulta escrita con bloques select-from-where anidados y utilizando los operadores de comparación = o IN *siempre* se expresa como una consulta de un solo bloque. Por ejemplo, C16 se puede escribir como en C16A:

```
C16A:  SELECT   E.Nombre, E.Apellido1
       FROM     EMPLEADO AS E, SUBORDINADO AS D
       WHERE    E.Dni=D.DniEmpleado AND E.Sexo=D.Sexo
                AND E.Nombre=D.NombSubordinado;
```

La implementación SQL original en SYSTEM R también tenía un operador de comparación **CONTAINS**, que se utilizaba para comparar dos conjuntos o multiconjuntos. Este operador se fue eliminando progresivamente del lenguaje, posiblemente debido a la complejidad de implementarlo eficazmente. La mayoría de las implementaciones comerciales de SQL *no tienen* este operador. El operador CONTAINS compara dos conjuntos de valores y devuelve TRUE si un conjunto contiene todos los valores del otro conjunto. La Consulta 3 ilustra el uso de este operador.

**Consulta 3.** Recuperar el nombre de los empleados que trabajan en *todos* los proyectos controlados por el departamento número 5.

```
C3:  SELECT   Nombre, Apellido1
     FROM     EMPLEADO
     WHERE    ( ( SELECT     NumProy
                  FROM       TRABAJA_EN
                  WHERE      Dni=DniEmpleado )
                CONTAINS
                ( SELECT     NumProyecto
                  FROM       PROYECTO
                  WHERE      NumDptoProyecto=5 ) );
```

En C3, la segunda consulta anidada (que no es correlacionada con la consulta exterior) recupera los números de proyecto de todos los proyectos controlados por el departamento 5. Por *cada* tupla de empleado, la primera consulta anidada (que es correlacionada) recupera los números de proyecto en los que trabaja el empleado; si esta consulta contiene todos los proyectos controlados por el departamento 5, la tupla empleado se selecciona y se recupera el nombre de ese empleado. El operador de comparación CONTAINS tiene una función parecida a la operación DIVISION del álgebra relacional (consulte la Sección 6.3.4) y a la cuantificación universal del cálculo relacional (consulte la Sección 6.6.7). Como la operación CONTAINS no forma parte de SQL, tenemos que utilizar otras técnicas, como la función EXISTS, para especificar estos tipos de consultas, como se describe en la Sección 8.5.4.

## 8.5.4 Las funciones EXISTS y UNIQUE en SQL

La función EXISTS de SQL se utiliza para comprobar si el resultado de una consulta anidada correlacionada está vacío (no contiene tuplas) o no. El resultado de EXISTS es un valor booleano, **TRUE** o **FALSE**. Vamos a ilustrar el uso de EXISTS (y de NOT EXISTS*)* con algunos ejemplos. En primer lugar, vamos a formular la Consulta 16 de una forma alternativa utilizando EXISTS:

```
C16B:  SELECT   E.Nombre, E.Apellido1
       FROM     EMPLEADO AS E
       WHERE    EXISTS ( SELECT      *
                         FROM        SUBORDINADO
                         WHERE       E.Dni=DniEmpleado AND E.Sexo=Sexo
                                     AND E.Nombre=NombSubordinado );
```

EXISTS y NOT EXISTS normalmente se utilizan en combinación con una consulta anidada correlacionada. En C16B, la consulta anidada hace referencia a los atributos Dni, Nombre y Sexo de la relación EMPLEADO de la consulta externa. Podemos pensar en C16B de este modo: por cada tupla EMPLEADO, se evalúa la consulta anidada, que recupera todas las tuplas SUBORDINADO con los mismos valores de DniEmpleado, Sexo y NombSubordinado que la tupla EMPLEADO; si en el resultado de la consulta anidada existe (EXISTS) al menos una tupla, entonces se selecciona esa tupla EMPLEADO. En general, EXISTS(Q) devuelve **TRUE** si hay *al menos una tupla* en el resultado de la consulta anidada Q, y devuelve **FALSE** en caso contrario. Por otro

lado, NOT EXISTS(Q) devuelve **TRUE** si *no hay tuplas* en el resultado de la consulta anidada Q, y devuelve **FALSE** en caso contrario. A continuación, ilustramos el uso de NOT EXISTS.

**Consulta 6.** Recuperar el nombre de los empleados que no tienen subordinados.

```
C6: SELECT Nombre, Apellido1
    FROM   EMPLEADO
    WHERE  NOT EXISTS ( SELECT *
                        FROM   SUBORDINADO
                        WHERE  Dni=DniEmpleado );
```

En C6, la consulta anidada correlacionada recupera todas las tuplas SUBORDINADO relacionadas con una tupla EMPLEADO en particular. Si *no existe ninguna,* la tupla EMPLEADO se selecciona. Podemos explicar C6 de este modo: por *cada* tupla EMPLEADO, la consulta anidada correlacionada selecciona todas las tuplas SUBORDINADO cuyo valor de DniEmpleado coincida con el Dni de EMPLEADO; si el resultado es vacío, no hay subordinados relacionados con el empleado, por lo que seleccionamos dicha tupla EMPLEADO y recuperamos su Nombre y su Apellido1.

**Consulta 7.** Listar los nombres de los directores que tienen al menos un subordinado.

```
C7: SELECT Nombre, Apellido1
    FROM   EMPLEADO
    WHERE  EXISTS ( SELECT *
                    FROM   SUBORDINADO
                    WHERE  Dni=DniEmpleado )
           AND
           EXISTS ( SELECT *
                    FROM   DEPARTAMENTO
                    WHERE  Dni=DniDirector );
```

Una forma de escribir esta consulta es como se muestra en C7, donde utilizamos dos consultas correlacionadas anidadas; la primera selecciona todas las tuplas SUBORDINADO relacionadas con un EMPLEADO, y la segunda selecciona todas las tuplas DEPARTAMENTO administradas por el EMPLEADO. Si existe al menos una de las primeras y al menos una de las segundas, seleccionamos la tupla EMPLEADO. ¿Puede reescribir esta consulta con una sola consulta anidada o sin consultas anidadas?

La C3 (*Recuperar el nombre de los empleados que trabajan en todos los proyectos controlados por el departamento número 5*: consulte la Sección 8.5.3) se puede enunciar mediante EXISTS y NOT EXISTS en los sistemas SQL. Hay dos opciones. La primera es utilizar la transformación de la teoría de conjuntos (*S1* CONTAINS *S2*) que es lógicamente equivalente a (*S2* EXCEPT *S1*) y que es vacío.[12] Esta opción se muestra en C3A.

```
C3A: SELECT Nombre, Apellido1
     FROM   EMPLEADO
     WHERE  NOT EXISTS ( ( SELECT   NumProyecto
                           FROM     PROYECTO
                           WHERE    NumDptoProyecto=5)
                           EXCEPT ( SELECT   NumProy
                                    FROM     TRABAJA_EN
                                    WHERE    Dni=DniEmpleado) );
```

[12] Recuerde que EXCEPT es el operador de diferencia de conjuntos. La palabra clave MINUS también se utiliza a veces, por ejemplo en Oracle.

En C3A, la primera subconsulta (que no es correlacionada) selecciona todos los proyectos controlados por el departamento 5, y la segunda subconsulta (que es correlacionada) selecciona todos los proyectos en los que un empleado en particular se considera que trabaja. Si la diferencia de conjuntos de la primera subconsulta menos (MINUS [EXCEPT]) la segunda subconsulta da como resultado vacío, significa que el empleado trabaja en todos los proyectos y, por tanto, se selecciona.

La segunda opción se muestra en la C3B: necesitamos un anidamiento de dos niveles. Esta formulación es un poco más compleja que C3, donde utilizamos el operador de comparación CONTAINS, y que C3A, donde utilizamos NOT EXISTS y EXCEPT. No obstante, CONTAINS no forma parte de SQL, y no todos los sistemas relacionales cuentan con el operador EXCEPT, aunque éste forme parte de SQL-99.

```
C3B: SELECT   Apellido1, Nombre
     FROM     EMPLEADO
     WHERE    NOT EXISTS ( SELECT  *
                           FROM    TRABAJA_EN B
                           WHERE   ( B.NumProy IN ( SELECT    NumProyecto
                                                    FROM      PROYECTO
                                                    WHERE     NumDptoProyecto=5 )
                           AND
                           NOT EXISTS ( SELECT    *
                                        FROM      TRABAJA_EN C
                                        WHERE     C.DniEmpleado=Dni
                                        AND       C.NumProy=B.NumProy ) );
```

En C3B, la consulta anidada exterior selecciona cualquier tupla TRABAJA_EN (B) cuyo NumProy corresponda a un proyecto controlado por el departamento 5, *si* no hay una tupla TRABAJA_EN (C) con el mismo NumProy y el mismo Dni que la tupla EMPLEADO bajo consideración de la consulta exterior. Si no existe dicha tupla, seleccionamos la tupla EMPLEADO. La forma de C3B coincide con la Consulta 3.

Hay otra function SQL, UNIQUE(Q), que devuelve TRUE si no hay tuplas duplicadas en el resultado de la consulta Q; en caso contrario, devuelve FALSE. Esto se puede utilizar para probar si el resultado de una consulta anidada es un conjunto o un multiconjunto.

### 8.5.5 Conjuntos explícitos y renombrado de atributos en SQL

Hemos visto varias consultas con una consulta anidada en la cláusula WHERE. También es posible utilizar un **conjunto explícito de valores** en dicha cláusula, en lugar de una consulta anidada. El conjunto debe ir entre paréntesis.

**Consulta 17.** Recuperar los números del documento nacional de identidad de todos los empleados que trabajan en los proyectos 1, 2 ó 3.

```
C17: SELECT   DISTINCT DniEmpleado
     FROM     TRABAJA_EN
     WHERE    NumProy IN (1, 2, 3);
```

En SQL, es posible renombrar cualquier atributo que aparezca en el resultado de una consulta añadiendo el calificador **AS** seguido por el nombre nuevo deseado. Por tanto, la estructura AS se puede utilizar para asignar alias a los nombres de atributos y relaciones, y tanto en la cláusula SELECT como en la cláusula FROM. Por ejemplo, C8A muestra unos ligeros cambios respecto a C8 para recuperar el primer apellido de los empleados y sus supervisores, a la vez que se renombran los nombres de atributo resultantes como NombreDeEmpleado y NombreDeSupervisor. Los nombres nuevos aparecerán como cabeceras de columna en el resultado de la consulta.

```
C8A: SELECT   E.Apellido1 AS NombreDeEmpleado, S.Apellido1 AS NombreDeSupervisor
     FROM     EMPLEADO AS E, EMPLEADO AS S
     WHERE    E.SuperDni=S.Dni;
```

### 8.5.6 Tablas concatenadas en SQL y concatenaciones exteriores

El concepto de **tabla concatenada** (o **relación concatenada**) se incorporó a SQL para poder especificar una tabla como resultado de una operación de concatenación *en la cláusula from* de una consulta. Esta estructura es más fácil que mezclar todas las condiciones de selección y concatenación en la cláusula WHERE. Por ejemplo, considere la consulta C1, que recupera el nombre y la dirección de los empleados que trabajan para el departamento 'Investigación'. Puede ser más fácil especificar primero la concatenación de las relaciones EMPLEADO y DEPARTAMENTO, y después seleccionar las tuplas y los atributos deseados. Esto se puede escribir en SQL como en C1A:

```
C1A: SELECT   Nombre, Apellido1, Dirección
     FROM     (EMPLEADO JOIN DEPARTAMENTO ON Dno=NumeroDpto)
     WHERE    NombreDpto='Investigación';
```

La cláusula FROM de C1A contiene una sola *tabla concatenada*. Los atributos de dicha tabla son todos los atributos de la primera tabla, EMPLEADO, seguidos por todos los atributos de la segunda tabla, DEPARTAMENTO. El concepto de tabla concatenada también permite especificar tipos diferentes de concatenación, como NATURAL JOIN y varios tipos de OUTER JOIN. En una concatenación natural (NATURAL JOIN) sobre las relaciones *R* y *S*, no se especifica condición de concatenación alguna; se crea una condición EQUIJOIN implícita para *cada par de atributos con el mismo nombre* que en *R* y *S*. Cada par de atributos se incluye una sola vez en la relación resultante (consulte la Sección 6.3.2).

Si los nombres de los atributos de concatenación no coinciden con los de las relaciones base, es posible renombrarlos para que coincidan, y después aplicar NATURAL JOIN. En este caso, se puede utilizar la estructura AS para renombrar una relación y todos sus atributos en la cláusula FROM. Es lo que se ilustra en la C1B, donde la relación DEPARTAMENTO se renombra como DEPT y sus atributos se renombran como NombreDpto, Dno (para que coincida con el nombre del atributo de concatenación deseado Dno de EMPLEADO), DniDelDirector y FechaInDirector. La condición de concatenación implicada en esta NATURAL JOIN es EMPLEADO.Dno=DEPT.Dno porque es el único par de atributos con el mismo nombre después de haber renombrado:

```
C1B: SELECT   Nombre, Apellido1, Dirección
     FROM     (EMPLEADO NATURAL JOIN
     (DEPARTAMENTO AS DEPT (NombreDpto, Dno, DniDelDirector, FechaInDirector)))
     WHERE    NombreDpto='Investigación';
```

El tipo predeterminado de concatenación en una tabla concatenada es una **concatenación interna**, en la que una tupla se incluye en el resultado si en la otra relación existe una tupla coincidente. Por ejemplo, en la consulta C8A, en el resultado sólo se incluyen los empleados que *tienen un supervisor*; se excluyen las tuplas EMPLEADO cuyo valor para SuperDni es NULL. Si se necesita que se incluyan todos los empleados, debe utilizarse explícitamente una OUTER JOIN (consulte la Sección 6.4.4 si desea una definición de OUTER JOIN). En SQL, esto se manipula especificando explícitamente la OUTER JOIN en una tabla concatenada, como se ilustra en la consulta C8B:

```
C8B: SELECT   E.Apellido1 AS NombreDeEmpleado,
             S.Apellido1 AS NombreDeSupervisor
     FROM     (EMPLEADO AS E LEFT OUTER JOIN EMPLEADO AS S
             ON E.SuperDni=S.Dni);
```

Las opciones disponibles para especificar las tablas concatenadas en SQL son INNER JOIN (igual que JOIN), LEFT OUTER JOIN, RIGHT OUTER JOIN y FULL OUTER JOIN. En las tres últimas opciones puede omitirse la palabra clave OUTER. Si los atributos de concatenación tienen el mismo nombre, también se puede especificar la variación de concatenación natural de las concatenaciones externas utilizando la palabra clave NATURAL antes que la operación (por ejemplo, NATURAL LEFT OUTER JOIN). La palabra clave CROSS JOIN se utiliza para especificar la operación PRODUCTO CARTESIANO (consulte la Sección 6.2.2), aunque esto sólo debe usarse con sumo cuidado, porque genera todas las posibles combinaciones de tuplas.

También se pueden *anidar* especificaciones de concatenación; es decir, una de las tablas de una concatenación puede ser una tabla concatenada. Es lo que se ilustra en la consulta C2A, que es una forma distinta de especificar la consulta C2, utilizando el concepto de una tabla concatenada:

```
C2A:  SELECT   NumProyecto, NumDptoProyecto, Apellido1, Dirección, FechaNac
      FROM     ((PROYECTO JOIN DEPARTAMENTO ON NumDptoProyecto=NúmeroDpto)
      JOIN     EMPLEADO ON DniDirector=Dni)
      WHERE    UbicaciónProyecto='Gijón';
```

No todas las implementaciones de SQL han implementado la sintaxis nueva de las tablas concatenadas. En algunos sistemas, se utiliza una sintaxis diferente para las concatenaciones externas, utilizando unos operadores de comparación diferentes (+=, =+ y +=+ para la concatenación externa izquierda, derecha y completa, respectivamente) al especificar la condición de concatenación. Por ejemplo, es la sintaxis que se utiliza en Oracle. Para especificar la concatenación externa izquierda de la consulta C8B utilizando esta sintaxis, podríamos escribir la consulta C8C de este modo:

```
C8C:  SELECT   E.Apellido1, S.Apellido1
      FROM     EMPLEADO E, EMPLEADO S
      WHERE    E.SuperDni += S.Dni;
```

## 8.5.7 Funciones agregadas en SQL

En la Sección 6.4.2, hablamos del concepto de función agregada como una operación relacional. Como en muchas aplicaciones de bases de datos se necesitan el agrupamiento y la agregación, SQL dispone de funciones que incorporan estos conceptos: **COUNT**, **SUM**, **MAX**, **MIN** y **AVG**.[13] La función COUNT devuelve el número de tuplas o valores especificados en una consulta. Las funciones SUM, MAX, MIN y AVG se aplican a un conjunto o multiconjunto de valores numéricos y devuelven, respectivamente, la suma, el valor máximo, el valor mínimo y el promedio de esos valores. Estas funciones se pueden utilizar en la cláusula SELECT o en una cláusula HAVING (de la que hablaremos más tarde). Las funciones MAX y MIN también se pueden utilizar con atributos que tienen dominios no numéricos si los valores del dominio tienen una *ordenación total* entre sí.[14] Ilustramos el uso de estas funciones con algunos ejemplos de consultas.

**Consulta 19.** Visualizar la suma de los salarios de todos los empleados, el salario más alto, el salario más bajo y el sueldo medio.

```
C19:  SELECT   SUM (Sueldo), MAX (Sueldo), MIN (Sueldo), AVG (Sueldo)
      FROM     EMPLEADO;
```

Si queremos obtener los valores de función anteriores para los empleados de un departamento específico (por ejemplo, el departamento 'Investigación') podemos escribir la Consulta 20, donde las tuplas EMPLEADO

[13] En SQL-99 se han añadido funciones agregadas adicionales para poder realizar cálculos estadísticos más avanzados.

[14] La ordenación total se refiere a que para dos valores cualesquiera del dominio, puede determinarse que uno aparece antes que el otro en el orden definido; por ejemplo, los dominios DATE, TIME y TIMESTAMP tienen ordenaciones totales en sus valores, como ocurre con las cadenas alfanuméricas.

están restringidas por la cláusula WHERE a los empleados que trabajan para el departamento 'Investigación'.

**Consulta 20.** Visualizar la suma de los salarios de todos los empleados del departamento 'Investigación', así como el salario más alto, el salario más bajo y el salario medio de este departamento.

```
C20:  SELECT   SUM (Sueldo), MAX (Sueldo), MIN (Sueldo), AVG (Sueldo)
      FROM     (EMPLEADO JOIN DEPARTAMENTO ON Dno=NumeroDpto)
      WHERE    NombreDpto='Investigación';
```

**Consultas 21 y 22.** Recuperar el número total de empleados de la empresa (C21) y el número de empleados del departamento 'Investigación' (C22).

```
C21:  SELECT   COUNT (*)
      FROM     EMPLEADO;

C22:  SELECT   COUNT (*)
      FROM     EMPLEADO, DEPARTAMENTO
      WHERE    Dno=NumeroDpto AND NombreDpto='Investigación';
```

El asterisco (*) se refiere a las *filas* (tuplas), por lo que COUNT (*) devuelve el número de filas del resultado de la consulta. También podemos utilizar la función COUNT para contar los valores de una columna en lugar de las tuplas, como en el siguiente ejemplo.

**Consulta 23.** Contar el número de sueldos diferentes almacenados en la base de datos.

```
C23:  SELECT   COUNT (DISTINCT Sueldo)
      FROM     EMPLEADO;
```

Si escribimos COUNT(SUELDO) en lugar de COUNT(DISTINCT SUELDO) en C23, los valores duplicados no se eliminarán. No obstante, no se contabilizarán las tuplas cuyo SUELDO es NULL. En general, los valores NULL se **descartan** cuando se aplican las funciones agregadas a una columna (atributo) en particular.

Los ejemplos anteriores extraen una *relación entera* (C19, C21, C23) o un subconjunto seleccionado de tuplas (C20, C22) y, por tanto, todas producen tuplas sencillas o valores sencillos. Ilustran cómo se aplican las funciones para recuperar un valor o una tupla de resumen de la base de datos. Estas funciones también se utilizan en las condiciones de selección que implican consultas anidadas. Podemos especificar una consulta anidada correlacionada con una función agregada, y después utilizar la consulta anidada en la cláusula WHERE de una consulta externa. Por ejemplo, para recuperar los nombres de todos los empleados que tienen dos o más subordinados (Consulta 5), podemos escribir lo siguiente:

```
C5:   SELECT   Apellido1, Nombre
      FROM     EMPLEADO
      WHERE    (SELECT     COUNT (*)
                FROM       SUBORDINADO
                WHERE      Dni=DniEmpleado ) >= 2;
```

La consulta anidada correlacionada contabiliza el número de subordinados que cada empleado tiene; si esta cantidad es mayor o igual que 2, se selecciona la tupla de empleado.

## 8.5.8 Agrupamiento: las cláusulas GROUP BY y HAVING

En muchos casos queremos aplicar las funciones agregadas a *subgrupos de tuplas de una relación*, estando los subgrupos basados en algunos valores de atributo. Por ejemplo, vamos a suponer que queremos saber el

sueldo medio de los empleados de *cada departamento* o el número de empleados que *trabajan en cada proyecto*. En estos casos, tenemos que **dividir** la relación en subconjuntos no solapados (o **grupos**) de tuplas. Cada grupo (partición) estará compuesto por las tuplas que tienen el mismo valor para algún(os) atributo(s), denominado(s) **atributo(s) de agrupamiento**. Después podemos aplicar la función independientemente a cada grupo. SQL tiene una cláusula **GROUP BY** para este propósito. Esta cláusula especifica los atributos de agrupamiento, que también *deben aparecer en la cláusula SELECT*, por lo que el valor resultante de aplicar la función de agregación a un grupo de tuplas aparece junto con el valor de los atributos de agrupamiento.

**Consulta 24.** Por cada departamento, recuperar el número de departamento, el número de empleados del mismo y el sueldo medio.

```
C24: SELECT   Dno, COUNT (*), AVG (Sueldo)
     FROM     EMPLEADO
     GROUP BY Dno;
```

En C24, las tuplas EMPLEADO se dividen en grupos: cada uno tiene el mismo valor para el atributo de agrupamiento Dno. Las funciones COUNT y AVG se aplican a cada uno de dichos grupos de tuplas. La cláusula SELECT sólo incluye el atributo de agrupamiento y las funciones que se aplican a cada grupo de tuplas. La Figura 8.6(a) ilustra el funcionamiento del agrupamiento en C24; también muestra el resultado de C24.

Si existen NULLs en el atributo de agrupamiento, se crea un **grupo separado** para todas las tuplas con un *valor NULL en el atributo de agrupamiento*. Por ejemplo, si la tabla EMPLEADO tiene a NULL como el atributo de agrupamiento Dno de algunas tuplas, habrá un grupo separado para esas tuplas en el resultado de C24.

**Consulta 25.** Por cada proyecto, recuperar el número de proyecto, el nombre del proyecto y el número de empleados que trabajan en ese proyecto.

```
C25: SELECT   NumProyecto, NombreProyecto, COUNT (*)
     FROM     PROYECTO, TRABAJA_EN
     WHERE    NumProyecto=NumProy
     GROUP BY NumProyecto, NombreProyecto;
```

C25 muestra cómo podemos utilizar una condición de concatenación en combinación con GROUP BY. En este caso, el agrupamiento y las funciones se aplican *después* de la concatenación de las dos relaciones. A veces necesitamos recuperar los valores de esas funciones sólo para aquellos grupos que *satisfacen ciertas condiciones*. Por ejemplo, suponga que queremos modificar la Consulta 25 para que en el resultado sólo aparezcan los proyectos con más de dos empleados. SQL proporciona una cláusula **HAVING**, que puede aparecer en combinación con una cláusula GROUP BY, con este propósito. HAVING proporciona una condición en el grupo de tuplas asociado a cada valor de los atributos de agrupamiento. En el resultado de la consulta sólo aparecen los grupos que satisfacen la condición (véase la Consulta 26).

**Consulta 26.** Por cada proyecto *en el que trabajan más de dos empleados,* recuperar el número, el nombre y la cantidad de empleados que trabajan en él.

```
C26: SELECT   NumProyecto, NombreProyecto, COUNT (*)
     FROM     PROYECTO, TRABAJA_EN
     WHERE    NumProyecto=NumProy
     GROUP BY NumProyecto, NombreProyecto
     HAVING   COUNT (*) > 2;
```

Observe que mientras que las condiciones de selección de la cláusula WHERE limitan las *tuplas* a las que se aplican las funciones, la cláusula HAVING sirve para elegir *grupos enteros*. La Figura 8.6(b) ilustra el uso de HAVING y visualiza el resultado de C26.

**Consulta 27.** Por cada proyecto, recuperar el número, el nombre y la cantidad de empleados del departamento 5 que trabajan en dicho proyecto.

**Figura 8.6.** Resultado de GROUP BY y HAVING. (a) C24. (b) C26.

**(a)**

| Nombre | Apellido1 | Apellido2 | Dni | . . . | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|
| José | Pérez | Pérez | 123456789 | | 30000 | 333445555 | 5 |
| Alberto | Campos | Sastre | 333445555 | | 40000 | 888665555 | 5 |
| Fernando | Ojeda | Ordóñez | 666884444 | | 38000 | 333445555 | 5 |
| Aurora | Oliva | Avezuela | 453453453 | . . . | 25000 | 333445555 | 5 |
| Alicia | Jiménez | Celaya | 999887777 | | 25000 | 987654321 | 4 |
| Juana | Sainz | Oreja | 987654321 | | 43000 | 888665555 | 4 |
| Luis | Pajares | Morera | 987987987 | | 25000 | 987654321 | 4 |
| Eduardo | Bong | Paredes | 888665555 | | 55000 | NULL | 1 |

Agrupamiento de tuplas EMPLEADO por el valor de Dno.

| Dno | Count (*) | Avg (Sueldo) |
|---|---|---|
| 5 | 4 | 33250 |
| 4 | 3 | 31000 |
| 1 | 1 | 55000 |

Resultado de C24.

**(b)**

| NombreProyecto | NumProyecto | . . . | DniEmpleado | NumProy | Horas |
|---|---|---|---|---|---|
| ProductoX | 1 | | 123456789 | 1 | 32.5 |
| ProductoX | 1 | | 453453453 | 1 | 20.0 |
| ProductoY | 2 | | 123456789 | 2 | 7.5 |
| ProductoZ | 2 | | 453453453 | 2 | 20.0 |
| ProductoY | 2 | | 333445555 | 2 | 10.0 |
| ProductoY | 3 | | 666884444 | 3 | 40.0 |
| ProductoZ | 3 | | 333445555 | 3 | 10.0 |
| Computación | 10 | | 333445555 | 10 | 10.0 |
| Computación | 10 | . . . | 999887777 | 10 | 10.0 |
| Computación | 10 | | 987987987 | 10 | 35.0 |
| Reorganización | 20 | | 333445555 | 20 | 10.0 |
| Reorganización | 20 | | 987654321 | 20 | 15.0 |
| Reorganización | 20 | | 888665555 | 20 | NULL |
| Comunicaciones | 30 | | 987987987 | 30 | 5.0 |
| Comunicaciones | 30 | | 987654321 | 30 | 20.0 |
| Comunicaciones | 30 | | 999887777 | 30 | 30.0 |

Estos grupos no son seleccionados por la condición HAVING de C26.

Después de aplicar la cláusula WHERE pero antes de aplicar HAVING.

| NombreProyecto | NumProyecto | . . . | DniEmpleado | NumProy | Horas |
|---|---|---|---|---|---|
| ProductoY | 2 | | 123456789 | 2 | 7.5 |
| ProductoY | 2 | | 453453453 | 2 | 20.0 |
| ProductoY | 2 | | 333445555 | 2 | 10.0 |
| Computación | 10 | | 333445555 | 10 | 10.0 |
| Computación | 10 | | 999887777 | 10 | 10.0 |
| Computación | 10 | | 987987987 | 10 | 35.0 |
| Reorganización | 20 | . . . | 333445555 | 20 | 10.0 |
| Reorganización | 20 | | 987654321 | 20 | 15.0 |
| Reorganización | 20 | | 888665555 | 20 | NULL |
| Comunicaciones | 30 | | 987987987 | 30 | 5.0 |
| Comunicaciones | 30 | | 987654321 | 30 | 20.0 |
| Comunicaciones | 30 | | 999887777 | 30 | 30.0 |

| NombreProyecto | Count (*) |
|---|---|
| ProductoY | 3 |
| Computación | 3 |
| Reorganización | 3 |
| Comunicaciones | 3 |

Resultado de C26
(No se muestra NumProyecto)

Después de aplicar la condición de la cláusula HAVING.

```
C27: SELECT    NumProyecto, NombreProyecto, COUNT (*)
     FROM      PROYECTO, TRABAJA_EN, EMPLEADO
     WHERE     NumProyecto=NumProy AND Dni=DniEmpleado AND Dno=5
     GROUP BY NumProyecto, NombreProyecto;
```

Aquí restringimos las tuplas de la relación (y, por tanto, las tuplas de cada grupo) a las que satisfacen la condición especificada en la cláusula WHERE (a saber, que trabajen en el departamento 5). Tenemos que ser aún más cuidadosos cuando se aplican dos condiciones diferentes (una a la función de la cláusula SELECT y otra a la función de la cláusula HAVING). Por ejemplo, suponga que queremos contar el número *total* de empleados de cada departamento cuyo sueldo es superior a 40.000, pero sólo en aquellos departamentos en los que trabajan más de cinco empleados. La condición (SUELDO>40000) sólo se aplica a la función COUNT de la cláusula SELECT. Suponga que escribimos la siguiente consulta *incorrecta*:

```
SELECT    NombreDpto, COUNT (*)
FROM      DEPARTAMENTO, EMPLEADO
WHERE     NumeroDpto=Dno AND Sueldo>40000
GROUP BY  NombreDpto
HAVING    COUNT (*) > 5;
```

Esto es incorrecto porque seleccionará únicamente los departamentos que tienen más de cinco empleados, cada uno de los cuales gana más de 40.000. La pauta es que primero se ejecuta la cláusula WHERE, para seleccionar tuplas individuales; la cláusula HAVING se aplica más tarde, para seleccionar grupos individuales de tuplas. Por tanto, las tuplas ya están restringidas a los empleados que ganan más de 40.000, *antes* de que se aplique la función de la cláusula HAVING. Una forma de escribir correctamente esta consulta es utilizando una consulta anidada, como se muestra en la Consulta 28.

**Consulta 28.** Por cada departamento que tiene más de cinco empleados, recuperar el número de departamento y el número de sus empleados que ganan más de 40.000.

```
C28: SELECT    NumeroDpto, COUNT (*)
     FROM      DEPARTAMENTO, EMPLEADO
     WHERE     NumeroDpto=Dno AND Sueldo>40000 AND
               Dno IN ( SELECT      Dno
                        FROM        EMPLEADO
                        GROUP BY    Dno
                        HAVING      COUNT (*) > 5)
     GROUP BY NumeroDpto;
```

## 8.5.9 Explicación y resumen de las consultas de SQL

Una consulta de recuperación de SQL puede constar de hasta seis cláusulas, pero sólo las dos primeras, SELECT y FROM, son obligatorias. Las cláusulas se especifican en el siguiente orden (las que aparecen entre corchetes son opcionales):

```
SELECT <lista de atributos y funciones >
FROM <lista de tablas>
[ WHERE <condición> ]
[ GROUP BY <atributo(s) de agrupamiento> ]
[ HAVING <condición de agrupamiento> ]
[ ORDER BY <lista de atributos> ];
```

La cláusula SELECT lista los atributos o funciones que se recuperarán. La cláusula FROM especifica todas las relaciones (tablas) que se necesitan en la consulta, incluyendo las relaciones concatenadas, pero no las que están en las consultas anidades. La cláusula WHERE especifica las condiciones de selección de tuplas para esas relaciones, incluyendo, si es necesario, las condiciones de concatenación. GROUP BY especifica los atributos de agrupamiento, mientras que HAVING especifica una condición en los grupos que se están seleccionando, más que en las tuplas individuales. Las funciones agregadas integradas COUNT, SUM, MIN, MAX y AVG se utilizan en combinación con el agrupamiento, pero también se pueden aplicar a todas las tuplas seleccionadas en una consulta sin una cláusula GROUP BY. Por último, ORDER BY especifica un orden para la visualización del resultado de la consulta.

Una consulta se evalúa *conceptualmente*[15] aplicando primero la cláusula FROM (para identificar todas las tablas implicadas en la consulta o para plasmar las tablas concatenadas), seguida por la cláusula WHERE, y después GROUP BY y HAVING. Conceptualmente, ORDER BY se aplica al final para ordenar el resultado de la consulta. Si no se especifica ninguna de las tres últimas cláusulas (GROUP BY, HAVING y ORDER BY), podemos *pensar conceptualmente* en una consulta que se está ejecutando de este modo: por *cada combinación de tuplas* (una de cada una de las relaciones especificadas en la cláusula FROM) se evalúa la cláusula WHERE; si se evalúa como TRUE, coloca en el resultado de la consulta los valores de los atributos especificados en la cláusula SELECT correspondientes a esta combinación de tuplas. Por supuesto, no es una forma eficaz de implementar la consulta en un sistema real, y cada DBMS tiene rutinas especiales de optimización de consultas que ofrecen una ejecución más eficiente. En los Capítulos 15 y 16 explicamos el procesamiento y la optimización de consultas.

En general, en SQL hay varias formas de especificar la misma consulta. Esta flexibilidad tiene ventajas e inconvenientes. La principal ventaja es que el usuario puede elegir la técnica con la que se encuentra más cómodo a la hora de definir una consulta. Por ejemplo, muchas consultas pueden especificarse con condiciones de concatenación en la cláusula WHERE, o utilizando relaciones concatenadas en la cláusula FROM, o con alguna forma de consultas anidadas y el operador de comparación IN. Cada uno puede sentirse cómodo con una metodología diferente. Desde el punto de vista del programador y del sistema respecto a la optimización de la consulta, generalmente es preferible escribir la consulta con la menor cantidad posible de opciones de anidamiento y ordenación.

El inconveniente de tener varias formas de especificar la misma consulta es que el usuario se puede sentir confundido, al no saber la técnica que tiene que utilizar para especificar determinados tipos de consultas. Otro problema es que puede ser más eficaz ejecutar una consulta de una forma que de otra. El DBMS debe procesar la misma consulta de la misma forma, independientemente de cómo se haya especificado. En la práctica, esto es muy complejo, ya que cada DBMS tiene métodos distintos para procesar las consultas especificadas de formas diferentes. Así las cosas, el usuario tiene el trabajo adicional de determinar la especificación alternativa más eficaz. Idealmente, el usuario sólo debe preocuparse de especificar correctamente la consulta. El DBMS tiene la responsabilidad de ejecutar la consulta eficazmente. Sin embargo, en la práctica, ayuda que el usuario sea consciente de cuáles son las estructuras de consulta que más cuesta procesar (consulte el Capítulo 16).

# 8.6 Sentencias INSERT, DELETE y UPDATE de SQL

En SQL se pueden utilizar tres comandos para modificar la base de datos: INSERT, DELETE y UPDATE.

## 8.6.1 Comando INSERT

En su formato más sencillo, INSERT se utiliza para añadir una sola tupla a una relación. Debemos especificar el nombre de la relación y una lista de valores para la tupla. Los valores deben suministrarse *en el mismo orden*

[15] El orden que realmente se sigue para evaluar una consulta depende de la implementación; esto es sólo una forma de ver conceptualmente una consulta a fin de formularla correctamente.

en el que se especificaron los atributos correspondientes en el comando CREATE TABLE. Por ejemplo, para añadir una tupla nueva a la relación EMPLEADO de la Figura 5.5 y especificada con el comando CREATE TABLE EMPLEADO . . . de la Figura 8.1, podemos utilizar U1:

```
U1:  INSERT INTO  EMPLEADO
     VALUES       ( 'Ricardo', 'Roca', 'Flores', '653298653', '30-12-1962',
                    'Los Jarales, 47', 'H', 37000, '653298653', 4 );
```

Una segunda forma de la sentencia INSERT permite especificar explícitamente los nombres de los atributos que se corresponden con los valores suministrados en el comando INSERT. Esto resulta útil si la relación tiene muchos atributos y sólo vamos a asignar valores a unos cuantos en la tupla nueva. Sin embargo, los valores deben incluir todos los atributos con la especificación NOT NULL *y* ningún valor predeterminado. Los atributos que permiten los valores NULL o DEFAULT son los que se pueden *omitir*. Por ejemplo, para introducir una tupla para un nuevo EMPLEADO del que únicamente conocemos los atributos Nombre, Apellido1, Dno y Dni, podemos utilizar U1A:

```
U1A: INSERT INTO  EMPLEADO (Nombre, Apellido1, Dno, Dni)
     VALUES       ( 'Ricardo', 'Roca', 4, '653298653' );
```

Los atributos no especificados en U1A se establecen a DEFAULT o a NULL, y los valores se suministran en el mismo orden que *los atributos enumerados en el propio comando INSERT*. También es posible insertar con un solo comando INSERT *varias tuplas* separadas por comas. Los valores de los atributos que constituyen *cada tupla* se encierran entre paréntesis.

Un DBMS que implementa completamente SQL-99 debe soportar e implementar todas las restricciones de integridad que se pueden especificar en el DDL. Sin embargo, algunos DBMSs no incorporan todas las restricciones, al objeto de mantener toda la eficacia del DBMS y debido a la complejidad de implementar todas las restricciones. Si un sistema no soporta alguna restricción (por ejemplo, la integridad referencial), el usuario o el programador tendrá que implementar la restricción. Por ejemplo, si ejecutamos el comando de U2 sobre la base de datos de la Figura 5.6, un DBMS que no soporte la integridad referencial realizará la inserción aunque no exista una tupla DEPARTAMENTO con NumeroDpto=2. Es responsabilidad del usuario comprobar que no se violan las restricciones *cuyas comprobaciones no están implementadas por el DBMS*. Sin embargo, el DBMS debe implementar comprobaciones para hacer cumplir todas las restricciones de integridad SQL que soporta. Un DBMS que implementa NOT NULL rechazará un comando INSERT en el que un atributo declarado como NOT NULL no tenga un valor; por ejemplo, U2A se *rechazaría* porque no se suministra un valor para Dni.

```
U2:  INSERT INTO  EMPLEADO (Nombre, Apellido1, Dni, Dno)
     VALUES       ( 'Casimiro', 'García', '980760540', 2 );
     (U2 será rechazada si la comprobación de la integridad referencial es proporcionada por el
     DBMS.)

U2A: INSERT INTO  EMPLEADO (Nombre, Apellido1, Dno)
     VALUES       ( 'Casimiro', 'García', 5 );
     (U2A es rechazada si la comprobación NOT NULL es proporcionada por el DBMS.)
```

Una variación del comando INSERT inserta varias tuplas en una relación, en combinación con la creación y la carga de la relación con el *resultado de una consulta*. Por ejemplo, para crear una tabla temporal con el nombre, el número de empleados y el total de sueldos por cada departamento, podemos escribir las sentencias de U3A y U3B:

```
U3A: CREATE TABLE  INFO_DEPTS
                   ( NombreDeDpto VARCHAR(15),
```

```
                    NumDeEmpleados      INTEGER,
                    TotalSueldos        INTEGER );

U3B:  INSERT INTO   INFO_DEPTS ( NombreDeDpto, NumDeEmpleados, TotalSueldos )
      SELECT        NombreDpto, COUNT (*), SUM (Sueldo)
      FROM          ( DEPARTAMENTO JOIN EMPLEADO ON NumeroDpto=Dno )
      GROUP BY      NombreDpto;
```

U3A crea una tabla INFO_DEPTS que se carga con la información de resumen recuperada de la base de datos por la consulta U3B. Ahora podemos consultar INFO_DEPTS como haríamos con cualquier otra relación; cuando ya no la necesitemos la podemos eliminar con el comando DROP TABLE. La tabla INFO_DEPTS puede que no esté actualizada; es decir, si actualizamos las relaciones DEPARTAMENTO o EMPLEADO después de ejecutar U3B, la información de INFO_DEPTS *estará desactualizada*. Tenemos que crear una vista (consulte la Sección 8.8) para mantener actualizada dicha tabla.

## 8.6.2 Comando DELETE

El comando DELETE elimina tuplas de una relación. Incluye una cláusula WHERE, parecida a la que se utiliza en una consulta SQL, para seleccionar las tuplas que se van a eliminar. Las tuplas se eliminan explícitamente sólo de una tabla a la vez. Sin embargo, la eliminación se puede propagar a tuplas de otras relaciones si se han especificado *acciones de activación referencial* en las restricciones de integridad referencial del DDL (consulte la Sección 8.2.2).[16] En función del número de tuplas seleccionadas por la condición de la cláusula WHERE, ninguna, una o varias tuplas pueden ser eliminadas por un solo comando DELETE. La ausencia de una cláusula WHERE significa que se borrarán todas las tuplas de la relación; sin embargo, la tabla permanece en la base de datos, pero vacía. Debemos utilizar el comando DROP TABLE para eliminar la definición de la tabla (consulte la Sección 8.3.1). El comando DELETE de U4A a U4D, si se aplica independientemente a la base de datos de la Figura 5.6, borrará ninguna, una, cuatro y todas las tuplas, respectivamente, de la relación EMPLEADO:

```
U4A:  DELETE FROM   EMPLEADO
      WHERE         Apellido1='Cabrera';

U4B:  DELETE FROM   EMPLEADO
      WHERE         Dni='123456789';

U4C:  DELETE FROM   EMPLEADO
      WHERE         Dno IN ( SELECT     NumeroDpto
                             FROM       DEPARTAMENTO
                             WHERE      NombreDpto='Investigación' );

U4D:  DELETE FROM   EMPLEADO;
```

## 8.6.3 Comando UPDATE

El comando **UPDATE** se utiliza para modificar los valores de atributo de una o más tuplas seleccionadas. Como en el comando DELETE, una cláusula WHERE en el comando UPDATE selecciona las tuplas de una relación que se van a modificar. No obstante, la actualización del valor de una clave principal puede propagarse a los valores de la *foreign key* de las tuplas de otras relaciones en caso de haberse especificado una *acción de activación referencial* en las restricciones de integridad referencial del DDL (consulte la Sección

[16] Se pueden aplicar automáticamente otras acciones a través de *triggers* (consulte la Sección 24.1) y otros mecanismos.

8.2.2). Una cláusula **SET** adicional en el comando UPDATE especifica los atributos que se modificarán y sus nuevos valores. Por ejemplo, para cambiar la ubicación y el número de departamento de control del número de proyecto 10 a 'Valencia' y 5, respectivamente, utilizamos U5:

```
U5:  UPDATE   PROYECTO
     SET      UbicaciónProyecto = 'Valencia', NumDptoProyecto = 5
     WHERE    NumProyecto=10;
```

Algunas tuplas se pueden modificar con un solo comando UPDATE: por ejemplo, subir el sueldo un 10% a todos los empleados del departamento 'Investigación', como se muestra en U6. En esta solicitud, el valor Sueldo modificado depende del valor Sueldo original de cada tupla, por lo que se necesitan dos referencias al atributo Sueldo. En la cláusula SET, la referencia al atributo Sueldo de la derecha se refiere al valor de Sueldo antiguo, *antes de la modificación*, y el situado a la izquierda se refiere al nuevo valor de Sueldo *después de la modificación*:

```
U6:  UPDATE   EMPLEADO
     SET      Sueldo = Sueldo * 1.1
     WHERE    Dno IN ( SELECT    NumeroDpto
                       FROM      DEPARTAMENTO
                       WHERE     NombreDpto='Investigación');
```

También es posible especificar NULL o DEFAULT como nuevo valor de un atributo. Cada comando UPDATE se refiere explícitamente a una sola relación. Para modificar varias relaciones, debemos ejecutar varios comandos UPDATE.

# 8.7 Restricciones como aserciones y *triggers*

En SQL, podemos especificar restricciones generales (las que no encajan en ninguna de las categorías descritas en la Seccción 8.2) mediante **aserciones declarativas**, utilizando la sentencia **CREATE ASSERTION** del DDL. A cada aserción se le asigna un nombre de restricción y se especifica a través de una condición parecida a la cláusula WHERE de una consulta SQL. Por ejemplo, para especificar en SQL la restricción de que *el sueldo de un empleado no debe ser superior al del director del departamento para el que trabaja ese empleado*, podemos escribir la siguiente aserción:

```
CREATE ASSERTION RESTR_SUELDO
CHECK ( NOT EXISTS ( SELECT    *
                     FROM      EMPLEADO E, EMPLEADO M, DEPARTAMENTO D
                     WHERE     E.Sueldo>M.Sueldo
                               AND E.Dno=D.NumeroDpto
                               AND D.DniDirector=M.Dni ) );
```

El nombre de restricción RESTR_SUELDO va seguido por la palabra clave CHECK, que a su vez va seguida por una **condición** entre paréntesis que debe ser verdadera en cada estado de la base de datos para que la aserción sea satisfecha. El nombre de la restricción se puede utilizar más tarde para referirse a dicha restricción o para modificarla o eliminarla. El DBMS es responsable de garantizar que no se viole la condición. Se puede utilizar la condición de cualquier cláusula WHERE, pero pueden especificarse muchas restricciones utilizando el estilo EXISTS y NOT EXISTS de las condiciones SQL. En cuanto algunas tuplas de la base de datos provocan que la condición de una sentencia ASSERTION se evalúe como FALSE, la restricción es **violada**. Por otro lado, la restricción es **satisfecha** por un estado de la base de datos si *ninguna combinación de tuplas* de dicho estado viola la restricción.

La técnica básica para escribir estas aserciones consiste en especificar una consulta que seleccione las tuplas *que violan la condición deseada*. Al incluir esta consulta dentro de una cláusula NOT EXISTS, la aserción especificará que el resultado de esta consulta debe estar vacío. De este modo, la aserción es violada si el resultado de la consulta no está vacío. En nuestro ejemplo, la consulta selecciona todos los empleados cuyo sueldo es mayor que el sueldo del director de su departamento. Si el resultado de la consulta no está vacío, la aserción es violada.

La cláusula CHECK y la condición de la restricción también se pueden utilizar para especificar restricciones en los atributos y las condiciones (consulte la Sección 8.2.1) y en las tuplas (consulte la Sección 8.2.4). Una diferencia más importante entre CREATE ASSERTION y las otras dos es que las cláusulas CHECK en los atributos, los dominios y las tuplas se comprueban en SQL *sólo cuando se insertan o actualizan las tuplas*. Por tanto, la comprobación de la restricción se puede implementar de forma más eficaz por parte de los DBMSs en estos casos. El diseñador del esquema debe utilizar CHECK en los atributos, los dominios y las tuplas sólo cuando está seguro de que la restricción sólo puede ser violada por la inserción o la actualización de tuplas. Por el contrario, el diseñador del esquema debe usar CREATE ASSERTION sólo en los casos en que no es posible utilizar CHECK en los atributos, los dominios o las tuplas, para que el DBMS pueda realizar las comprobaciones con más eficacia.

Otra sentencia relacionada con CREATE ASSERTION es CREATE TRIGGER, pero los *triggers* se utilizan de modo diferente. En muchos casos es conveniente especificar el tipo de acción que se llevará a cabo cuando se producen ciertos eventos y cuando se satisfacen determinadas condiciones. En lugar de ofrecer al usuario la opción de abortar una operación que provoca una violación (como con CREATE ASSERTION) el DBMS debe contar con otras opciones. Por ejemplo, puede ser útil especificar una condición que, en caso de ser violada, informe al usuario de alguna forma. Un director puede estar interesado en ser informado de si los gastos de viaje de un empleado exceden un cierto límite, recibiendo un mensaje cuando esto sucede. La acción que el DBMS debe tomar en este caso es enviar un mensaje a ese usuario. La condición se utiliza entonces para **monitorizar** la base de datos. Es posible especificar otras acciones, como la ejecución de un procedimiento almacenado específico o la activación de otras actualizaciones. En SQL se utiliza la sentencia CREATE TRIGGER para implementar dichas acciones. Un ***trigger*** especifica un **evento** (por ejemplo, una operación de actualización de la base de datos), una **condición** y una **acción**. La acción se ejecuta automáticamente si se satisface la condición cuando se produce el evento. En la Sección 24.1 se explican en detalle los *triggers*.

# 8.8 Vistas (tablas virtuales) en SQL

En esta sección introducimos el concepto de vista. Veremos cómo se especifican, explicaremos el problema de actualizar vistas y cómo se pueden implementar en los DBMS.

## 8.8.1 Concepto de vista en SQL

Una **vista** en terminología SQL es una tabla que deriva de otras tablas.[17] Esas otras tablas pueden ser tablas base o vistas definidas anteriormente. Una vista no existe necesariamente en formato físico; está considerada como una **tabla virtual**, en oposición a las tablas base, cuyas tuplas están realmente almacenadas en la base de datos. Esto limita las posibles operaciones de actualización que pueden aplicarse a las vistas, pero no ofrecen limitación alguna al consultar una vista.

Podemos pensar que una vista es una forma de especificar una tabla a la que nos referimos con frecuencia, aunque no exista físicamente. Por ejemplo, en la Figura 5.5 podemos emitir consultas frecuentes para recuperar el nombre del empleado y el nombre de los proyectos en los que trabaja. En lugar de tener que especificar

[17] Como se utiliza en SQL, el término vista es más limitado que el término vista de usuario que explicamos en los Capítulos 1 y 2, puesto que es posible que una vista de usuario incluya muchas relaciones.

la concatenación de las tablas EMPLEADO, TRABAJA_EN y PROYECTO cada vez que realizamos esta consulta, podemos definir una vista que es el resultado de estas concatenaciones. Después podemos emitir consultas en la vista, que se especifican como recuperaciones de una sola tabla y no como recuperaciones en las que se ven implicadas dos concatenaciones o tres tablas. Podemos llamar a las tablas EMPLEADO, TRABAJA_EN y PROYECTO **tablas de definición** de la vista.

## 8.8.2 Especificación de vistas en SQL

En SQL se utiliza el comando **CREATE VIEW** para especificar una vista. A una vista se le asigna un nombre de tabla (virtual), o nombre de vista, una lista de nombres de atributos y una consulta que sirve para especificar el contenido de la vista. Si ninguno de los atributos de la vista resulta de aplicar funciones u operaciones aritméticas, no tenemos que especificar nombres de atributos para la vista, puesto que serían idénticos a los de los atributos de las tablas de definición. Las vistas V1 y V2 crean tablas virtuales, cuyos esquemas se ilustran en la Figura 8.7, cuando se aplica al esquema de la base de datos de la Figura 5.5.

```
V1:  CREATE VIEW    TRABAJA_EN1
     AS  SELECT     Nombre, Apellido1, NombreProyecto, Horas
         FROM       EMPLEADO, PROYECTO, TRABAJA_EN
         WHERE      Dni=DniEmpleado AND NumProy=NumProyecto;

V2:  CREATE VIEW    INFO_DPTO(NombreDeDpto, NumDeEmpleados, TotalSueldos)
     AS  SELECT     NombreDpto, COUNT (*), SUM (Sueldo)
         FROM       DEPARTAMENTO, EMPLEADO
         WHERE      NumeroDpto=Dno
         GROUP BY   NombreDpto;
```

En V1, no especificamos ningún nombre de atributo nuevo para la vista TRABAJA_EN1 (aunque lo podríamos hacer); en este caso, TRABAJA_EN1 *hereda* los nombres de los atributos de la vista de las tablas de definición EMPLEADO, PROYECTO y TRABAJA_EN. La vista V2 especifica explícitamente nombres de atributo nuevos para la vista INFO_DPTO, mediante una correspondencia "uno a uno" entre los atributos especificados en la cláusula CREATE VIEW y los especificados en la cláusula SELECT de la consulta que define la vista.

Ahora ya podemos especificar consultas SQL sobre la vista (o tabla virtual) de la misma forma que especificamos consultas contra las tablas base. Por ejemplo, para recuperar el nombre y el primer apellido de todos los empleados que trabajan en el proyecto 'ProductoX', podemos utilizar la vista TRABAJA_EN1 y especificar la consulta como en CV1:

```
CV1: SELECT     Nombre, Apellido1
     FROM       TRABAJA_EN1
     WHERE      NombreProyecto='ProductoX';
```

**Figura 8.7.** Dos vistas especificadas en el esquema de la base de datos de la Figura 5.5.

**TRABAJA_EN1**

| Nombre | Apellido1 | NombreProyecto | Horas |
|---|---|---|---|

**INFO_DPTO**

| NombreDeDpto | NumDeEmpleados | TotalSueldos |
|---|---|---|

La misma consulta requeriría la especificación de dos concatenaciones en caso de especificarse sobre las relaciones base; una de las principales ventajas de una vista es que simplifica la especificación de determinadas consultas. Las vistas también se utilizan como mecanismo de seguridad y autorización (consulte el Capítulo 23).

Se supone que una vista *siempre está actualizada*; si modificamos las tuplas de las tablas base sobre las que se define la vista, esta última debe reflejar esos cambios automáticamente. Por tanto, la vista no se materializa al *definir la vista*, sino al *especificar una consulta* en la vista. La tarea de que la vista esté actualizada es responsabilidad del DBMS, y no del usuario.

Si ya no necesitamos una vista, podemos utilizar el comando **DROP VIEW** para deshacernos de ella. Por ejemplo, para deshacernos de la vista V1, podemos utilizar la sentencia SQL de V1A:

```
V1A: DROP VIEW TRABAJA_EN1;
```

### 8.8.3 Implementación y actualización de vistas

El problema de implementar eficazmente una vista para consultas es complejo. Se han sugerido dos metodologías principalmente. La denominada **modificación de consulta** implica modificar la consulta de la vista para que sea una consulta de las tablas base subyacentes. Por ejemplo, el DBMS modificaría automáticamente la consulta CV1 por la siguiente consulta:

```
SELECT   Nombre, Apellido1
FROM     EMPLEADO, PROYECTO, TRABAJA_EN
WHERE    Dni=DniEmpleado AND NumProy=NumProyecto
         AND NombreProyecto='ProductoX';
```

El inconveniente de esta metodología es su ineficacia para las vistas definidas mediante consultas complejas cuya ejecución lleva mucho tiempo, especialmente si se aplican varias consultas a la vista dentro de un corto periodo de tiempo. La otra estrategia, denominada **materialización de la vista**, implica la creación física de una tabla de vista temporal cuando la vista es consultada primero y se mantiene esa tabla en la suposición de que en la vista seguirán otras consultas. En este caso, debe desarrollarse una estrategia eficaz para actualizar automáticamente la tabla de la vista cuando las tablas base son actualizadas, a fin de mantener la vista actualizada. Con este propósito se han desarrollado técnicas que utilizan el concepto de **actualización incremental**, según las cuales se determinan las tuplas que deben insertarse, eliminarse o modificarse en una tabla de vista materializada cuando se introduce un cambio en alguna de las tablas base de definición. La vista se conserva mientras se esté consultando. Si no se consulta la vista en un periodo de tiempo determinado, el sistema puede entonces eliminar automáticamente la tabla de vista física y volver a calcularla desde el principio cuando en el futuro alguna consulta haga referencia a la vista.

La actualización de las vistas es compleja y puede ser ambigüa. En general, la actualización en una vista definida en *una sola tabla* sin *funciones agregadas* puede asignarse a una actualización de la tabla base subyacente si se cumplen ciertas condiciones. En el caso de una vista que implica concatenaciones, una operación de actualización puede mapearse de varias formas a operaciones de actualización de las relaciones base subyacentes. Para ilustrar los problemas potenciales que pueden surgir con la actualización de una vista definida con varias tablas, considere la vista TRABAJA_EN1, y suponga que emitimos un comando para actualizar el atributo NombreProyecto correspondiente a 'José Pérez' de 'ProductoX' a 'ProductoY'. En UV1 mostramos la actualización:

```
UV1:  UPDATE   TRABAJA_EN1
      SET      NombreProyecto = 'ProductoY'
      WHERE    Apellido1='Pérez' AND Nombre='José'
               AND NombreProyecto='ProductoX';
```

Esta consulta puede mapearse a varias actualizaciones en las relaciones base para obtener el efecto de actualización deseado en la vista. Aquí mostramos dos posibles actualizaciones, (a) y (b), en las relaciones base correspondientes a UV1:

```
(a):  UPDATE   TRABAJA_EN
      SET      NumProy =     ( SELECT   NumProyecto
                               FROM     PROYECTO
                               WHERE    NombreProyecto='ProductoY' )
      WHERE    DniEmpleado IN ( SELECT  Dni
                              FROM      EMPLEADO
                              WHERE     Apellido1='Pérez' AND Nombre='José' )
               AND
               NumProy =     ( SELECT   NumProyecto
                               FROM     PROYECTO
                               WHERE    NombreProyecto='ProductoX' );

(b):  UPDATE   PROYECTO SET              NombreProyecto = 'ProductoY'
      WHERE    NombreProyecto = 'ProductoX';
```

La actualización (a) relaciona a 'José Pérez' con la tupla PROYECTO 'ProductoY' en lugar de con la tupla PROYECTO 'ProductoX' y es probablemente la actualización deseada. Sin embargo, (b) también ofrecería el efecto de actualización deseado en la vista, pero se logra cambiando el nombre de la tupla 'ProductoX' de la relación PROYECTO por 'ProductoY'. Es bastante improbable que el usuario que especificó la actualización de vista UV1 quisiera interpretar la actualización como en (b), puesto que esto también tiene el efecto secundario de cambiar todas las tuplas de vista con NombreProyecto='ProductoX'.

Algunas actualizaciones de vista no tienen mucho sentido; por ejemplo, la modificación del atributo TotalSueldos de la vista INFO_DPTO no tiene sentido porque TotalSueldos está definido como la suma de los sueldos individuales de los empleados. Esta solicitud se muestra en UV2:

```
UV2:  UPDATE   INFO_DPTO
      SET      TotalSueldos=100000
      WHERE    NombreDpto='Investigación';
```

Un gran número de actualizaciones en las relaciones base subyacentes pueden satisfacer esta actualización de vista.

Una actualización de vista es factible cuando *sólo una actualización posible* en las relaciones base puede lograr el efecto de actualización deseado en la vista. Siempre que una actualización en la vista pueda mapearse a *más de una actualización* en las relaciones base subyacentes, deberemos tener un determinado procedimiento para elegir la actualización deseada. Algunos investigadores han desarrollado métodos para elegir la actualización más probable, mientras que otros prefieren dejar que sea el usuario quien elija el mapeado de actualización deseado durante la definición de la vista.

En resumen, podemos hacer estas observaciones:

- Una vista con una sola tabla de definición es actualizable si los atributos de la vista contienen la clave principal de la relación base, así como todos los atributos con la restricción NOT NULL *que no tienen valores predeterminados* especificados.
- Las vistas definidas sobre varias tablas y que utilizan concatenaciones normalmente no son actualizables.
- Las vistas definidas utilizando funciones de agrupamiento y agregación no son actualizables.

En SQL, debe añadirse la cláusula **WITH CHECK OPTION** al final de la definición de la vista si ésta *va a actualizarse*. De este modo, el sistema puede comprobar la posibilidad de actualización de la vista y planificar una estrategia de ejecución para las actualizaciones de la vista.

## 8.9 Características adicionales de SQL

SQL tiene una serie de características adicionales que no hemos descrito en este capítulo, pero que explicamos en otra parte del libro. Son las siguientes:

- SQL dispone de varias técnicas diferentes para escribir programas en distintos lenguajes de programación, en los que se pueden incluir sentencias de SQL para acceder a una o más bases de datos. Nos referimos a SQL incrustado (dinámico), SQL/CLI (Interfaz de lenguaje de llamadas) y su predecesor ODBC (Conectividad de bases de datos abierta, *Open Data Base Connectivity*), y SQL/PSM (Módulos de programa almacenados, *Program Stored Modules*). En la Sección 9.1 explicamos las diferencias entre estas técnicas, y después explicamos las distintas técnicas en las Secciones 9.2 a 9.4. También explicamos cómo acceder a las bases de datos SQL a través del lenguaje de programación Java utilizando JDBC y SQLJ.
- Cada RDBMS comercial tendrá, además de los comandos SQL, un conjunto de comandos para especificar los parámetros de diseño físico de la base de datos, las estructuras de fichero de las relaciones, y las rutas de acceso, como, por ejemplo, los índices. En el Capítulo 2 denominamos a estos comandos *lenguajes de definición de almacenamiento (SDL)*. Las versiones anteriores de SQL tenían comandos para **crear índices**, pero se eliminaron del lenguaje porque no se encontraban en el nivel de esquema conceptual (consulte el Capítulo 2).
- SQL tiene comandos para controlar las transacciones. Se utilizan para especificar las unidades de procesamiento de bases de datos al objeto de controlar la concurrencia y la recuperación. Explicamos estos comandos en el Capítulo 17, después de explicar el concepto de transacción más en detalle.
- SQL tiene estructuras de lenguaje para especificar la *concesión y revocación de privilegios* a los usuarios. Los privilegios normalmente corresponden al derecho de utilizar ciertos comandos SQL para acceder a determinadas relaciones. A cada relación se le asigna un propietario, y el propietario o el personal del DBA pueden otorgar a los usuarios seleccionados el privilegio de utilizar una sentencia SQL (por ejemplo, SELECT, INSERT, DELETE o UPDATE) para acceder a la relación. Además, el personal del DBA puede otorgar los privilegios para crear esquemas, tablas o vistas a ciertos usuarios. Estos comandos SQL (denominados **GRANT** y **REVOKE**) se explican en el Capítulo 23, donde hablamos de la seguridad de la base de datos y de la autorización.
- SQL tiene estructuras de lenguaje para ciertos *triggers*. Normalmente se conocen como técnicas de **base de datos activa**, puesto que especifican acciones que son automáticamente ejecutadas por los eventos (por ejemplo, las actualizaciones de bases de datos). Explicamos estas características en la Sección 24.1, donde hablamos del concepto de base de datos activa.
- SQL ha incorporado muchas características de los modelos orientados a objetos, sobre todo para mejorar los sistemas relacionales conocidos como de objetos relacionales (**objeto-relacional**). En el Capítulo 22 explicamos capacidades como la creación de atributos estructurados complejos (también conocidos como **relaciones anidadas**), la especificación de tipos de datos abstractos (denominados UDT o tipos definidos por el usuario) para atributos y tablas, la creación de **identificadores de objetos** para hacer referencia a las tuplas, y la especificación de **operaciones** sobre los tipos.
- SQL y las bases de datos relacionales pueden interactuar con las nuevas tecnologías, como XML (consulte el Capítulo 27) y OLAP (consulte el Capítulo 28).

# 8.10 Resumen

En este capítulo hemos presentado el lenguaje de bases de datos SQL. Este lenguaje y sus variaciones se han implementado como interfaces en muchos DBMSs relacionales comerciales, como Oracle, DB2 y SQL/DS de IBM, SQL Server y ACCESS de Microsoft, SYBASE, INFORMIX, Rdb de Digital[18] e INGRES. Algunos sistemas OpenSource también proporcionan SQL, como MySQL. La versión original de SQL se implementó en el DBMS experimental denominado SYSTEM R, que fue desarrollado por IBM Research. SQL está diseñado como un lenguaje global que incluye sentencias para la definición de datos, consultas, actualizaciones, especificación de restricciones y definición de vistas. Hemos explicado muchas de ellas en secciones independientes de este capítulo. En la última sección vimos las características adicionales que se explican en otras partes del libro. Nos hemos centrado en el estándar SQL-99.

La Tabla 8.2 resume la sintaxis (o estructura) de varias sentencias de SQL. Este resumen no está pensado para describir todas las posibles estructuras de SQL, sino como referencia rápida de los principales tipos de estructuras disponibles en SQL. Utilizamos la notación BNF, según la cual los símbolos obligatorios se escriben entre corchetes angulares <. . .>, las partes opcionales se muestran entre corchetes [. . .], las repeteciones entre llaves {. . .}, y las alternativas entre paréntesis (. . . | . . . | . . .).[19]

## Preguntas de repaso

**8.1.** ¿En qué se diferencian las relaciones (tablas) de SQL de las relaciones definidas formalmente en el Capítulo 5? Explique las diferencias de terminología. ¿Por qué SQL permite tuplas duplicadas en una tabla o en el resultado de una consulta?

**8.2.** Enumere los tipos de datos permitidos para los atributos SQL.

**8.3.** ¿Cómo permite SQL la implementación de las restricciones de integridad de entidad y de integridad referencial descritas en el Capítulo 5? ¿Qué sabe sobre las acciones de activación referencial?

**8.4.** Describa las seis cláusulas de la sintaxis de una consulta de recuperación SQL. Muestre los tipos de estructuras que se pueden especificar en cada una de las seis cláusulas. ¿Cuál de las seis cláusulas son necesarias y cuáles son opcionales?

**8.5.** Describa conceptualmente cómo se ejecutará una consulta de recuperación SQL al especificar el orden conceptual de ejecución de cada una de las seis cláusulas.

**8.6.** Explique cómo se tratan los valores NULL en los operadores de comparación de SQL. ¿Cómo se tratan los NULLs cuando se aplican funciones agregadas en una consulta SQL? ¿Cómo se tratan los NULLs si existen en los atributos de agrupamiento?

## Ejercicios

**8.7.** Considere la base de datos de la Figura 1.2, cuyo esquema aparece en la Figura 2.1. ¿Cuáles son las restricciones de integridad referencial que deben mantenerse en el esquema? Escriba las sentencias DDL SQL apropiadas para definir la base de datos.

**8.8.** Repita el Ejercicio 8.7, pero utilice el esquema de la base de datos LÍNEA_AÉREA de la Figura 5.8.

**8.9.** Tome el esquema de la base de datos relacional BIBLIOTECA de la Figura 6.14. Seleccione la acción apropiada (rechazar, cascada, establecer a NULL, establecer al valor predeterminado) para cada una de las restricciones de integridad referencial, tanto para *eliminar* una tupla referenciada como para *actualizar* el valor de atributo de una clave principal en una tupla referenciada. Justifique sus opciones.

---

[18] Rdb fue adquirido por Oracle de Digital en 1994 y desde entonces se ha mejorado.

[19] La sintaxis completa de SQL-99 se describe en muchos documentos voluminosos, de cientos de páginas.

**Tabla 8.2.** Resumen de la sintaxis de SQL.

| |
|---|
| CREATE TABLE <nombre de tabla> ( <nombre de columna>< tipo de columna>[<restricción de atributo> ]<br>{ , <nombre de columna><tipo de columna> [ <restrición de atributo> ] }<br>[ <restricción de tabla> { , <restricción de tabla> } ] ) |
| DROP TABLE <nombre de tabla> |
| ALTER TABLE <nombre de tabla> ADD <nombre de columna><tipo de columna> |
| SELECT [ DISTINCT ] <lista de atributos><br>FROM ( <nombre de tabla> { <alias> } \| <tabla concatenada> )<br>{ , ( <nombre de tabla> { <alias> } \| <tabla concatenada> ) }<br>[ WHERE <condición>]<br>[ GROUP BY <atributos de agrupamiento>[ HAVING <condición de selección de grupo> ] ]<br>[ ORDER BY <nombre de columna> [ <orden>] { , <nombre de columna> [ <orden> ] } ] |
| <attribute list> ::= ( * \| ( <nombre de columna> \| <función> ( ( [ DISTINCT ] <nombre de columna> \| * * ) ) )<br>{ , ( <nombre de columna> \| <función> ( ( [ DISTINCT] <nombre de columna> \| * * ) ) } ) ) |
| <atributos de agrupamiento> ::= <nombre de columna> { , <nombre de columna> } |
| <orden> ::= ( ASC \| DESC ) |
| INSERT INTO <nombre de tabla> [ ( <nombre de columna> { , <nombre de columna> } ) ]<br>( VALUES ( <valor constante> , { <valor constante> } ) { , ( <valor constante> { , <valor constante> } ) }<br>\| <sentencia de selección>) |
| DELETE FROM <nombre de tabla><br>[ WHERE <condición de selección> ] |
| UPDATE <nombre de tabla><br>SET <nombre de columna>= <expresión valor> { , <nombre de columna> = <expresión valor> }<br>[ WHERE <condición de selección>] |
| CREATE [ UNIQUE] INDEX <nombre de índice><br>ON <nombre de tabla>( <nombre de columna>[ <orden> ] { , <nombre de columna> [ <orden> ] } )<br>[ CLUSTER ] |
| DROP INDEX <nombre de índice> |
| CREATE VIEW <nombre de vista> [ ( <nombre de columna> { , <nombre de columna>>} ) ]<br>AS <sentencia de selección> |
| DROP VIEW <nombre de vista> |

NOTA: Los comandos para crear y eliminar índices no forman parte del estándar SQL2.

**8.10.** Escriba las sentencias DDL SQL apropiadas para declarar el esquema de la base de datos relacional BIBLIOTECA de la Figura 6.14. Especifique las claves apropiadas y las acciones de activación referencial.

**8.11.** Escriba consultas SQL para las consultas a la base de datos BIBLIOTECA ofrecidas en el Ejercicio 6.18.

**8.12.** ¿Cómo puede implementar el DBMS las restricciones de clave y de *foreign key*? ¿Es difícil llevar a cabo la técnica de implementación que ha sugerido? ¿Se pueden ejecutar eficazmente las comprobaciones de las restricciones cuando se aplican actualizaciones a la base de datos?

**8.13.** Especifique en SQL las consultas del Ejercicio 6.16. Muestre el resultado de las consultas si se aplican a la base de datos EMPRESA de la Figura 5.6.

**8.14.** Especifique en SQL las siguientes consultas adicionales sobre la base de datos de la Figura 5.5. Muestre el resultado de las consultas si se aplica a la base de datos de la Figura 5.6.

- **a.** Por cada departamento cuyo salario medio es superior a 30.000, recupere el nombre del departamento y el número de empleados que trabajan en él.
- **b.** Suponga que queremos saber el número de empleados *masculinos* de cada departamento, y no el número total de empleados (como en el Ejercicio 8.14a). ¿Puede especificar esta consulta en SQL? ¿Por qué o por qué no?

**8.15.** Especifique las actualizaciones del Ejercicio 5.10 utilizando los comandos de actualización SQL.

**8.16.** Especifique en SQL las siguientes consultas sobre el esquema de la base de datos de la Figura 1.2.

- **a.** Recupere el nombre de todos los estudiantes sénior especializados en 'CC' (ciencias de la computación).
- **b.** Recupere el nombre de todos los cursos impartidos por el Profesor Pedro en 2004 y 2005.
- **c.** Por cada sección impartida por el Profesor Pedro, recupere el número de curso, el semestre, el año y el número de estudiantes que tomaron la sección.
- **d.** Recupere el nombre y el certificado de estudios de cada estudiante sénior (Clase=4) especializado en CC. Un certificado de estudios incluye el nombre y el número del curso, las horas de crédito, el semestre, el año y la calificación obtenida en cada curso completado por el estudiante.
- **e.** Recupere los nombres y los departamentos de especialización de todos los estudiantes con matrícula de honor (estudiantes que han obtenido la calificación A en todos sus cursos).
- **f.** Recupere los nombres y los departamentos de especialización de todos los estudiantes que no tienen una calificación de A en ninguno de sus cursos.

**8.17.** Escriba sentencias de actualización SQL para hacer lo siguiente sobre el esquema de la base de datos de la Figura 1.2.

- **a.** Inserte un estudiante nuevo, <'Federico', 25, 1, 'Mat'>, en la base de datos.
- **b.** Cambie la clase del estudiante 'Luis' a 2.
- **c.** Inserte un curso nuevo, <'Ingeniería del conocimiento', 'CC4390', 3, 'CC'>.
- **d.** Elimine el registro del estudiante cuyo nombre es 'Luis' y cuyo número de estudiante es 17.

**8.18.** Especifique en SQL las consultas y las actualizaciones de los Ejercicios 6.17 y 5.11, que se refieren a la base de datos LÍNEA_AÉREA (véase la Figura 5.8).

**8.19.** Diseñe un esquema de base de datos relacional para su aplicación de base de datos.

- **a.** Declare sus relaciones, utilizando el DDL de SQL.
- **b.** Especifique varias consultas en SQL que su aplicación de base de datos necesita.
- **c.** Basándose en el uso que se espera hacer de la base de datos, seleccione algunos atributos que deberían tener unos índices especificados.
- **d.** Implemente su base de datos, si tiene un DBMS que soporte SQL.

**8.20.** Especifique en SQL las respuestas a los Ejercicios 6.19 a 6.21 y el Ejercicio 6.23.

**8.21.** En SQL, especifique las siguientes consultas en la base de datos de la Figura 5.5 utilizando el concepto de consultas anidadas y los conceptos descritos en este capítulo.

**a.** Recupere los nombres de todos los empleados que trabajan en el departamento que cuenta con el empleado con el sueldo más alto de todos los empleados.

**b.** Recupere los nombres de todos los empleados cuyo supervisor tiene el DNI '888665555'.

**c.** Recupere los nombres de los empleados que ganan al menos 10.000 más que el empleado que menos gana de la empresa.

**8.22.** Considere la restricción SUPERFKEMP de la tabla EMPLEADO, que se ha modificado del siguiente modo respecto a lo especificado en la Figura 8.2:

```
CONSTRAINT SUPERFKEMP
        FOREIGN KEY (SuperDni) REFERENCES EMPLEADO(Dni)
                ON DELETE CASCADE ON UPDATE CASCADE,
```

Responda las siguientes cuestiones:

**a.** ¿Qué ocurre al ejecutar el siguiente comando sobre el estado de base de datos mostrado en la Figura 5.6?

```
DELETE EMPLEADO WHERE Apellido1 = 'Ochoa'
```

**b.** ¿Es mejor CASCADE o SET NULL en caso de la restricción SUPERFKEMP ON DELETE?

**8.23.** Escriba sentencias SQL para crear una tabla EMPLEADO_BACKUP que sirva como copia de seguridad de la tabla EMPLEADO de la Figura 5.6.

**8.24.** Especifique las siguientes vistas en SQL sobre el esquema de la base de datos EMPRESA de la Figura 5.5.

**a.** Una vista por departamentos, con el nombre de cada departamento, el nombre de su director y su sueldo.

**b.** Una vista del departamento 'Investigación' con los nombres de sus empleados, el nombre del supervisor y el sueldo de los primeros.

**c.** Una vista por proyectos, con el nombre del proyecto, el nombre del departamento de control, el número de empleados y el total de horas trabajadas por semana en el proyecto.

**d.** Una vista de los proyectos *en los que trabaje más de un empleado*, con el nombre del proyecto, el nombre del departamento de control, el número de empleados y el total de horas trabajadas por semana en el proyecto.

**8.25** Teniendo en cuenta la siguiente vista, RESUMEN_DPTO, definida en la base de datos EMPRESA de la Figura 5.6:

```
CREATE VIEW     RESUMEN_DPTO (D, C, Total_s, Promedio_s)
AS SELECT       Dno, COUNT (*), SUM (Sueldo), AVG (Sueldo)
FROM            EMPLEADO
GROUP BY        Dno;
```

Explique cuáles de las siguientes consultas y actualizaciones estarían permitidas en la vista. Si una consulta o actualización se pudiera permitir, muestre el aspecto que tendría esa consulta o actualización en las relaciones base, y proporcione el resultado de su aplicación a la base de datos de la Figura 5.6.

**a.**
```
SELECT  *
FROM    RESUMEN_DPTO;
```

**b.**
```
SELECT D, C
FROM    RESUMEN_DPTO
WHERE   TOTAL_S > 100000;
```

```
c.  SELECT   D, PROMEDIO_S
    FROM     RESUMEN_DPTO
    WHERE    C > (SELECT C FROM RESUMEN_DPTO WHERE D=4);
d.  UPDATE   RESUMEN_DPTO
    SET      D=3
    WHERE    D=4;
e.  DELETE   FROM RESUMEN_DPTO
    WHERE    C > 4;
```

## Ejercicios de práctica

**8.26.** Partiendo del esquema SQL generado para la base de datos UNIVERSIDAD del ejercicio de práctica 7.9:

**a.** Cree las tablas de la base de datos en Oracle utilizando la interfaz SQLPlus.

**b.** Cree ficheros de datos separados por comas con los datos de al menos tres departamentos (Ciencias de la computación, Matemáticas y Biología), 20 estudiantes y 20 cursos. Puede distribuir los estudiantes y los cursos uniformemente entre los departamentos. También puede asignar especializaciones secundarias a algunos estudiantes. Para un subconjunto de cursos, cree secciones para el otoño de 2006 y la primavera de 2007. Asegúrese de asignar varias secciones para algunos cursos. Asumiendo que están disponibles las calificaciones correspondientes a otoño de 2006, añada matriculaciones de estudiantes en las secciones y asigne calificaciones numéricas para las matriculaciones. No añada ninguna matriculación para las secciones de la primavera de 2007.

**c.** Con la utilidad SQLLoader de Oracle, cargue en la base de datos los datos creados en el apartado (b).

**d.** Escriba consultas SQL para lo siguiente y ejecútelas dentro de una sesión SQLPlus:

i. Recupere el número, el nombre y los apellidos, y la especialidad de los estudiantes con una especialidad menor en biología.

ii. Recupere el número de estudiante y el nombre y los apellidos de los estudiantes que nunca han asistido a una clase impartida por el Profesor Pedro.

iii. Recupere el número de estudiante y el nombre y los apellidos de los estudiantes que sólo han asistido a las clases impartidas por el Profesor Pedro.

iv. Recupere los nombres de departamento y el número de estudiantes con ese departamento como especialidad principal (resultado ordenado descendentemente por el número de estudiantes).

v. Recupere los nombres de los profesores que imparten cursos de Ciencias de la computación, las secciones (número de curso, sección, semestre y año) que están impartiendo y el número total de estudiantes en las secciones.

vi. Recupere el número, nombre y apellidos, y las especialidades de los estudiantes que no tienen una 'A' en ninguno de los cursos en los que están matriculados.

vii. Recupere el número de estudiante, el nombre y los apellidos, y las especialidades de los estudiantes con matrícula de honor (estudiantes que han obtenido 'A' en todos los cursos en los que se han matriculado).

viii Por cada estudiante del departamento de Ciencias de la computación, recupere el número de estudiante, el nombre y los apellidos, y el GPA.

**8.27.** Partiendo del esquema SQL generado para la base de datos PEDIDOS_CORREO del Ejercicio 7.10:

**a.** Cree las tablas de la base de datos en Oracle utilizando la interfaz SQLPlus.

**b.** Cree ficheros de datos separados por comas con los datos de al menos seis clientes, seis empleados y 20 repuestos. Además, cree datos para 20 pedidos, con un promedio de tres a cuatro repuestos por pedido.

**c.** Con la utilidad SQLLoader de Oracle, cargue en la base de datos los datos creados en el apartado (b).

**d.** Escriba consultas SQL para lo siguiente y ejecútelas dentro de una sesión SQLPlus:

i. Recupere los nombres de los clientes que han pedido al menos un repuesto cuyo precio es superior a 30,00 euros.

ii. Recupere los nombres de los clientes que han pedido repuestos que cuestan todos menos de 20,00 euros.

iii. Recupere los nombres de los clientes que sólo han realizado pedidos a los empleados que viven en su misma ciudad.

iv. Cree una lista con el número, nombre y cantidad total de los repuestos pedidos. Genere una lista ordenada descendentemente por la cantidad total pedida.

v. Calcule el tiempo medio de espera (número de días) de todos los pedidos. El tiempo de espera se calcula como la diferencia entre la fecha de recepción del pedido y la fecha de venta, redondeando al día más cercano.

vi. Por cada empleado, recupere su número, nombre y total de ventas (en términos de euros) en un año dado (por ejemplo, 2006).

## Bibliografía seleccionada

El lenguaje SQL, originalmente conocido como SEQUEL, estaba basado en el lenguaje SQUARE (*Specifying Queries as Relational Expressions*), descrito por Boyce y otros (1975). La sintaxis de SQUARE se modificó y quedó en SEQUEL (Chamberlin and Boyce 1974) y después pasó a denominarse SEQUEL 2 (Chamberlin y otros, 1976), en el que está basado SQL. La implementación original de SEQUEL se realizó en IBM Research, San José, California.

Reisner (1977) describe una evaluación de los factores humanos de SEQUEL en los que ella encontró que los usuarios tienen alguna dificultad a la hora de especificar correctamente las condiciones de concatenación y el agrupamiento. Date (1984b) contiene una crítica del lenguaje SQL centrándose en sus puntos fuertes y sus fallos. Date y Darwen (1993) describe SQL2. ANSI (1986) perfila el estándar SQL original, y ANSI (1992) describe el estándar SQL2. Los manuales de diferentes fabricantes describen las características de SQL tal como está implementado en DB2, SQL/DS, Oracle, INGRES, INFORMIX y otros productos DBMS comerciales. Melton y Simon (1993) es un texto extenso de SQL2. Horowitz (1992) explica algunos de los problemas relacionados con la integridad referencial y la propagación de actualizaciones en SQL2.

La cuestión de las actualizaciones de vista se trata en Dayal y Bernstein (1978), Keller (1982) y Langerak (1990), entre otros. La implementación de vistas se explica en Blakeley y otros (1989). Negri y otros (1991) describe la semántica formal de las consultas SQL. Hay muchos libros que describen distintos aspectos de SQL. Por ejemplo, dos referencias que describen SQL-99 son Melton y Simon (2002) y Melton (2003).

CAPÍTULO 9

# Introducción a las técnicas de programación SQL

En el capítulo anterior describimos algunos aspectos del lenguaje SQL, el estándar para las bases de datos relacionales. Vimos las sentencias SQL para definir los datos, modificar los esquemas y especificar consultas, vistas y actualizaciones. También describimos cómo se especifican las restricciones comunes, como las de clave y de integridad referencial.

En este capítulo explicaremos diferentes técnicas para acceder a las bases de datos desde los programas. Casi todos estos tipos de accesos se realizan a través de aplicaciones que implementan bases de datos. Este software se desarrolla normalmente en un lenguaje de propósito general, como Java, COBOL o C/C++. Como recordará de la Sección 2.3.1, cuando en un programa se incluyen sentencias de bases de datos, el lenguaje de programación de propósito general se conoce como *lenguaje host*, mientras que el lenguaje de base de datos (SQL, en nuestro caso) se conoce como *sublenguaje de datos*. En algunos casos, se desarrollan específicamente lenguajes de programación de bases de datos, destinados a escribir aplicaciones de bases de datos. Muchos se desarrollaron como prototipos de investigación, pero el uso de algunos de ellos se ha extendido ampliamente, como el caso de PL/SQL (Lenguaje de programación/SQL, *Programming Language/SQL*) de Oracle.

Es importante saber que la programación de bases de datos es un tema muy amplio. Hay textos enteros dedicados a cada técnica de programación de bases de datos y cómo esa técnica se materializa en un sistema específico. Siempre se están desarrollando nuevas técnicas y efectuándose cambios en las existentes que luego se incorporan en las versiones y los lenguajes más modernos. Una dificultad adicional al presentar este tema es que, aunque existen unos estándares para SQL, cada fabricante puede introducir variaciones de esos estándares. Por ello, hemos optado por realizar una introducción y una comparativa de las principales técnicas de programación de bases de datos, en lugar de estudiar en detalle un método particular o un sistema. Los ejemplos ofrecidos sirven para ilustrar las principales diferencias a las que un programador podría enfrentarse. Intentaremos utilizar los estándares SQL en nuestra presentación y mostrar ejemplos en lugar de describir un sistema específico. Al hacerlo así, este capítulo puede servir como una introducción, pero debe complementarse con los manuales del sistema o con libros que describan el sistema específico.

Empezaremos nuestra presentación de la programación de bases de datos en la Sección 9.1 con una panorámica de las diferentes técnicas desarrolladas para acceder a una base de datos desde un programa. La Sección 9.2 explica las reglas destinadas a incrustar sentencias SQL en un lenguaje de programación de propósito general; es lo que se conoce como *SQL incrustado*. Esta sección también explica brevemente *SQL dinámico*, en el que las consultas se pueden construir dinámicamente en tiempo de ejecución, y presenta los fundamen-

tos de la variante SQLJ de SQL incrustado, que se desarrolló específicamente para el lenguaje de programación Java. En la Sección 9.3 explicamos la técnica *SQL/CLI* (Interfaz de nivel de llamadas, *Call Level Interface*), en la que se proporciona una biblioteca de procedimientos y funciones para acceder a la base de datos. Se han propuesto varios conjuntos de funciones de biblioteca. El conjunto de funciones SQL/CLI es el del estándar SQL. Otra biblioteca de funciones es *ODBC* (Conectividad de base de datos abierta, *Open Data Base Connectivity*). No describiremos ODBC porque está considerado como el predecesor de SQL/CLI. Una tercera biblioteca de funciones (que no explicaremos) es *JDBC*, que fue desarrollada específicamente para acceder a las bases de datos desde Java. Por último, en la Sección 9.4 describiremos *SQL/PSM* (Módulos almacenados persistentes, *Persistent Stored Modules*), que es una parte del estándar SQL que permite que los módulos de programa (procedimientos y funciones) sean almacenados por el DBMS y accedidos a través de SQL. La Sección 9.5 resume el capítulo.

# 9.1 Programación de bases de datos: problemas y técnicas

Centraremos nuestra atención en las técnicas que se han desarrollado para acceder a las bases de datos desde los programas y, en particular, en el problema de cómo acceder a las bases de datos SQL desde los programas de aplicación. Hasta ahora, nuestra presentación de SQL se ha centrado en las estructuras que ofrece para distintas operaciones con bases de datos, desde la definición del esquema y la especificación de las restricciones, hasta la especificación de consultas, actualizaciones y vistas. La mayoría de los sistemas de bases de datos tienen una **interfaz interactiva** donde se pueden escribir directamente estos comandos SQL para entrar en el sistema de bases de datos. Por ejemplo, en un computador que tenga instalado Oracle RDBMS, el comando SQLPLUS inicia la interfaz interactiva. El usuario puede escribir comandos de SQL o consultas directamente en varias líneas, que deben terminar con un punto y coma y la tecla Intro (es decir, "; <cr>"). Como alternativa, es posible crear un **fichero de comandos** y ejecutarlo desde la interfaz interactiva escribiendo @<*nombrefichero*>. El sistema ejecutará los comandos almacenados en el fichero y mostrará el resultado, si lo hay.

La interfaz interactiva es muy adecuada para la creación del esquema y las restricciones o para las consultas temporales ocasionales. No obstante, en la práctica, la mayoría de las interacciones con una base de datos se ejecutan a través de programas que se han diseñado y probado cuidadosamente. Estos programas se conocen generalmente como **programas de aplicación** o **aplicaciones de bases de datos**, y los usuarios finales los utilizan como *transacciones enlatadas*, como se explica en la Sección 1.4.3. Otro uso común de la programación de bases de datos es acceder a una base de datos a través de un programa de aplicación que implementa una **interfaz web**, por ejemplo, para hacer reservas en una aerolínea. De hecho, la inmensa mayoría de las aplicaciones de comercio electrónico incluyen algunos comandos de acceso a bases de datos. El Capítulo 26 ofrece una panorámica de la programación de bases de datos web utilizando PHP, un lenguaje de *scripting* que últimamente se utiliza ampliamente.

En esta sección, primero veremos una panorámica de las principales metodologías de programación de bases de datos. Después, explicaremos algunos de los problemas que surgen al intentar acceder a una base de datos desde un lenguaje de programación de propósito general, y la secuencia típica de comandos para interactuar con la base de datos desde un programa.

## 9.1.1 Metodologías de programación de bases de datos

Existen varias técnicas para incluir interacciones de bases de datos en los programas de aplicación. A continuación tiene las principales:

1. **Incrustación de comandos de bases de datos en un lenguaje de programación de propósito general.** En esta metodología, las sentencias de base de datos se **incrustan** en el lenguaje de programación *host*, identificándolas con un prefijo especial. Por ejemplo, el prefijo para SQL incrustado es la cade-

na EXEC SQL, que precede a todos los comandos SQL en un programa escrito en el lenguaje *host*.[1] Un **precompilador** o **preprocesador** explora el código fuente para identificar las sentencias de base de datos y las extrae para que el DBMS las procese. En el programa son reemplazadas por llamadas a funciones del código generado por el DBMS. Esta técnica se conoce generalmente como **SQL incrustado**.

2. **Uso de una biblioteca de funciones de bases de datos.** En el lenguaje de programación *host* se dispone de una **biblioteca de funciones** para las llamadas a la base de datos. Por ejemplo, puede haber funciones para conectar con una base de datos, ejecutar una consulta, ejecutar una actualización, etcétera. Los comandos de consulta y actualización de bases de datos, así como cualquier otra información necesaria, se incluyen como parámetros en las llamadas de funciones. Esta metodología ofrece lo que se conoce como **APIs** (**Interfaz de programación de aplicaciones**, *Application Programming Interface*) para acceder a una base de datos desde los programas de aplicación.
3. **Diseño de un lenguaje completamente nuevo.** Un **lenguaje de programación de bases de datos** se diseña desde el principio para que sea compatible con el modelo de base de datos y el lenguaje de consulta. Al lenguaje de programación se le añaden estructuras de programación adicionales, como bucles y sentencias condicionales, para convertirlo en un lenguaje de programación "con todas las de la ley".

En la práctica, las dos primeras metodologías son las más comunes, pues muchas aplicaciones ya están escritas en lenguajes de programación de propósito general pero requieren algo de acceso a bases de datos. La tercera metodología es más apropiada para las aplicaciones que tienen una interacción intensiva con bases de datos. Uno de los principales problemas con las dos primeras metodologías es el *desajuste de impedancia*, que no se da en la tercera metodología.

## 9.1.2 Desajuste de impedancia

**Desajuste de impedancia** es el término que se utiliza para referirse a los problemas derivados de las diferencias entre el modelo de base de datos y el modelo del lenguaje de programación. Por ejemplo, el modelo relacional práctico tiene tres estructuras principales: atributos y sus tipos de datos, tuplas (registros) y tablas (conjuntos o multiconjuntos de registros). El primer problema que se puede dar es que los tipos de datos del lenguaje de programación difieran de los tipos de datos de atributo del modelo de datos. Por tanto, es necesario contar con un enlace por cada lenguaje de programación *host* que especifique para cada tipo de atributo los tipos de lenguaje de programación compatibles. Es necesario tener un **enlace** por cada lenguaje de programación porque los diferentes lenguajes tienen distintos tipos de datos; por ejemplo, los tipos de datos disponibles en C y Java son diferentes, y ambos difieren de los de SQL.

Otro problema es que los resultados de la mayoría de las consultas son conjuntos o multiconjuntos de tuplas, y cada tupla está formada por una secuencia de valores de atributo. En el programa, a menudo es necesario acceder a los valores individuales de tuplas individuales por razones de impresión o procesamiento. Por consiguiente, se hace necesario un enlace para mapear la *estructura de datos del resultado de la consulta*, que es una tabla, a una estructura de datos apropiada del lenguaje de programación. También se necesita un mecanismo de bucle para recorrer las tuplas del resultado de una consulta a fin de acceder a una sola tupla a la vez y para extraer valores individuales de esa tupla. Los valores de atributo extraídos normalmente se copian en variables del programa (distintas y del tipo adecuado) para su posterior procesamiento por parte de éste. Para recorrer las tuplas del resultado de una consulta necesitamos un **cursor** o **variable de iteración**.

El desajuste de impedancia es un problema menor cuando se diseña un lenguaje de programación de bases de datos especial para utilizar el mismo modelo de datos y los mismos tipos de datos que el modelo de base de datos. Un ejemplo de este tipo de lenguaje es PL/SQL de Oracle. Para las bases de datos de objetos, el

[1] A veces se utilizan otros prefijos, pero éste es el más común.

modelo de datos de objeto (consulte el Capítulo 20) es muy parecido al modelo de datos del lenguaje de programación Java, por lo que el desajuste de impedancia se reduce mucho al utilizar Java como lenguaje *host* para acceder a las bases de datos de objetos compatibles con Java. Son varios los lenguajes de programación de bases de datos que se han implementado como prototipos de investigación (consulte la sección "Bibliografía seleccionada").

### 9.1.3 Secuencia típica de interacción en la programación de bases de datos

Cuando un programador o un ingeniero de software escribe un programa que requiere acceso a una base de datos, es muy común que el programa se ejecute en un computador y la base de datos en otro. Como recordará de la Sección 2.5, el modelo cliente/servidor es una arquitectura muy común para acceder a una base de datos, donde un **programa cliente** se encarga de manipular la lógica de la aplicación software, pero incluye algunas llamadas a uno o más **servidores de bases de datos** para acceder o actualizar los datos.[2] Al escribir un programa de este tipo, una secuencia de interacción puede ser la siguiente:

1. Cuando el programa cliente requiere acceso a una base de datos particular, el programa primero debe *establecer* o *abrir* una **conexión** con el servidor de bases de datos. Normalmente, esto implica especificar la dirección de Internet (URL) de la máquina donde se encuentra el servidor de bases de datos, además de proporcionar un nombre de cuenta y una contraseña de inicio de sesión para acceder a la base de datos.
2. Una vez establecida la conexión, el programa puede interactuar con la base de datos emitiendo consultas, actualizaciones y otros comandos de bases de datos. En general, en un programa de aplicación se pueden incluir casi todos los tipos de sentencias SQL.
3. Cuando el programa ya no necesita acceso a una base de datos en particular, debe *terminar* o *cerrar* la conexión con la base de datos.

Un programa puede acceder a varias bases de datos. En algunas metodologías de programación de bases de datos, sólo una conexión puede estar activa en cada momento, mientras que en las otras metodologías se pueden establecer varias conexiones simultáneas.

En las siguientes tres secciones veremos ejemplos de cada una de las tres metodologías de programación de bases de datos. La Sección 9.2 describe cómo se *incrusta* SQL en un lenguaje de programación. La Sección 9.3 explica el uso de las *llamadas a funciones* para acceder a la base de datos, y la Sección 9.4 explica una extensión de SQL denominada SQL/PSM que permite *estructuras de programación de propósito general* para la definición de módulos (procedimientos y funciones) que se almacenan dentro del sistema de bases de datos.[3]

## 9.2 SQL incrustado, SQL dinámico y SQLJ

### 9.2.1 Recuperación de tuplas sencillas con SQL incrustado

En esta sección ofrecemos una panorámica de la incrustación de sentencias SQL en un lenguaje de programación de propósito general, como C, ADA, COBOL o PASCAL. El lenguaje de programación se denomina lenguaje host. La mayoría de las sentencias SQL (incluyendo las definiciones de datos o restricciones, consultas,

[2] Como se explicó en la Sección 2.5, hay arquitecturas de dos y tres capas; para no complicar las cosas, vamos a asumir una arquitectura cliente/servidor de dos capas. En el Capítulo 25 explicaremos otras variaciones de estas arquitecturas.

[3] Aunque SQL/PSM no está considerado un lenguaje de programación auténtico, ilustra cómo se pueden incorporar a SQL las estructuras típicas de la programación de propósito general, como los bucles y las estructuras condicionales.

actualizaciones o definiciones de vistas) pueden incrustarse en un programa de lenguaje host. Una sentencia de SQL incrustado se distingue de las sentencias del lenguaje de programación porque se le añaden como prefijo las palabras clave EXEC SQL para que un **preprocesador** (o **precompilador**) pueda separarlas del código escrito en el lenguaje *host*. Las sentencias SQL se pueden finalizar con un punto y coma (;) o con el END-EXEC correspondiente.

Para ilustrar los conceptos de SQL incrustado, utilizaremos C como lenguaje de programación *host*.[4] Dentro de un comando de SQL incrustado, nos podemos referir a las variables del programa C especialmente declaradas. Son las denominadas **variables compartidas**, porque se utilizan en el programa C y en las sentencias SQL incrustadas. Las variables van prefijadas con dos puntos (:) *cuando aparecen en una sentencia SQL*. De este modo, se distinguen los nombres de las variables del programa de los nombres de las estructuras de esquema de la base de datos, como los atributos y las relaciones. También permite que las variables del programa tengan los mismos nombres que los atributos, puesto que el prefijo de los dos puntos (:) permite distinguirlas en la sentencia SQL. Los nombres de las estructuras del esquema de la base de datos (como los atributos y las relaciones) sólo se pueden utilizar en los comandos SQL, pero las variables del programa compartidas se pueden utilizar en cualquier parte del programa C sin el prefijo de los dos puntos (:).

Vamos a suponer que queremos escribir programas C para procesar la base de datos EMPRESA de la Figura 5.5. Tenemos que declarar variables en el programa que coincidan en tipo con los atributos de la base de datos que el programa procesará. El programador puede elegir los nombres de las variables del programa; pueden o no ser idénticos a los de los atributos. Utilizaremos las variables declaradas en la Figura 9.1 en todos nuestros ejemplos y mostraremos los fragmentos de los programas C sin dichas declaraciones. Las variables compartidas se declaran en la **sección `declare`** del programa, como se muestra en las líneas 1 a 7 de la Figura 9.1.[5] Algunos de los enlaces más comunes de los tipos C con los tipos SQL son los siguientes. Los tipos SQL INTEGER, SMALLINT, REAL y DOUBLE se mapean a los tipos C `long`, `short`, `float` y `double`, respectivamente. Las cadenas de longitud fija y de longitud variable (CHAR[*i*], VARCHAR[*i*]) de SQL se pueden mapear a arrays de caracteres (`char [i+1]`, `varchar [i+1]`) de C, que tienen un carácter más de longitud que el tipo SQL porque las cadenas de C terminan con un carácter NULL (\0), que no forma parte de la propia cadena de caracteres.[6] Aunque `varchar` no es un tipo de datos C estándar, está permitido cuando se utiliza C para la programación de bases de datos SQL.

Observe que los únicos comandos SQL incrustados de la Figura 9.1 son las líneas 1 a 7, que le indican al precompilador que tome nota de los nombres de variable C entre BEGIN DECLARE y END DECLARE porque se pueden incluir en las sentencias SQL incrustadas, con tal de que vayan precedidas por los dos puntos (:). Las líneas 2 a 5 son declaraciones de programa C normales. Las variables de programa C declaradas en las líneas 2 a 5 corresponden a los atributos de las tablas EMPLEADO y DEPARTAMENTO de la base de datos EMPRESA de la Figura 5.5 que se declaró en el DDL de SQL de la Figura 8.1. Las variables declaradas en la línea 6 (SQLCODE y SQLSTATE) se utilizan para comunicar errores y condiciones de excepción entre el sistema de bases de datos y el programa. La línea 0 muestra la variable de programa `loop` que no se utilizará en ninguna sentencia SQL incrustada, por lo que se declara fuera de la sección `declare` de SQL.

**Conexión con la base de datos.** El comando SQL para establecer una conexión a una base de datos tiene la siguiente forma:

**CONNECT TO** <nombre del servidor> **AS** <nombre de la conexión>
**AUTHORIZATION** <nombre de la cuenta de usuario y contraseña> ;

---

[4] Esta explicación también se aplica al lenguaje de programación C++, puesto que no utilizamos ninguna de las características de orientación a objetos, pero nos centramos en el mecanismo de programación de bases de datos.

[5] Los números de línea los utilizamos para facilitar su referencia; estos números no forman parte del código real.

[6] Las cadenas de SQL también se pueden mapear a tipos `char*` de C.

**Figura 9.1.** Variables de programa C que se utilizan en los ejemplos de SQL incrustado E1 y E2.

```
0) int loop ;
1) EXEC SQL BEGIN DECLARE SECTION ;
2) varchar nombredpto [16], nombrepila [16], apellido1 [16], direcc [31] ;
3) char dni [10], fechanac [11], sexo [2] ;
4) float sueldo, subida ;
5) int dno, numdpto ;
6) int SQLCODE ; char SQLSTATE [6] ;
7) EXEC SQL END DECLARE SECTION ;
```

---

En general, como un usuario o un programa puede acceder a varios servidores de bases de datos, se pueden establecer varias conexiones, pero sólo una de ellas puede estar activa en cada momento. El programador o usuario puede utilizar <nombre de la conexión> para cambiar de la conexión activa actual a una diferente utilizando el siguiente comando:

**SET CONNECTION** <nombre de la conexión> ;

Cuando una conexión ya no se necesita, se puede dar por terminada con este comando:

**DISCONNECT** <nombre de la conexión> ;

En los ejemplos de este capítulo asumimos que ya se ha establecido la conexión apropiada con la base de datos EMPRESA, y que es la conexión activa actualmente.

**Comunicación entre el programa y el DBMS utilizando SQLCODE y SQLSTATE.** Las dos **variables de comunicación** especiales que el DBMS utiliza para comunicar las condiciones de excepción y error al programa son SQLCODE y SQLSTATE. La variable **SQLCODE** que aparece en la Figura 9.1 es una variable de tipo entero. Una vez ejecutado cada comando de base de datos, el DBMS devuelve un valor en SQLCODE. El valor 0 indica que el DBMS ejecutó satisfactoriamente la sentencia. SQLCODE > 0 (o, más concretamente, SQLCODE = 100), indica que en el resultado de una consulta ya no hay disponibles más datos (registros). Si SQLCODE < 0, entonces se ha producido algún error. En algunos sistemas (por ejemplo, en el RDBMS de Oracle) SQLCODE es un campo de una estructura de registro denominada SQLCA (área de comunicación de SQL), por lo que se hace referencia a él como SQLCA.SQLCODE. En este caso, la definición de SQLCA debe incluirse en el programa C insertando la siguiente línea:

```
EXEC SQL include SQLCA ;
```

En las últimas versiones del estándar SQL se ha añadido la variable de comunicación **SQLSTATE**, que es una cadena de cinco caracteres. Si su valor es '00000', indica que no se ha producido ningún error o excepción; otros valores indican distintos errores o excepciones. Por ejemplo, '02000' indica 'no hay más datos' al utilizar SQLSTATE. Actualmente, SQLSTATE y SQLCODE están disponibles en el estándar SQL. Es de suponer que muchos de los códigos de error y excepción devueltos en SQLSTATE están normalizados para todos los desarrolladores y plataformas de SQL,[7] mientras que los códigos devueltos en SQLCODE no están normalizados pero están definidos por el desarrollador del DBMS. Por tanto, es recomendable utilizar SQLSTATE porque, de este modo, la manipulación de errores en los programas de aplicación es independiente de un DBMS en particular. Como ejercicio, puede reescribir los ejemplos que se ofrecen más adelante en este capítulo utilizando SQLSTATE en lugar de SQLCODE.

**Ejemplo de programación SQL incrustada.** Nuestro primer ejemplo para ilustrar la programación con SQL incrustado es un fragmento de programa repetitivo (bucle) que lee el número del Documento Nacional

[7] En particular, se supone que los códigos de SQLSTATE que empiezan con los caracteres 0 a 4 o A a H están normalizados, mientras que el resto de valores pueden estar definidos por la implementación.

**Figura 9.2.** Fragmento de programa E1: fragmento de un programa C con SQL incrustado.

```
//Fragmento de programa E1:
0)  loop = 1 ;
1)  while (loop) {
2)        prompt("Introduzca un número de DNI: ", dni) ;
3)        EXEC SQL
4)         select Nombre, Apellido1, Direccion, Sueldo
5)         into :nombrepila, :apellido1, :direcc, :sueldo
6)         from EMPLEADO where Dni = :dni ;
7)        if (SQLCODE == 0) printf(nombrepila, apellido1, direcc, sueldo)
8)         else printf("Este número de DNI no existe: ", dni) ;
9)        prompt("Más números de DNI (introduzca 1 para Sí, 0 para No): ", loop) ;
10)       }
```

de Identidad de un empleado e imprime algo de información correspondiente al registro EMPLEADO de la base de datos. En la Figura 9.2 se muestra el código del fragmento del programa C (E1). El programa lee (entrada) un valor Dni y después recupera de la base de datos la tupla de EMPLEADO que tiene ese Dni, utilizando SQL incrustado. **INTO CLAUSE** (línea 5) especifica las variables de programa en las que se recuperan los valores de los atributos de la base de datos. Las variables del programa C de la cláusula INTO tienen como prefijo el símbolo de los dos puntos (:), como explicamos anteriormente.

La línea 7 del fragmento E1 ilustra la comunicación entre la base de datos y el programa a través de la variable especial SQLCODE. Si el valor devuelto por el DBMS en SQLCODE es 0, la sentencia anterior se ejecutaría sin condiciones de error o excepción. La línea 7 comprueba esto y asume que si se ha producido un error, es porque no existe una tupla de EMPLEADO con el Dni dado; por consiguiente, ofrece como salida un mensaje al respecto (línea 8).

En el fragmento E1, la consulta SQL incrustada sólo selecciona una tupla; por eso podemos asignar los valores de sus atributos directamente a las variables del programa C de la cláusula INTO de la línea 5. En general, una consulta SQL puede recuperar muchas tuplas. En tal caso, el programa C normalmente recorrerá las tuplas recuperadas y las procesará una por una, para lo que se utiliza un *cursor*. A continuación describiremos los cursores.

### 9.2.2 Recuperación de varias tuplas con SQL incrustado y utilizando cursores

Podemos pensar en un **cursor** como en un puntero que apunta a *una sola tupla (fila)* del resultado de una consulta que recupera varias tuplas. El cursor se declara cuando se declara el comando de consulta SQL en el programa. Más tarde en el programa, un comando **OPEN CURSOR** toma el resultado de la consulta de la base de datos y establece el cursor a una posición *anterior a la primera fila* de dicho resultado. Ésta se convierte en la **fila actual** para el cursor. A continuación, se ejecutan comandos **FETCH** en el programa; cada uno mueve el cursor a la *siguiente fila* del resultado de la consulta, convirtiéndola en la fila actual y copiando los valores de sus atributos en las variables del programa C (lenguaje *host*) especificadas en el comando FETCH mediante una cláusula INTO. La variable de cursor es básicamente un **iterador** que itera por las tuplas del resultado de la consulta (una tupla a la vez). Esto es parecido al procesamiento tradicional por registros.

Para determinar cuándo se han procesado todas las tuplas del resultado de la consulta, se comprueba la variable de comunicación SQLCODE (o, como alternativa, SQLSTATE). Si se ejecuta un comando FETCH y como resultado se mueve el cursor más allá de la última tupla del resultado de la consulta, se devuelve un valor positivo (SQLCODE > 0) en SQLCODE, indicando que no se han encontrado datos (una tupla) (o se devuelve la cadena ‘02000’ en SQLSTATE). Esto se utiliza para terminar el bucle por las tuplas del resultado de la con-

**Figura 9.3.** Fragmento de programa E2: fragmento de un programa C que utiliza cursores con SQL incrustado con propósitos de actualización.

```
    //Fragmento de programa E2:
0)  prompt("Introduzca el nombre del departamento: ", nombredpto) ;
1)  EXEC SQL
2)       select NumeroDpto into :numdpto
3)       from DEPARTAMENTO where NombreDpto = :nombredpto ;
4)  EXEC SQL DECLARE EMP CURSOR FOR
5)       select Dni, Nombre, Apellido1, Sueldo
6)       from EMPLEADO where Dno = :numdpto
7)       FOR UPDATE OF Sueldo ;
8)  EXEC SQL OPEN EMP ;
9)  EXEC SQL FETCH from EMP into :dni, :nombrepila, :apellido1, :sueldo ;
10) while (SQLCODE == 0) {
11)      printf("Nombre del empleado: ", Nombre, Apellido1) ;
12)      prompt("Introduzca la subida del sueldo: ", subida) ;
13)      EXEC SQL
14)            update EMPLEADO
15)            set Sueldo = Sueldo + :subida
16)            where CURRENT OF EMP ;
17)      EXEC SQL FETCH from EMP into :dni, :nombrepila, :apellido1, :sueldo ;
18)      }
19) EXEC SQL CLOSE EMP ;
```

sulta. En general, se pueden abrir numerosos cursores al mismo tiempo. Se ejecuta un comando **CLOSE CURSOR** para indicar que se ha terminado el procesamiento del resultado de la consulta asociada al cursor.

En la Figura 9.3 puede ver un ejemplo del uso de los cursores; en la línea 4 se declara un cursor denominado EMP. Este cursor está asociado con la consulta SQL declarada en las líneas 5 a 6, pero la consulta no se ejecuta hasta haberse procesado el comando OPEN EMP (línea 8). El comando OPEN <nombre de cursor> ejecuta la consulta y toma su resultado como una tabla en el espacio de trabajo del programa, donde el programa puede efectuar un bucle por las filas (tuplas) individuales mediante subsiguientes comandos FETCH <nombre de cursor> (línea 9). Asumimos que se han declarado las variables de programa C adecuadas, como en la Figura 9.1. El fragmento de programa E2 lee el nombre de un departamento (línea 0), recupera su número de departamento (líneas 1 a 3) y, después, recupera los empleados que trabajan en ese departamento a través de un cursor. Un bucle (líneas 10 a 18) itera por cada registro de empleado, uno a la vez, e imprime el nombre del empleado. El programa lee después un aumento de sueldo para ese empleado (línea 12) y actualiza su sueldo en la base de datos (líneas 14 a 16).

Cuando se define un cursor para las filas que se van a modificar (actualizar), debemos añadir una cláusula **FOR UPDATE OF** en la declaración del cursor y listar los nombres de los atributos que el programa actualizará. Es lo que se ilustra en la línea 7 del segmento de código E2. Si se van a eliminar las filas, hay que añadir las palabras clave **FOR UPDATE** sin especificar atributo alguno. En el comando UPDATE (o DELETE) incrustado, la condición **WHERE CURRENT OF** <nombre de cursor> especifica que la tupla actual indicada por el cursor es la única que se va a actualizar (o borrar), como en la línea 16 de E2.

La declaración de un cursor y su asociación con una consulta (líneas 4 a 7 de E2) no ejecuta la consulta; la consulta sólo se ejecuta cuando se ejecuta el comando OPEN <nombre de cursor> (línea 8). Además, no es necesario incluir la cláusula **FOR UPDATE OF** de la línea 7 de E2 si el resultado de la consulta sólo se va a utilizar *con propósitos de recuperación* (y no para actualización o borrado).

Con la declaración de un cursor se pueden especificar varias opciones. La forma general de la declaración de un cursor es la siguiente:

**DECLARE** <nombre de cursor> **[ INSENSITIVE ] [ SCROLL ] CURSOR**
[ **WITH HOLD** ] **FOR** <especificación de consulta>
[ **ORDER BY** <especificación de orden> ]
[ **FOR READ ONLY** | **FOR UPDATE** [ **OF** <lista de atributos> ] ] ;

Ya hemos explicado brevemente las opciones de la última línea. Lo predeterminado es que la consulta sea para recuperación (FOR READ ONLY). Si alguna de las tuplas del resultado de la consulta debe actualizarse, tenemos que especificar FOR UPDATE OF <lista de atributos> y enumerar los atributos que han de actualizarse. Si es preciso borrar algunas tuplas, tenemos que especificar FOR UPDATE sin la lista de atributos.

Cuando en la declaración de un cursor se especifica la palabra clave SCROLL opcional, es posible colocar el cursor de otras formas que únicamente para un acceso puramente secuencial. Al comando FETCH se le puede añadir una **orientación de extracción**, cuyo valor puede ser NEXT, PRIOR, FIRST, LAST, ABSOLUTE *i* y RELATIVE *i*. En los dos últimos comandos, *i* debe evaluarse como un valor entero que especifica la posición absoluta o la posición relativa de una tupla respecto a la posición actual del cursor, respectivamente. La orientación de extracción predeterminada, que hemos utilizado en nuestros ejemplos, es NEXT. La orientación permite al programador mover el cursor por las tuplas del resultado de la consulta con una flexibilidad mayor, proporcionando un acceso aleatorio por posición o un acceso en orden inverso. Cuando en el cursor se especifica SCROLL, la forma general de un comando FETCH es la siguiente, teniendo en cuenta que las partes que aparecen entre corchetes son opcionales:

**FETCH** [ [ <orientación de extracción> ] **FROM** ] <nombre de cursor>
**INTO** <lista de destino de la extracción> ;

La cláusula ORDER BY ordena las tuplas para que el comando FETCH las extraiga en el orden especificado. Se especifica de un modo parecido a la cláusula correspondiente de las consultas SQL (consulte la Sección 8.4.6). Las últimas dos opciones cuando se declara un cursor (INSENSITIVE y WITH HOLD) se refieren a las características de transacción de los programas de bases de datos, que explicaremos en el Capítulo 17.

### 9.2.3 Especificación de consultas en tiempo de ejecución con SQL dinámico

En los ejercicios anteriores, las consultas SQL incrustadas se escribían como parte del código fuente del programa *host*. Por tanto, cuando queremos escribir una consulta diferente, debemos escribir un programa nuevo, y pasar por todos los pasos pertinentes (compilación, depuración, prueba, etcétera). En algunos casos, es conveniente escribir un programa que pueda ejecutar diferentes consultas o actualizaciones SQL (u otras operaciones) *dinámicamente en tiempo de ejecución*. Por ejemplo, queremos escribir un programa que acepte una consulta SQL escrita desde el monitor, la ejecute y muestre su resultado, como si se tratara de las interfaces interactivas disponibles en la mayoría de los DBMSs. Otro ejemplo es cuando una interfaz amigable genera consultas SQL dinámicamente para el usuario basándose en operaciones "apuntar y clic" en un esquema gráfico (por ejemplo, una interfaz parecida a QBE; consulte el Apéndice D). En esta sección ofrecemos una breve panorámica de cómo funciona **SQL dinámico**, que es una técnica para escribir este tipo de programas de bases de datos.

El fragmento de código E3 de la Figura 9.4 lee una cadena introducida por el usuario (debe tratarse de un comando de actualización SQL) en la variable de cadena `sqlupdatestring` de la línea 3. Después, se prepara esto en la línea 4 como un comando SQL asociándolo con la variable SQL `sqlcommand`. La línea 5 ejecuta el comando. En este caso, como estamos en *tiempo de compilación*, no es posible comprobar la sintaxis ni hacer otras comprobaciones, pues el comando no está disponible hasta el momento de la ejecución. Esto contrasta con los ejemplos anteriores de SQL incrustado, en los que la consulta se comprueba en tiempo de compilación porque su texto está en el código fuente del programa.

**Figura 9.4.** Fragmento de programa E3: fragmento de programa C que utiliza SQL dinámico para actualizar una tabla.

```
    //Fragmento de programa E3:
0)  EXEC SQL BEGIN DECLARE SECTION ;
1)  varchar sqlupdatestring [256] ;
2)  EXEC SQL END DECLARE SECTION ;
    ...
3)  prompt("Introduzca el comando de actualización: ", sqlupdatestring) ;
4)  EXEC SQL PREPARE sqlcommand FROM :sqlupdatestring ;
5)  EXEC SQL EXECUTE sqlcommand ;
    ...
```

Aunque la inclusión de un comando de actualización dinámico es relativamente directo en SQL dinámico, una consulta dinámica es mucho más compleja. Esto es porque, al escribir el programa, generalmente no conocemos el tipo o la cantidad de atributos que serán recuperados por la consulta SQL. A veces se necesita una estructura de datos compleja para permitir diferentes cantidades y tipos de atributos en el resultado de la consulta en caso de no tener información sobre la consulta dinámica. Es posible utilizar técnicas parecidas a éstas para asignar resultados de consultas (y parámetros de consultas) a variables del programa *host* (consulte la Sección 9.3).

En E3, la razón de separar PREPARE y EXECUTE es que si el comando va a ejecutarse varias veces en un programa, debe prepararse sólo una vez. La preparación del comando generalmente implica que el sistema compruebe la sintaxis y realice otros tipos de comprobaciones, así como que genere el código para ejecutar dicho comando. Es posible combinar los comandos PREPARE y EXECUTE (líneas 4 y 5 de E3) en una sola sentencia escribiendo lo siguiente:

```
EXEC SQL EXECUTE IMMEDIATE :sqlupdatestring ;
```

Esto resulta útil si el comando sólo se va a ejecutar una vez. De forma alternativa, se pueden separar los dos para capturar los errores que se pudieran producir después de la sentencia PREPARE.

### 9.2.4 SQLJ: comandos SQL incrustados en Java

En las secciones anteriores ofrecimos una panorámica de cómo se pueden incrustar comandos SQL en un lenguaje de programación tradicional, y para ello utilizamos C. Ahora centraremos nuestra atención en cómo puede incrustarse SQL en un lenguaje de programación orientado a objetos,[8] en particular, el lenguaje Java. SQLJ es un estándar por el que han apostado varios desarrolladores para incrustar SQL en Java.

Históricamente, SQLJ se desarrolló después que JDBC, que se utiliza para acceder a las bases de datos SQL desde Java utilizando llamadas a funciones. En la Sección 9.3.2 hablaremos de JDBC. Nos centraremos en cómo se utiliza SQLJ en el RDBMS de Oracle. Generalmente, un intérprete de SQLJ convierte las sentencias de SQL en Java, para que luego puedan ejecutarse a través de la interfaz JDBC. Por tanto, es preciso instalar un *controlador JDBC* para poder utilizar SQLJ.[9] En esta sección nos centraremos en cómo utilizar los conceptos de SQLJ para escribir SQL incrustado en un programa de Java.

Antes de poder procesar SQLJ con Java en Oracle, hay que importar varias bibliotecas de clases (véase la Figura 9.5), entre las que se incluyen las clases JDBC y de E/S (líneas 1 y 2), además de las clases adicionales enumeradas en las líneas 3, 4 y 5. Además, el programa debe conectar primero con la base de datos deseada utilizando una llamada a la función `getConnection`, que es uno de los métodos de la clase `oracle` de la

[8] Esta sección asume que está familiarizado con los conceptos de orientación a objetos y con los conceptos básicos de JAVA. Si no es así, deje esta sección para cuando haya leído el Capítulo 20.

[9] En la Sección 9.3.2 explicamos los controladores JDBC.

**Figura 9.5.** Importación de las clases necesarias para incluir SQLJ en los programas de Java en Oracle, y establecimiento de una conexión y un contexto predeterminado.

```
1)  import java.sql.* ;
2)  import java.io.* ;
3)  import sqlj.runtime.* ;
4)  import sqlj.runtime.ref.* ;
5)  import oracle.sqlj.runtime.* ;
    ...
6)  DefaultContext cntxt =
7)     oracle.getConnection("<nombre de url>","<nombre del usuario>","<contraseña>",true);
8)  DefaultContext.setDefaultContext(cntxt) ;
    ...
```

**Figura 9.6.** Las variables de programa Java utilizadas en los ejemplos J1 y J2 de SQLJ.

```
1)  string nombredpto, dni , nombrepila, np, apellido1, ap1, fechanac, direcc ;
2)  char sexo ;
3)  double sueldo, sal ;
4)  integer dno, numdpto ;
```

línea 5 de la Figura 9.5. El formato de esta llamada, que devuelve un objeto de tipo *contexto predeterminado*,[10] es el siguiente:

```
public static DefaultContext
getConnection(String url, String user, String password,
   Boolean autoCommit)
throws SQLException ;
```

Por ejemplo, podemos escribir las sentencias de las líneas 6 a 8 de la Figura 9.5 para conectar con una base de datos de Oracle ubicada en el URL <nombre de url> utilizando el inicio de sesión <nombre del usuario> y <contraseña> con el envío automático de cada comando,[11] y después establecer esta conexión como **contexto predeterminado** para los comandos subsiguientes.

En los siguientes ejemplos no mostramos las clases o los programas de Java completos, porque nuestra intención no es enseñar Java; sólo mostraremos los fragmentos de programa que ilustran el uso de SQLJ. La Figura 9.6 muestra las variables de programa Java que se utilizan en los ejemplos. El fragmento de programa J1 de la Figura 9.7 lee el DNI de un empleado e imprime información de éste extraída de la base de datos.

Observe que como Java ya utiliza el concepto de **excepciones** para la manipulación de errores, se utiliza una excepción especial denominada `SQLException` para devolver los errores o las condiciones de excepción después de ejecutar un comando de bases de datos SQL. Esto juega un papel parecido a SQLCODE y SQLSTATE en SQL incrustado. Java tiene muchos tipos de excepciones predefinidas. Cada operación (función) Java debe especificar las excepciones que se pueden lanzar; es decir, las condiciones de excepción que se pueden dar al ejecutar el código Java de esa operación. Si se produce una excepción definida, el sistema transfiere el control al código Java especificado para la manipulación de las excepciones. En J1, la manipulación de excepciones para una `SQLException` se especifica en las líneas 7 y 8. En Java, se utiliza la siguiente estructura:

[10] Al establecerse, un *contexto predeterminado* se aplica a los comandos subsiguientes del programa hasta que cambia.

[11] La *confirmación automática* significa, a grandes rasgos, que cada comando se aplica a la base de datos después de haberse ejecutado. La alternativa es que el programador quiera ejecutar varios comandos relacionados con las bases de datos y después enviarlos todos juntos. En el Capítulo 17 explicamos los conceptos de confirmación, al describir las transacciones de bases de datos.

**Figura 9.7.** Fragmento de programa J1: fragmento de programa Java con SQLJ.

```
    //Fragmento de programa J1:
1)  dni = readEntry("Introduzca un número de DNI: ") ;
2)  try {
3)     #sql { select Nombre, Apellido1, Direccion, Sueldo
4)          into :nombrepila, :apellido1, :direcc, :sueldo
5)          from EMPLEADO where Dni = :dni} ;
6)  } catch (SQLException se) {
7)          System.out.println("Este número de DNI no existe: " + dni) ;
8)          Return ;
9)     }
10) System.out.println(nombrepila + " " + apellido1 + " " + direcc + " " + sueldo)
```

---

```
try {<operación>} catch (<excepción>) {<código de manipulación de excepciones>}
        <continuación del código>
```

para tratar las excepciones que se producen durante la ejecución de `<operación>`. Si no se producen excepciones, se procesa directamente `<continuación del código>`. Las excepciones que el código puede lanzar en una operación particular deben especificarse como parte de la declaración de operación o *interfaz*, por ejemplo, con este formato:

```
<tipo devuelto por la operación> <nombre de la operación>(<parámetros>)
   throws SQLException, IOException ;
```

En SQLJ, los comandos de SQL incrustados dentro de un programa de Java van precedidos por `#sql`, como se muestra en la línea 3 de J1, a fin de ser identificados por el preprocesador. SQLJ utiliza una *cláusula INTO* (parecida a la que se utiliza en SQL incrustado) para devolver los valores de atributo recuperados de la base de datos por una consulta SQL en las variables del programa Java. Estas últimas van precedidas por los dos puntos (:) en la sentencia SQL, como en SQL incrustado.

En J1, la consulta SQLJ incrustada sólo selecciona una tupla; por eso podemos asignar los valores de sus atributos directamente a las variables del programa Java en la cláusula INTO de la línea 4. Para las consultas que recuperan muchas tuplas, SQLJ utiliza el concepto de *iterador*, que es algo parecido a un cursor en SQL incrustado.

### 9.2.5 Recuperación de varias tuplas en SQLJ utilizando iteradores

En SQLJ, un **iterador** es un tipo de objeto asociado con una colección (conjunto o multiconjunto) de tuplas del resultado de una consulta.[12] El iterador está asociado con las tuplas y los atributos que aparecen en el resultado de una consulta. Hay dos tipos de iteradores:

1. Un **iterador con nombre** está asociado con el resultado de una consulta enumerando los *nombres y los tipos de los atributos* que aparecen en dicho resultado. Los nombres de atributo deben corresponderse con las variables del programa Java, como se muestra en la Figura 9.6.
2. Un **iterador posicional** sólo enumera los *tipos de atributos* que aparecen en el resultado de la consulta.

En los dos casos, la lista debe tener el *mismo orden* que los atributos de la cláusula SELECT de la consulta. Sin embargo, el bucle por el resultado de una consulta es diferente para los dos tipos de iteradores, como veremos. En primer lugar, veremos en la Figura 9.8 (fragmento J2A) el uso de un iterador con nombre. La línea 9 de la Figura 9.8 muestra la declaración del iterador con nombre `Emp`. Los nombres de los atributos en un

[12] En el Capítulo 21 explicamos los iteradotes más en profundidad, al hablar de las bases de datos de objetos.

**Figura 9.8.** Fragmento de programa J2A: fragmento de programa Java que utiliza un iterador con nombre para imprimir información de un empleado de un departamento en particular.

```
    //Fragmento de pograma J2A:
0)  nombredpto = readEntry("Introduzca el nombre del departamento: ") ;
1)  try {
2)    #sql { select NumeroDpto into :numdpto
3)         from DEPARTAMENTO where NombreDpto = :nombredpto} ;
4)  } catch (SQLException se) {
5)     System.out.println("El departamento no existe: " + nombredpto) ;
6)     Return ;
7)     }
8)  System.out.printline("Información del empleado de este departamento: " + nombredpto) ;
9)  #sql iterator Emp(String dni, String nombrepila, String apellido1, double sueldo) ;
10) Emp e = null ;
11) #sql e = { select dni, nombrepila, apellido1, sueldo
12)      from EMPLEADO where Dno = :numdpto} ;
13) while (e.next()) {
14)       System.out.printline(e.dni + " " + e.nombrepila + " " + e.apellido1 + " " +
                 e.sueldo) ;
15) } ;
16) e.close() ;
```

tipo de iterador con nombre deben coincidir con los nombres de los atributos del resultado de la consulta SQL. La línea 10 muestra cómo se crea un objeto iterador `e` de tipo `Emp` y se asocia con una consulta (líneas 11 y 12).

Cuando el objeto iterador está asociado con una consulta (líneas 11 y 12 de la Figura 9.8), el programa extrae el resultado de la consulta de la base de datos y establece el iterador a una posición *anterior a la primera fila* de ese resultado, que se convierte en la **fila actual** del iterador. Posteriormente, se pueden emitir operaciones **`next`** en el iterador; cada una lo mueve a la *siguiente fila* del resultado de la consulta, que se convierte en la fila actual. Si la fila existe, la operación recupera los valores de atributo de esa fila en las correspondientes variables del programa. Si ya no existen más filas, la siguiente operación devuelve NULL, por lo que podemos utilizar este valor para controlar el bucle. El iterador con nombre no necesita una cláusula INTO porque las variables del programa correspondientes a los atributos recuperados ya están especificadas cuando se declara el tipo iterador (línea 9 de la Figura 9.8).

En la Figura 9.8, el comando `(e.next())` de la línea 13 lleva a cabo dos funciones: obtiene la siguiente tupla del resultado de la consulta y controla el bucle `while`. Una vez terminado el resultado de la consulta, el comando `e.close()` (línea 16) cierra el iterador.

A continuación, vamos a ver el mismo ejemplo utilizando los *iteradores posicionales*, como se muestra en la Figura 9.9 (fragmento de programa J2B). La línea 9 de la Figura 9.9 muestra la declaración del tipo iterador posicional `Emppos`. La principal diferencia entre un iterador posicional y uno con nombre es que en el primero no hay atributos (correspondientes a los nombres de las variables del programa), sólo tipos de atributos. Aun así, deben ser compatibles con los tipos de los atributos del resultado de la consulta SQL y aparecer en el mismo orden. La línea 10 muestra cómo se crea en el programa la variable de iterador posicional `e` del tipo `Emppos` y después se asocia con una consulta (líneas 11 y 12).

El iterador posicional se comporta de una forma que es más parecida a SQL incrustado (consulte la Sección 9.2.2). Se necesita un comando **FETCH** <**variable iterador**> **INTO** <**variables de programa**> para obtener la siguiente tupla del resultado de una consulta. La primera vez que se ejecuta `fetch`, se obtiene la primera tupla (línea 13 de la Figura 9.9). La línea 16 obtiene la siguiente tupla hasta que no haya más tuplas en el resultado de la consulta. Para controlar el bucle, se utiliza una función de iterador posicional, `e.endFetch()`. Esta

**Figura 9.9.** Fragmento de programa J2B: fragmento de programa Java que utiliza un iterador posicional para imprimir la información de un empleado de un departamento en particular.

```
    //Fragmento de programa J2B:
0)  nombredpto = readEntry("Introduzca el nombre del departamento: ") ;
1)  try {
2)     #sql { select NumeroDpto into :numdpto
3)          from DEPARTAMENTO where NombreDpto = :nombredpto} ;
4)  } catch (SQLException se) {
5)    System.out.println("El departamento no existe: " + nombredpto) ;
6)    Return ;
7)  }
8)  System.out.println("Información del empleado de este departamento: " + nombredpto) ;
9)  #sql iterator Emppos(String, String, String, double) ;
10) Emppos e = null ;
11) #sql e = { select dni, nombrepila, apellido1, sueldo
12)      from EMPLEADO where Dno = :numdpto} ;
13) #sql { fetch :e into :dni, :np, :ap1, :sal} ;
14) while (!e.endFetch()) {
15)      System.out.println(dni + " " + np + " " + ap1 + " " + sal) ;
16)      #sql { fetch :e into :dni, :np, :ap1, :sal} ;
17) } ;
18) e.close() ;
```

función se establece a TRUE cuando el iterador está inicialmente asociado con una consulta SQL (línea 11), y se establece a FALSE cada vez que un comando `fetch` devuelve una tupla válida del resultado de la consulta. Se establece de nuevo a TRUE cuando un comando `fetch` ya no encuentra más tuplas. La línea 14 muestra cómo se controla el bucle mediante la negación.

# 9.3 Programación de bases de datos con llamadas a funciones: SQL/CLI y JDBC

A veces, a SQL incrustado (consulte la Sección 9.2) se le denomina metodología **estática** de programación de bases de datos porque el texto de la consulta se escribe dentro del programa y no puede modificarse sin tener que recompilar o reprocesar el código fuente. El uso de llamadas a funciones es una metodología más **dinámica** de programación de bases de datos que SQL incrustado. Ya hemos visto una técnica de este tipo de programación, SQL dinámico, en la Sección 9.2.3. Las técnicas aquí explicadas proporcionan otra metodología de programación dinámica de bases de datos. Se utiliza una **biblioteca de funciones**, también conocida como **interfaz de programación de aplicaciones (API)**, para acceder a la base de datos. Aunque esto ofrece mayor flexibilidad porque no se necesita ningún preprocesador, su inconveniente es que la verificación de la sintaxis y otras comprobaciones de los comandos de SQL han de llevarse a cabo en tiempo de ejecución. Otro inconveniente es que a veces se necesita una programación más compleja para acceder al resultado de la consulta, porque puede que no se conozcan con antelación los tipos y la cantidad de atributos del mismo.

En esta sección ofrecemos una panorámica de dos interfaces de llamadas a funciones. En primer lugar, explicaremos la **Interfaz de nivel de llamadas** (**SQL/CLI**, ***Call Level Interface***), que es parte del estándar SQL. Se desarrolló como una continuación de la técnica anterior conocida como ODBC (Conectividad de bases de datos abierta). En los ejemplos SQL/CLI utilizamos C como lenguaje *host*. Después, ofrecemos una panorámica de **JDBC**, que es la interfaz de llamadas a funciones, para acceder a las bases de datos desde Java. Aunque es muy común asumir que JDBC significa Conectividad de bases de datos de Java (*Java Database Connectivity*), JDBC es realmente una marca registrada de Sun Microsystems, *no un acrónimo*.

La principal ventaja del uso de una interfaz de llamadas a funciones es que facilita el acceso a varias bases de datos dentro del mismo programa de aplicación, aunque estén almacenadas en diferentes paquetes DBMS. Lo explicaremos más tarde, en la Sección 9.3.2, cuando hablemos de la programación de bases de datos Java con JDBC, aunque esta ventaja también se aplica a la programación de bases de datos con SQL/CLI y ODBC (consulte la Sección 9.3.1).

### 9.3.1 Programación de bases de datos con SQL/CLI utilizando C como lenguaje *host*

Antes de utilizar las llamadas a funciones en SQL/CLI, es necesario instalar las bibliotecas apropiadas en el servidor de bases de datos. Estos paquetes se obtienen del desarrollador del DBMS que se utiliza. Vamos a ver ahora una panorámica de cómo se puede utilizar SQL/CLI en un programa C.[13] Ilustraremos nuestra presentación con el fragmento de programa CLI1 de la Figura 9.10.

Al utilizar SQL/CLI, las sentencias SQL se crean dinámicamente y se pasan como parámetros de cadena en las llamadas a las funciones. Por tanto, es necesario hacer un seguimiento de la información sobre las interacciones del programa *host* con la base de datos en estructuras de datos en tiempo de ejecución, porque los comandos de bases de datos se procesan en tiempo de ejecución. La información se guarda en cuatro tipos de registros, representados como *structs* en los tipos de datos C. Se utiliza un **registro de entorno** como contenedor para rastrear una o más conexiones de bases de datos y para establecer la información de entorno. Un **registro de conexión** rastrea la información necesaria para una conexión de base de datos en particular.

**Figura 9.10.** Fragmento de programa CLI1: fragmento de un programa C con SQL/CLI.

```
//Programa CLI1:
0)  #include sqlcli.h ;
1)  void printSal() {
2)  SQLHSTMT stmt1 ;
3)  SQLHDBC con1 ;
4)  SQLHENV env1 ;
5)  SQLRETURN ret1, ret2, ret3, ret4 ;
6)  ret1 = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env1) ;
7)  if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1, &con1) else exit ;
8)  if (!ret2) ret3 = SQLConnect(con1, "dbs", SQL_NTS, "js", SQL_NTS, "xyz", SQL_NTS)
         else exit ;
9)  if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1, &stmt1) else exit ;
10) SQLPrepare(stmt1, "select Apellido1, Sueldo from EMPLEADO where Dni = ?", SQL_NTS) ;
11) prompt("Introduzca un número de DNI: ", dni) ;
12) SQLBindParameter(stmt1, 1, SQL_CHAR, &dni, 9, &fetchlen1) ;
13) ret1 = SQLExecute(stmt1) ;
14) if (!ret1) {
15)   SQLBindCol(stmt1, 1, SQL_CHAR, &apellido1, 15, &fetchlen1) ;
16)   SQLBindCol(stmt1, 2, SQL_FLOAT, &sueldo, 4, &fetchlen2) ;
17)   ret2 = SQLFetch(stmt1) ;
18)   if (!ret2) printf(dni, apellido1, sueldo)
19)     else printf("Este número de DNI no existe: ", dni) ;
20)   }
21) }
```

[13] Nuestra explicación también se puede aplicar al lenguaje de programación C++, ya que no utilizamos ninguna de las características de orientación a objetos, sino que nos centramos en el mecanismo de programación de bases de datos.

Un **registro de sentencia** mantiene la información necesaria para una sentencia SQL. Un **registro de descripción** mantiene la información sobre las tuplas o parámetros; por ejemplo, la cantidad de atributos y sus tipos de una tupla, o la cantidad y los tipos de parámetros de una llamada a una función.

Cada registro es accesible al programa a través de una variable puntero de C, denominada **manejador**. El manejador es devuelto cuando se crea primero un registro. Para crear un registro y devolver su manejador, se utiliza la siguiente función SQL/CLI:

```
SQLAllocHandle(<tipo_manejador>, <manejador_1>, <manejador_2>)
```

Los parámetros de esta función son los siguientes:

- `<tipo_manejador>` indica el tipo de registro que se crea. Los posibles valores para este parámetro son `SQL_HANDLE_ENV`, `SQL_HANDLE_DBC`, `SQL_HANDLE_STMT` o `SQL_HANDLE_DESC`, para un registro de entorno, conexión, sentencia o descripción, respectivamente.
- `<manejador_1>` indica el contenedor en el que se está creando el manejador nuevo. Por ejemplo, para un registro de conexión sería el entorno en el que se está creando la conexión, y para un registro de sentencia sería la conexión para esa sentencia.
- `<manejador_2>` es el puntero (manejador) al registro de tipo `<tipo_manejador>` recién creado.

Al escribir un programa C que incluirá llamadas de bases de datos a través de SQL/CLI, se suelen dar los siguientes pasos. Los ilustramos haciendo referencia al ejemplo CLI1 de la Figura 9.10, que lee un número del Documento Nacional de Identidad de un empleado e imprime el primer apellido y el sueldo de ese empleado.

1. En el programa C debe incluirse la *biblioteca de funciones* que abarca SQL/CLI. Se llama `sqlcli.h`, y se incluye en la línea 0 de la Figura 9.10.
2. Declare las *variables de manejador* de los tipos `SQLHSTMT`, `SQLHDBC`, `SQLHENV` y `SQLHDESC` para las sentencias, conexiones, entornos y descripciones necesarias en el programa, respectivamente (líneas 2 a 4).[14] Además, declare variables del tipo `SQLRETURN` (línea 5) para guardar los códigos devueltos por las llamadas a las funciones SQL/CLI. El código 0 (cero) indica que la *ejecución de la llamada a la función ha sido satisfactoria*.
3. En el programa debe configurarse un *registro de entorno* con `SQLAllocHandle`. La función para hacer esto se muestra en la línea 6. Como en ningún otro registro se guarda un registro de entorno, el parámetro `<manejador_1>` es el manejador NULL SQL_NULL_HANDLE (puntero NULL) al crear un entorno. El manejador (puntero) al registro de entorno recién creado se devuelve en la variable `env1` de la línea 6.
4. En el programa se configura un *registro de conexión* utilizando `SQLAllocHandle`. En la línea 7, el registro de conexión creado tiene el manejador `con1` y está contenido en el entorno `env1`. Se establece entonces una **conexión** en `con1` con un servidor concreto de bases de datos utilizando la función `SQLConnect` de SQL/CLI (línea 8). En nuestro ejemplo, el nombre del servidor de bases de datos con el que estamos conectando es `dbs` y el nombre de cuenta y la contraseña para el inicio de sesión son `js` y `xyz`, respectivamente.
5. En el programa se configura un *registro de sentencia* utilizando `SQLAllocHandle`. En la línea 9, el registro de sentencia creado tiene el manejador `stmt1` y utiliza la conexión `con1`.
6. La sentencia se ha *preparado* utilizando la función SQL/CLI `SQLPrepare`. En la línea 10, esto asigna la cadena de sentencia SQL (la consulta en nuestro ejemplo) a la sentencia `handle stmt1`. El inte-

[14] Para que la presentación siga siendo sencilla, no mostraremos aquí los registros de descripción.

rrogante (?) de la línea 10 representa un **parámetro de sentencia**, que es un valor que se determina en tiempo de ejecución (normalmente, enlazándolo con una variable del programa C). En general, podría haber varios parámetros, que se distinguen por el orden de aparición de los interrogantes en la sentencia (el primer ? representa el parámetro 1, el segundo ? representa el parámetro 2, etcétera). El último parámetro de `SQLPrepare` debe tener la longitud de la cadena de la sentencia SQL en bytes, pero si introducimos la palabra clave `SQL_NTS`, indicamos que la cadena que alberga la consulta es una *cadena terminada en NULL*, de modo que SQL puede calcular automáticamente la longitud de la cadena. Esto también se aplica a los otros parámetros de cadena de las llamadas a funciones.

7. Antes de ejecutar la consulta, debe enlazarse cualquier parámetro de la cadena de consulta con las variables del programa utilizando la función SQL/CLI `SQLBindParameter`. En la Figura 9.10, el parámetro (indicado por ?) a la consulta preparada referenciada por `stmt1` se enlaza con la variable del programa C `dni` en la línea 12. Si hay *n* parámetros en la sentencia SQL, debemos tener *n* llamadas a la función `SQLBindParameter`, cada una con una posición de parámetro diferente (1, 2, . . ., *n*).

8. Siguiendo estas preparaciones, ahora podemos ejecutar la sentencia SQL especificada por el manejador `stmt1`, utilizando la función `SQLExecute` (línea 13). Aunque la consulta se ejecutará en la línea 13, los resultados de la consulta todavía no se han asignado a ninguna variable del programa C.

9. Para determinar dónde se devuelve el resultado de la consulta, se suele recurrir a la técnica de **columnas enlazadas**. Aquí, cada columna del resultado de una consulta se enlaza a una variable del programa C utilizando la función `SQLBindCol`. Las columnas se distinguen por su orden de aparición en la consulta SQL. En las líneas 15 y 16 de la Figura 9.10, las dos columnas de la consulta (`Apellido1` y `Sueldo`) se enlazan con las variables `apellido1` y `sueldo` del programa C, respectivamente.[15]

10. Por último, para recuperar los valores de columna en las variables del programa C, se utiliza la función `SQLFetch` (línea 17). Esta función se parece al comando FETCH de SQL incrustado. Si el resultado de una consulta tiene una colección de tuplas, cada llamada a `SQLFetch` obtiene la siguiente tupla y devuelve sus valores de columna en las variables del programa enlazadas. `SQLFetch` devuelve un código de excepción (*nonzero*) si no hay más tuplas.[16]

Como podemos ver, el uso dinámico de llamadas a funciones requiere mucha preparación para configurar las sentencias SQL y para enlazar los parámetros y los resultados de la consulta a las variables de programa adecuadas.

En CLI1, la consulta SQL sólo selecciona una tupla. La Figura 9.11 muestra un ejemplo de recuperación de varias tuplas. Asumimos que se han declarado las variables del programa C adecuadas, como en la Figura 9.1. El fragmento de programa CLI2 lee el número de un departamento y, después, recupera los empleados que trabajan en él. A continuación, un bucle itera por los registros de empleado, de uno en uno, e imprime el primer apellido y el sueldo del empleado.

---

[15] Una técnica alternativa denominada **columnas enlazadas** utiliza diferentes funciones SQL/CLI, particularmente `SQLGetCol` o `SQLGetData`, para recuperar columnas del resultado de una consulta sin enlazarlas previamente; se aplican después del comando `SQLFetch` de la línea 17.

[16] Si se utilizan variables de programa enlazadas, `SQLFetch` devuelve la tupla en un área de programa temporal. Cada `SQLGetCol` (o `SQLGetData`) posterior devuelve el valor de un atributo según el orden. Básicamente, por cada fila del resultado de una consulta, el programa debe iterar por los valores de atributo (columnas) de esa fila. Esto es útil si el número de columnas del resultado de la consulta es variable.

**Figura 9.11.** Fragmento de programa CLI2: fragmento de un programa C que utiliza SQL/CLI para una consulta con una colección de tuplas en su resultado.

```
    //Fragmento de programa CLI2:
0)  #include sqlcli.h ;
1)  void printDepartmentEmps() {
2)  SQLHSTMT stmt1 ;
3)  SQLHDBC con1 ;
4)  SQLHENV env1 ;
5)  SQLRETURN ret1, ret2, ret3, ret4 ;
6)  ret1 = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env1) ;
7)  if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1, &con1) else exit ;
8)  if (!ret2) ret3 = SQLConnect(con1, "dbs", SQL_NTS, "js", SQL_NTS, "xyz", SQL_NTS)
            else exit ;
9)  if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1, &stmt1) else exit ;
10) SQLPrepare(stmt1, "select Apellido1, Sueldo from EMPLEADO where Dno = ?", SQL_NTS) ;
11) prompt("Introduzca el número de departamento: ", dno) ;
12) SQLBindParameter(stmt1, 1, SQL_INTEGER, &dno, 4, &fetchlen1) ;
13) ret1 = SQLExecute(stmt1) ;
14) if (!ret1) {
15)   SQLBindCol(stmt1, 1, SQL_CHAR, &apellido1, 15, &fetchlen1) ;
16)   SQLBindCol(stmt1, 2, SQL_FLOAT, &sueldo, 4, &fetchlen2) ;
17)   ret2 = SQLFetch(stmt1) ;
18)   while (!ret2) {
19)        printf(apellido1, sueldo) ;
20)        ret2 = SQLFetch(stmt1) ;
21)        }
22)   }
23) }
```

## 9.3.2 JDBC: llamadas a funciones SQL para la programación en Java

Vamos a centrar nuestra atención en cómo puede llamarse a SQL desde el lenguaje de programación orientado a objetos Java.[17] Las bibliotecas de funciones para este acceso se conocen como **JDBC**.[18] El lenguaje de programación Java se diseñó para que fuera independiente de la plataforma; es decir, un programa debería poder ejecutarse en cualquier tipo de computador que tenga instalado un intérprete Java. Debido a esta portabilidad, muchos desarrolladores de RDBMS proporcionan controladores JDBC para que sea posible acceder a sus sistemas a través de programas Java. Un **controlador JDBC** es básicamente una implementación de las llamadas a las funciones especificadas en la interfaz de programación de aplicaciones (API) JDBC para el RDBMS de un desarrollador en particular. Por tanto, un programa Java con llamadas a funciones JDBC puede acceder a cualquier RDBMS que tenga un controlador JDBC.

Como Java está orientado a objetos, sus bibliotecas de funciones están implementadas como clases. Antes de poder procesar con Java las llamadas a funciones JDBC, es necesario importar las bibliotecas de clases JDBC, que se denominan `java.sql.*`. Se pueden descargar e instalar a través de la Web.[19]

[17] Esta sección asume que está familiarizado con los conceptos de la orientación a objetos y con los conceptos básicos de Java. Si no es así, posponga esta sección hasta haber leído el Capítulo 20.

[18] Como mencionamos anteriormente, JDBC es una marca registrada de Sun Microsystems, aunque normalmente se piensa que es un acrónimo de Conectividad de bases de datos de java (*Java Database Connectivity*).

[19] Están disponibles en varios sitios web; por ejemplo, http://industry.java.sun.com/products/jdbc/drivers.

JDBC está diseñado para que un solo programa Java se conecte a varias bases de datos diferentes, que a veces son denominadas orígenes de datos por parte del programa Java. Estos orígenes de datos podrían guardarse utilizando RDBMSs de diferentes desarrolladores y pueden residir en máquinas diferentes. Por tanto, diferentes accesos a orígenes de datos dentro del mismo programa Java pueden requerir controladores JDBC distintos de los diferentes desarrolladores. Para lograr esta flexibilidad se emplea una clase JDBC especial denominada **administrador de controladores**, que hace un seguimiento de los controladores instalados. Antes de utilizarlo, el controlador debe *registrarse* con el administrador de controladores. Las operaciones (métodos) de la clase del administrador de controladores son `getDriver`, `registerDriver` y `deregisterDriver`, que se pueden utilizar para añadir o eliminar dinámicamente los controladores. Como veremos, hay otras funciones para configurar y cerrar conexiones con los orígenes de datos.

Para cargar explícitamente un controlador JDBC, se puede utilizar la función genérica de Java que carga una clase. Por ejemplo, para cargar el controlador JDBC para el RDBMS de Oracle, puede utilizarse el siguiente comando:

```
Class.forName("oracle.jdbc.driver.OracleDriver")
```

Este comando registra el controlador con el administrador de controladores, de modo que queda disponible para el programa. También es posible cargar y registrar el o los controladores necesarios en la línea de comandos que ejecuta el programa, por ejemplo, incluyendo lo siguiente en la línea de comandos:

```
-Djdbc.drivers = oracle.jdbc.driver
```

A continuación mostramos los pasos típicos que se dan al escribir un programa de aplicación en Java para acceder a bases de datos a través de llamadas a funciones JDBC. Ilustramos los pasos haciendo referencia al ejemplo JDBC1 de la Figura 9.12, que lee el número del Documento Nacional de Identidad de un empleado e imprime el primer apellido y el sueldo de ese empleado.

1. En el programa Java debe importar la *biblioteca de clases* de JDBC. Estas clases se denominan `java.sql.*`, y las puede importar mediante la línea 1 de la Figura 9.12. También es preciso importar todas las bibliotecas de clases Java adicionales que el programa necesite.
2. Cargue el controlador JDBC como explicamos anteriormente (líneas 4 a 7). La excepción Java de la línea 5 ocurre si el controlador no se carga satisfactoriamente.
3. Cree las variables apropiadas que el programa Java necesite (líneas 8 y 9).
4. Se crea un **objeto de conexión** utilizando la función `getConnection` de la clase `DriverManager` de JDBC. En las líneas 12 y 13, el objeto de conexión se crea utilizando la llamada a la función `getConnection(cadenaurl)`, donde `cadenaurl` tiene este formato:

   ```
   jdbc:oracle:<tipocontrolador>:<cuentabd>/<contraseña>
   ```

   Un formato alternativo es el siguiente:

   ```
   getConnection(url, cuentabd, contraseña)
   ```

   Es posible establecer varias propiedades para un objeto de conexión, pero están relacionadas principalmente con las propiedades transaccionales, que explicaremos en el Capítulo 17.
5. En el programa se crea un **objeto de sentencia**. En JDBC, hay una clase de sentencia básica, `Statement`, con dos subclases especializadas: `PreparedStatement` y `CallableStatement`. Este ejemplo ilustra la creación y el uso de objetos `PreparedStatement`. El siguiente ejemplo (Figura 9.13) ilustra el otro tipo de objetos `Statement`. En la línea 14 se crea una cadena de consulta con un solo parámetro (indicado por el símbolo ?) en la variable `stmt1`. En la línea 15 se crea un objeto `p` de tipo `PreparedStatement` basado en la cadena de consulta `stmt1` y utilizando el objeto de conexión `conn`. En general, el programador debe utilizar objetos `PreparedStatement` si una consulta se va a ejecutar varias veces.

**Figura 9.12.** Fragmento de programa JDBC1: fragmento de un programa Java con JDBC.

```
    //Programa JDBC1:
0)  import java.io.* ;
1)  import java.sql.*
    ...
2)  class getEmpInfo {
3)     public static void main (String args []) throws SQLException, IOException {
4)          try { Class.forName("oracle.jdbc.driver.OracleDriver")
5)          } catch (ClassNotFoundException x) {
6)            System.out.println ("No se ha cargado el controlador") ;
7)          }
8)          String cuentabd, clave, dni, apellido1 ;
9)          Double sueldo ;
10)         cuentabd = readentry("Introduzca una cuenta de base de datos:") ;
11)         clave = readentry("Introduzca la contraseña:") ;
12)         Connection conn = DriverManager.getConnection
13)              ("jdbc:oracle:oci8:" + cuentabd + "/" + clave) ;
14)         String stmt1 = "select Apellido1, Sueldo from EMPLEADO where Dni = ?" ;
15)         PreparedStatement p = conn.prepareStatement(stmt1) ;
16)         dni = readentry("Introduzca un número de DNI: ") ;
17)         p.clearParameters() ;
18)         p.setString(1, dni) ;
19)         ResultSet r = p.executeQuery() ;
20)         while (r.next()) {
21)             apellido1 = r.getString(1) ;
22)             sueldo = r.getDouble(2) ;
23)             system.out.println(apellido1 + sueldo) ;
24)     } }
25) }
```

6. El símbolo de interrogación (?) de la línea 14 representa un **parámetro de sentencia**, que es un valor que se determinará en tiempo de ejecución, normalmente enlazándolo con una variable del programa Java. En general, podría haber varios parámetros, distinguidos por el orden de aparición de los símbolos de interrogación (el primer ? representa el parámetro 1, el segundo ? representa el parámetro 2, etcétera) en la sentencia, como explicamos anteriormente.
7. Antes de ejecutar una consulta `PreparedStatement`, debe enlazar los parámetros con variables del programa. En función del tipo del parámetro, debe aplicar funciones como `setString`, `setInteger`, `setDouble`, etcétera, al objeto `PreparedStatement` para establecer sus parámetros. En la Figura 9.12, el parámetro (indicado por ?) del objeto `p` se enlaza con la variable del programa Java `dni` en la línea 18. Se utiliza la función `setString` porque `dni` es una variable de cadena. Si hay *n* parámetros en la sentencia SQL, debemos tener *n* funciones `Set...`, cada una con una posición de parámetro diferente (1, 2, . . ., *n*). Generalmente, es aconsejable borrar todos los parámetros antes de establecer valores nuevos (línea 17).
8. Siguiendo estos preparativos, ahora ya puede ejecutar la sentencia SQL referenciada por el objeto `p` utilizando la función `executeQuery` (línea 19). JDBC tiene una función `execute` genérica, más dos funciones especilizadas: `executeUpdate` y `executeQuery`. `executeUpdate` se utiliza para las sentencias SQL de inserción, eliminación y actualización, y devuelve un valor entero que indica el número de tuplas que se han visto afectadas. `executeQuery` se utiliza para las sentencias de recuperación SQL, y devuelve un objeto de tipo `ResultSet`, que explicamos a continuación.
9. En la línea 19, el resultado de la consulta se devuelve en un objeto `r` de tipo `ResultSet`. Se parece a un array o tabla bidimensional, donde las tuplas son las filas y los atributos devueltos son las

**Figura 9.13.** Fragmento de programa JDBC2: fragmento de un programa Java que utiliza JDBC para una consulta con una colección de tuplas en el resultado.

```
    //Fragmento de programa JDBC2:
0)  import java.io.* ;
1)  import java.sql.*
    ...
2)  class printDepartmentEmps {
3)     public static void main (String args [ ]) throws SQLException, IOException {
4)          try { Class.forName("oracle.jdbc.driver.OracleDriver")
5)          } catch (ClassNotFoundException x) {
6)            System.out.println ("No se ha cargado el controlador") ;
7)          }
8)          String cuentabd, clave, apellido1 ;
9)          Double sueldo ;
10)         Integer dno ;
11)         cuentabd = readentry("Introduzca una cuenta de base de datos:") ;
12)         clave = readentry("Introduzca la contraseña:") ;
13)         Connection conn = DriverManager.getConnection
14)              ("jdbc:oracle:oci8:" + cuentabd + "/" + clave) ;
15)         dno = readentry("Introduzca un número de departamento: ") ;
16)         String q = "select Apellido1, Sueldo from EMPLEADO where Dno = " +
                 dno.tostring();
17)         Statement s = conn.createStatement() ;
18)         ResultSet r = s.executeQuery(q) ;
19)         while (r.next()) {
20)              apellido1 = r.getString(1) ;
21)              sueldo = r.getDouble(2) ;
22)              system.out.println(apellido1 + sueldo) ;
23)       } }
24) }
```

columnas. Un objeto `ResultSet` es parecido a un cursor en SQL incrustado y a un iterador en SQLJ. En nuestro ejemplo, cuando se ejecuta la consulta, `r` se refiere a una tupla anterior a la primera tupla del resultado de la consulta. La función `r.next()` (línea 20) mueve a la siguiente tupla (fila) del objeto `ResultSet` y devuelve NULL si ya no hay más objetos. Esto se utiliza para controlar el bucle. El programador se puede referir a los atributos de la tupla actual utilizando diversas funciones `get...` que dependen de tipo de cada atributo (por ejemplo, `getString`, `getInteger`, `getDouble`, etcétera). El programador puede utilizar las posiciones de los atributos (1, 2) o los nombres de los mismos (`"Apellido1"`, `"Sueldo"`) con las funciones `get...`. En nuestros ejemplos, utilizamos la notación posicional en las líneas 21 y 22.

En general, el programador puede comprobar las excepciones SQL después de cada llamada a una función JDBC.

JDBC no distingue entre consultas que devuelven una sola tupla y consultas que devuelven varias tuplas, a diferencia de otras técnicas. Esto es justificable porque un resultado de una sola tupla es un caso especial.

En el ejemplo JDBC1, la consulta SQL sólo selecciona *una tupla*, por lo que el bucle de las líneas 20 a 24 se ejecuta a lo sumo una vez. El siguiente ejemplo, mostrado en la Figura 9.13, ilustra la recuperación de varias tuplas. El fragmento de programa JDBC2 lee el número de un departamento y después recupera los empleados que trabajan en él. Un bucle itera después por esos empleados, de uno en uno, e imprime el primer apellido y el sueldo de los mismos. Este ejemplo también ilustra cómo podemos ejecutar directamente una consulta, sin tener que prepararla como ocurría en el ejercicio anterior. Esta técnica es preferible para las con-

sultas que se ejecutarán una sola vez, pues es más sencilla de programar. En la línea 17 de la Figura 9.13, el programador crea un objeto `Statement` (en lugar de `PreparedStatement`, como en el ejemplo anterior) sin asociarlo con una cadena de consulta en particular. La cadena de consulta `q` se pasa al objeto de sentencia `s` cuando se ejecuta en la línea 18.

Con esto concluimos nuestra breve introducción a JDBC. El lector interesado puede visitar el sitio web *http://java.sun.com/docs/books/tutorial/jdbc/*, que contiene muchos más detalles sobre JDBC.

# 9.4 Procedimientos almacenados de bases de datos y SQL/PSM

Vamos a concluir este capítulo con dos temas adicionales relacionados con la programación de bases de datos. En la Sección 9.4.1 explicamos el concepto de procedimientos almacenados, que son módulos de programa que el DBMS almacena en el servidor de bases de datos. Después, en la Sección 9.4.2, explicamos las extensiones de SQL que se especifican en el estándar para incluir en SQL estructuras de programación de propósito general. Estas extensiones se conocen como SQL/PSM (SQL/Módulos almacenados persistentes, *SQL/Persistent Stored Modules*) y se pueden utilizar para escribir procedimientos almacenados. SQL/PSM también sirve como ejemplo de lenguaje de programación de bases de datos que amplía un modelo y lenguaje de bases de datos (SQL) con algunas estructuras de programación, como sentencias condicionales y bucles.

## 9.4.1 Procedimientos almacenados de bases de datos y funciones

Hasta el momento, en nuestra presentación de las técnicas de programación de bases de datos asumíamos implícitamente que el programa de aplicación de bases de datos se ejecutaba en un computador cliente, que es una máquina diferente a la que alberga el servidor de bases de datos (y la parte principal del paquete DBMS). Aunque esto es adecuado para muchas aplicaciones, en ocasiones es útil crear módulos de programa de bases de datos (procedimientos o funciones) que el DBMS almacena y ejecuta en el servidor de bases de datos. Es lo que históricamente se conoce como **procedimientos almacenados** de bases de datos, aunque pueden ser funciones o procedimientos. El término utilizado en el estándar SQL para los procedimientos almacenados es **módulos almacenados persistentes**, porque el DBMS almacena persistentemente estos programas, algo parecido a los datos persistentes almacenados por el DBMS.

Los procedimientos almacenados son útiles en las siguientes circunstancias:

- Si varias aplicaciones necesitan un mismo programa de bases de base de datos, este último se puede almacenar en el servidor e invocarlo desde esas aplicaciones. Esto reduce la duplicidad del esfuerzo y mejora la modularidad del software.
- La ejecución de un programa en el servidor puede reducir el coste derivado de la transferencia y la comunicación de datos entre el cliente y el servidor en ciertas situaciones.
- Estos procedimientos pueden mejorar la potencia de modelado de las vistas al permitir que los usuarios de bases de datos cuenten con tipos más complejos de datos derivados. Además, se pueden utilizar esos tipos para comprobar restricciones más complejas que quedan fuera de la especificación de aserciones y *triggers*.

En general, muchos DBMSs comerciales permiten escribir procedimientos y funciones almacenados en un lenguaje de programación de propósito general. De forma alternativa, un procedimiento almacenado puede estar compuesto por comandos SQL sencillos, como recuperaciones y actualizaciones. La forma general para declarar un procedimiento almacenado es la siguiente:

```
CREATE PROCEDURE <nombre del procedimiento> (<parámetros>)
<declaraciones locales>
<cuerpo del procedimiento> ;
```

Los parámetros y las declaraciones locales son opcionales, y sólo se especifican si se necesitan. Para declarar una función, se necesita un tipo de devolución:

```
CREATE FUNCTION <nombre de la función> (<parámetros>)
RETURNS <tipo de devolución>
<declaraciones locales>
<cuerpo de la función> ;
```

Si el procedimiento (o función) se escribe en un lenguaje de programación de propósito general, es típico especificar el lenguaje, así como el nombre del fichero donde se almacenará el código del programa. Por ejemplo, se puede utilizar el siguiente formato:

```
CREATE PROCEDURE <nombre del procedimiento> (<parámetros>)
LANGUAGE <nombre del lenguaje de programación>
EXTERNAL NAME <nombre de ruta del fichero> ;
```

En general, cada parámetro debe tener un **tipo de parámetro**; uno de los tipos de datos de SQL. También debe tener un **modo de parámetro**, que es IN, OUT, o INOUT. Estos modos se corresponden con los parámetros cuyos valores son de sólo entrada, sólo salida, o de entrada y salida.

Como el DBMS almacena persistentemente los procedimientos y las funciones, debe ser posible llamarlos desde varias interfaces SQL y técnicas de programación. Se puede utilizar la **sentencia CALL** del estándar SQL para invocar un procedimiento almacenado (desde una interfaz interactiva o desde SQL incrustado o SQLJ). El formato de la sentencia es el siguiente:

```
CALL <nombre del procedimiento o función> (<lista de argumentos>) ;
```

Si esta sentencia es llamada desde JDBC, debe asignarse a un objeto de sentencia de tipo `CallableStatement` (consulte la Sección 9.3.2).

### 9.4.2 SQL/PSM: ampliación de SQL para especificar módulos almacenados persistentes

SQL/PSM es la parte del estándar SQL encargada de especificar cómo han de escribirse los módulos almacenados persistentes. Incluye las sentencias para crear las funciones y los procedimientos que describimos en la sección anterior. También incluye las estructuras de programación adicionales que permiten mejorar la potencia de SQL con el propósito de escribir el código (o cuerpo) de los procedimientos o funciones almacenados.

En esta sección explicamos las estructuras de SQL/PSM de las sentencias condicionales (bifurcaciones) y de las sentencias de bucle. Esto nos ofrecerá una introducción del tipo de estructuras que SQL/PSM ha incorporado;[20] después ofrecemos un ejemplo para ilustrar cómo se utilizan estas estructuras.

La sentencia de bifurcación condicional en SQL/PSM tiene la siguiente forma:

```
IF <condición> THEN <lista de sentencias>
  ELSEIF <condición> THEN <lista de sentencias>
  . . .
  ELSEIF <condición> THEN <lista de sentencias>
  ELSE <lista de sentencias>
  END IF ;
```

[20] Sólo ofrecemos una breve introducción a SQL/PSM. Este estándar tiene otras muchas características.

**Figura 9.14.** Declaración de una función en SQL/PSM.

```
    //Función PSM1:
0)  CREATE FUNCTION TamDpto(IN nodpto INTEGER)
1)  RETURNS VARCHAR [7]
2)  DECLARE NumEmps INTEGER ;
3)  SELECT COUNT(*) INTO NumEmps
4)  FROM EMPLEADO WHERE Dno = nodpto ;
5)  IF NumEmps > 100 THEN RETURN "ENORME"
6)     ELSEIF NumEmps > 25 THEN RETURN "GRANDE"
7)     ELSEIF NumEmps > 10 THEN RETURN "MEDIO"
8)     ELSE RETURN "PEQUEÑO"
9)  END IF ;
```

Considere el ejemplo de la Figura 9.14, que ilustra el uso de la estructura de bifurcación condicional en una función de SQL/PSM. La función devuelve un valor de cadena (línea 1) que describe el tamaño de un departamento en base al número de empleados. Hay un parámetro IN entero, `nodpto`, que facilita el número de departamento. En la línea 2 se declara la variable local `NumEmps`. La consulta de las líneas 3 y 4 devuelve el número de empleados del departamento, y la rama condicional de las líneas 5 a 8 devuelve después uno de estos valores, basándose en el número de empleados: {'ENORME', 'GRANDE', 'MEDIO', 'PEQUEÑO'}.

SQL/PSM tiene varias estructuras para crear bucles. Cuenta con las estructuras estándar `while` y `repeat`, que tienen los siguientes formatos:

```
WHILE <condición> DO
  <lista de sentencias>
END WHILE ;
REPEAT
  <lista de sentencias>
UNTIL <condición>
END REPEAT ;
```

También existe una estructura de bucle basada en un cursor. La lista de sentencias de dicho bucle se ejecuta una vez por cada tupla del resultado de la consulta. Tiene el siguiente formato:

```
FOR <nombre del bucle> AS <nombre de cursor> CURSOR FOR <consulta> DO
   <lista de sentencias>
END FOR ;
```

Los bucles pueden tener nombres, y hay una sentencia LEAVE <nombre del bucle> para salir del bucle cuando se satisface una condición. SQL/PSM tiene muchas otras características, pero quedan fuera de los objetivos de esta presentación.

# 9.5 Resumen

En este capítulo hemos presentado las características adicionales del lenguaje de bases de datos SQL. En particular, en la Sección 9.1 ofrecimos una panorámica de las técnicas más importantes de programación de bases de datos. Después, en las Secciones 9.2 a 9.4 explicamos las distintas metodologías de programación de aplicaciones de base de datos.

En la Sección 9.2 explicamos la técnica general conocida como SQL incrustado, donde las consultas forman parte del código fuente del programa. Normalmente, se utiliza un precompilador para extraer los comandos SQL del programa para que el DBMS los procese y los sustituya por llamadas a funciones del código DBMS

compilado. Ofrecimos una introducción de SQL incrustado haciendo uso del lenguaje de programación C como lenguaje *host*. También explicamos la técnica SQLJ para incrustar SQL en programas Java. Explicamos e ilustramos con ejemplos los conceptos de cursor (para SQL incrustado) e iterador (para SQLJ) a fin de mostrar su uso para recorrer con un bucle las tuplas del resultado de una consulta y extraer los valores de los atributos en variables del programa para después procesarlos.

En la Sección 9.3 explicamos el uso de las bibliotecas de llamadas a funciones para acceder a las bases de datos SQL. Esta técnica es más dinámica que e SQL incrustado, pero requiere una programación más compleja porque los tipos y la cantidad de atributos del resultado de una consulta pueden determinarse en tiempo de ejecución. También hemos ofrecido una visión general del estándar SQL/CLI, con ejemplos utilizando C como lenguaje *host*. Asimismo, hemos explicado algunas de las funciones de la biblioteca SQL/CLI, cómo se pasan las consultas como cadenas, cómo se asignan los parámetros de una consulta en tiempo de ejecución, y cómo se devuelven los resultados en variables del programa. A continuación, ofrecimos una visión general de la biblioteca de clases JDBC, que se utiliza con Java, y explicamos algunas de sus clases y operaciones. En particular, la clase `ResultSet` se utiliza para crear objetos que guardan el resultado de una consulta, y que después se pueden recorrer con la operación `next()`. También hemos explicado las funciones `get` y `set` para recuperar los valores de los atributos y establecer los valores de los parámetros.

Por último, en la Sección 9.4, ofrecimos una introducción a los procedimientos almacenados, y explicamos SQL/PSM como un ejemplo de lenguaje de programación de bases de datos. Es importante saber que hemos preferido ofrecer una introducción comparativa de las tres metodologías de programación de bases de datos, puesto que el estudio en profundidad de una metodología en particular es un tema digno de tener su propio libro de texto.

## Preguntas de repaso

**9.1.** ¿Qué es ODBC? ¿Cómo está relacionado con SQL/CLI?

**9.2.** ¿Qué es JDBC? ¿Es un ejemplo de SQL incrustado o de uso de llamadas a funciones?

**9.3.** Enumere las tres metodologías principales para programar bases de datos. ¿Cuáles son las ventajas y los inconvenientes de cada una de ellas?

**9.4.** ¿Qué es el problema del desajuste de impedancia? ¿Cuál de las tres metodologías de programación minimiza este problema?

**9.5.** Describa el concepto de cursor y cómo se utiliza en SQL incrustado.

**9.6.** ¿Para qué se utiliza SQLJ? Describa los dos tipos de iteradores disponibles en SQLJ.

## Ejercicios

**9.7.** Partiendo de la base de datos de la Figura 1.2, cuyo esquema aparece en la Figura 2.1, escriba un fragmento de programa que lea el nombre de un estudiante e imprima su calificación media, asumiendo que A=4, B=3, C=2 y D=1 puntos. Utilice SQL incrustado con C como lenguaje *host*.

**9.8.** Repita el Ejercicio 9.7, pero utilice SQLJ con Java como lenguaje *host*.

**9.9.** Considere el esquema de la base de datos relacional de la Figura 6.14. Escriba un fragmento de programa que recupere la lista de libros que debieron entregarse ayer e imprima el título del libro y el nombre del prestatario de cada uno. Utilice SQL incrustado con C como lenguaje *host*.

**9.10.** Repita el Ejercicio 9.9, pero utilice SQLJ con Java como lenguaje *host*.

**9.11.** Repita los Ejercicios 9.7 y 9.9, pero utilice SQL/CLI con C como lenguaje *host*.

**9.12.** Repita los Ejercicios 9.7 y 9.9, pero utilice JDBC con Java como lenguaje *host*.

**9.13.** Repita el Ejercicio 9.7, pero escriba una función en SQL/PSM.

**9.14.** Cree una función en PSM que calcule el sueldo medio de la tabla EMPLEADO de la Figura 5.5.

## Ejercicios de práctica

**9.15.** Considere la base de datos UNIVERSIDAD del Ejercicio 3.16 y del Ejercicio de práctica 3.31 que se creó y rellenó en el Ejercicio de práctica 8.26 utilizando Oracle. Escriba y pruebe un programa Java que lleve a cabo las funciones ilustradas en la siguiente sesión de terminal:

```
$ java p1
Número de estudiante: 1234
Semestre: Otoño
Año: 2005

Menú principal
(1) Añadir una clase
(2) Eliminar una clase
(3) Ver mi planificación
(4) Salir
Introduzca su opción: 1
Número de curso: CC 1010
Sección: 2
Clase añadida

Menú principal
(1) Añadir una clase
(2) Eliminar una clase
(3) Ver mi planificación
(4) Salir
Introduzca su opción: 1
Número de curso: MAT 2010
Sección: 1
Clase añadida

Menú principal
(1) Añadir una clase
(2) Eliminar una clase
(3) Ver mi planificación
(4) Salir
Introduzca su opción: 3
Su planificación actual es:
CC 1010 Sección 2, Introducción a los computadores, Profesor:
Eduardo
MAT 2010 Sección 1, Cálculo I, Profesor: Antonio

Menú principal
(1) Añadir una clase
(2) Eliminar una clase
(3) Ver mi planificación
(4) Salir
Introduzca su opción: 2
Número de curso: CC 1010
Sección: 2
Clase eliminada

Menú principal
(1) Añadir una clase
(2) Eliminar una clase
```

```
(3) Ver mi planificación
(4) Salir
Introduzca su opción: 3
Su planificación actual es:
MAT 2010 Sección 1, Cálculo I, Profesor: Antonio

Menú principal
(1) Añadir una clase
(2) Eliminar una clase
(3) Ver mi planificación
(4) Salir
Introduzca su opción: 4
$
```

Como la sesión de terminal muestra, el estudiante se registra en el programa con un número de identificación y el término y el año de registro. La opción *Añadir una clase* permite al estudiante añadir una clase a su agenda actual, y la opción *Eliminar una clase* le permite eliminar una clase de su agenda. La opción *Ver mi planificación* lista las clases en las que el estudiante está registrado.

**9.16** Considere la base de datos PEDIDOS_CORREO del Ejercicio de práctica 3.32; la base de datos relacional se creó y rellenó con Oracle en el Ejercicio de práctica 8.27. Escriba y pruebe un programa Java que imprima una factura para un número de pedido dado, como se ilustra en la siguiente sesión de terminal:

```
$java p2
   Número de pedido: 1020
   ----------------------------------------
   Cliente: Carlos Campos
   Número de cliente: 1111
   Código postal: 67226

   ____________________________________________

   Número de pedido: 1020
   Recepcionado por: Juan Sanz (emp. núm. 1122)
   Fecha de recepción: 10-DIC-1994
   Fecha de venta: 13-DIC-1994
   Núm. Rep.    Nombre rep.    Cantidad    Precio    Suma

   ____________________________________________________

   10506        Tuerca         200         1,99      398,00
   10509        Cerrojo        100         1,55      155,00

   ____________________________________________________

   Total: 553.00
   $
```

## Bibliografía seleccionada

Hay muchos libros que describen distintos aspectos de la programación de bases de datos SQL. Por ejemplo, Sunderraman (2004) describe la programación en el DBMS de Oracle 9i y Price (2002) se centra en la programación JDBC en el sistema Oracle 9i. En la Web también puede encontrar muchos recursos.

# PARTE 3

# Teoría y metodología del diseño de bases de datos

CAPÍTULO 10

# Dependencias funcionales y normalización en bases de datos relacionales

En los Capítulos del 5 al 9, mostramos varios aspectos del modelo relacional y los lenguajes asociados a él. Cada *esquema de relación* consta de un número de atributos, mientras que un *esquema de base de datos relacional* está compuesto por un número de esquemas de relación. Hasta ahora sólo hemos utilizado el sentido común del diseñador de la base de datos para agrupar los atributos y formar así un esquema de relación, o bien hemos utilizado un diseño de esquema de base de datos a partir de un modelo de datos conceptual como ER o EER, o algún otro. Estos modelos hacen que el diseñador identifique los tipos de entidad y de relación y sus respectivos atributos, lo que nos lleva a un agrupamiento natural y lógico de los atributos en relaciones cuando van seguidos por los procedimientos de mapeado del Capítulo 7. Sin embargo, aún necesitamos algún tipo de medida formal que nos indique por qué un agrupamiento de atributos en el esquema de una relación puede ser mejor que otro. Hasta el momento, en nuestro debate sobre el diseño conceptual de los Capítulos 3 y 4 y su asignación en el modelo relacional del Capítulo 7, no hemos desarrollado ningún método que nos indique la idoneidad de la calidad del diseño, aparte de la intuición del diseñador. En este capítulo vamos a ver parte de la teoría desarrollada con el objetivo de evaluar esquemas relacionales encaminados a la calidad del diseño; es decir, mediremos formalmente por qué un conjunto de agrupaciones de atributos en un esquema de relación es mejor que otro.

Hay dos niveles a los que podemos explicar la *bondad* de los esquemas de relación. El primero es el **nivel lógico** (o **conceptual**): cómo los usuarios interpretan los esquemas de relación y el significado de sus atributos. Disponer de un buen esquema de relación a este nivel permite a los usuarios comprender con claridad el significado de los datos en la relaciones y, por consiguiente, formular sus consultas correctamente. El segundo nivel es el de **implementación** (o **almacenamiento**): de qué modo se almacenan y actualizan las tuplas en una relación base. Este nivel se aplica sólo a esquemas de relación base (cómo se almacenarán físicamente los ficheros), mientras que a nivel lógico nos interesan tanto las relaciones base como las vistas (relaciones virtuales). La teoría de diseño de una base de datos relacional desarrollada en este capítulo se aplica fundamentalmente a las *relaciones base,* aunque algunos criterios de idoneidad también se utilizan en las vistas (consulte la Sección 10.1).

Como ocurre con otros muchos problemas de diseño, el de una base de datos debe llevarse a cabo usando dos metodologías: ascendente (*bottom-up*) o descendente (*top-down*). Una **metodología de diseño de tipo ascendente**, llamada también *diseño por síntesis*, tiene como punto de partida las relaciones básicas entre *atributos*

*individuales*, y los usa para construir los esquemas de relación. Esta metodología no es muy popular en la práctica[1], ya que tiene el problema de tener que recopilar al principio una gran cantidad de relaciones binarias entre los atributos. Como contrapartida, en una **metodología descendente**, conocida también como *diseño por análisis*, se empieza con varios agrupamientos de atributos de una relación que están juntos de forma natural, como por ejemplo, en una factura, un formulario o un informe. Las relaciones son entonces analizadas individual y colectivamente, lo que conduce a una descomposición posterior que permite conocer todas las propiedades deseables. La teoría descrita en este capítulo es aplicable a ambas metodologías de diseño, aunque es más práctica cuando se emplea en la de tipo descendente.

Iniciamos este capítulo comentando de manera informal algunos criterios en la Sección 10.1 para determinar un buen o mal esquema de relación. En la Sección 10.2 definimos el concepto de *dependencia funcional,* una restricción formal entre los atributos que es la herramienta principal para la medida formal de la idoneidad del agrupamiento de atributos en los esquemas de relación. También se estudian y analizan las propiedades de las dependencias funcionales. La Sección 10.3 se centra en las formas normales y en el proceso de normalización usando dependencias funcionales. Las formas normales sucesivas están definidas para cumplir el conjunto de restricciones deseables expresadas mediante dependencias funcionales. El procedimiento de normalización consiste en la aplicación de una serie de comprobaciones de las relaciones para cumplir con unos requisitos cada vez más restrictivos y descomponer las relaciones cuando sea necesario. En la Sección 10.4 tratamos las definiciones más generales de las formas normales que pueden aplicarse directamente a un diseño concreto y que no precisan de un análisis paso a paso y una normalización.

El Capítulo 11 continúa con el desarrollo de la teoría para un buen diseño del esquema relacional. Comentamos las propiedades deseables de la descomposición relacional (propiedad de reunión no aditiva y de preservación de dependencia funcional) y después consideramos la metodología de tipo ascendente en el diseño de una base de datos que consiste en un conjunto de algoritmos. Estos algoritmos asumen como entrada un conjunto dado de dependencias funcionales y consiguen un diseño relacional en una forma normal de destino a la vez que añade las propiedades deseables antes comentadas. También se presenta un algoritmo general que verifica si una descomposición tiene o no la propiedad de reunión sin pérdida (Algoritmo 11.1). El Capítulo 11 contiene además la definición de tipos de dependencias adicionales y formas normales avanzadas que lleva más allá la *idoneidad* de un esquema de relación.

El lector que sólo está interesado en una introducción informal a la normalización puede saltarse las Secciones 10.2.3, 10.2.4 y 10.2.5. Si no se estudia el Capítulo 11 en un curso, recomendamos una introducción rápida a las propiedades deseables de la descomposición mostradas de la Sección 11.1 y un debate de la propiedad NJB, además del Capítulo 10.

# 10.1 Directrices de diseño informales para los esquemas de relación

Antes de entrar en detalles con la teoría formal del diseño de bases de datos relacionales, vamos a ver en esta sección cuatro *medidas informales* de calidad para el diseño de un esquema de relación:

- La semántica de los atributos.
- La reducción de información redundante en las tuplas.
- La reducción de los valores NULL en las tuplas.
- Prohibición de la posibilidad de generar tuplas falsas.

Como podremos ver, estas medidas no siempre son independientes entre sí.

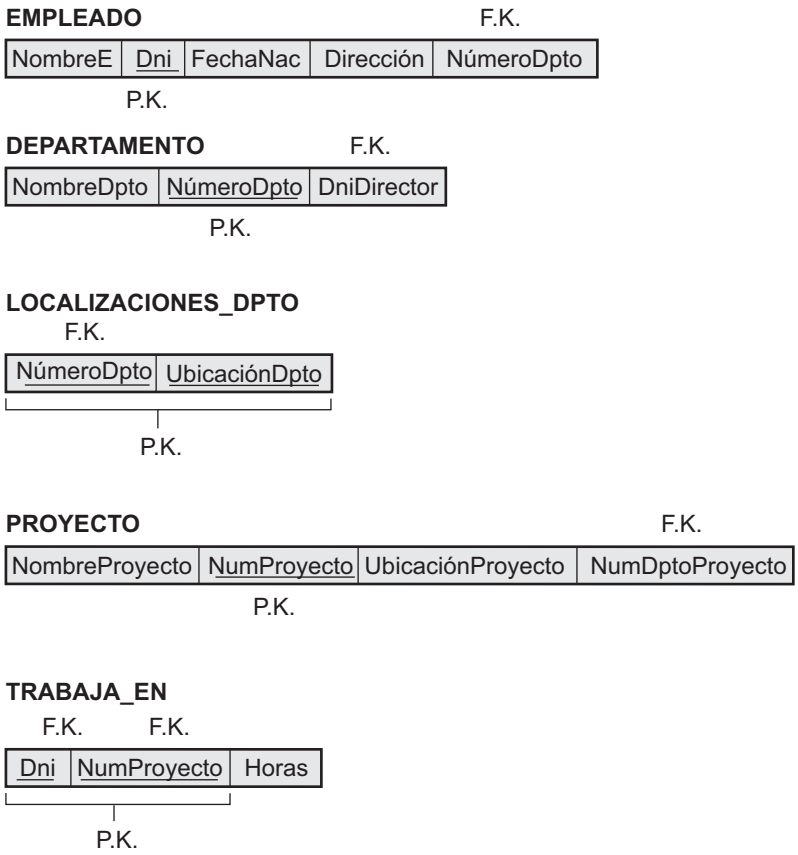
[1] El modelo relacional binario es una excepción en la que se usa esta metodología en la práctica. Un ejemplo del mismo es la metodología NIAM (Verheijen y VanBekkum 1982).

### 10.1.1 Impartir una semántica clara a los atributos de las relaciones

Siempre que agrupamos atributos para formar un esquema de relación asumimos que pertenecen a una relación que tiene cierta similitud con el mundo real y una interpretación propia asociada a ellos. La semántica de una relación hace referencia a la interpretación de los valores de un atributo en una tupla. En el Capítulo 5 vimos que una relación puede interpretarse como un conjunto de hechos. Si el diseño conceptual descrito en los Capítulos 3 y 4 se lleva a cabo cuidadosamente y el procedimiento de mapeado del Capítulo 7 se sigue sistemáticamente, el diseño del esquema relacional debería tener un significado claro.

En general, cuanto más sencillo es explicar la semántica de la relación, mejor será el diseño del esquema de relación. Para ilustrar esto, considere la Figura 10.1, una versión simplificada del esquema de base de datos relacional EMPRESA de la Figura 5.5, y la Figura 10.2, que muestra un ejemplo de estado de relación. El significado del esquema de relación EMPLEADO es muy simple: cada tupla representa a un empleado, con valores que contienen su nombre (NombreE), su Documento Nacional de Identidad (Dni), su fecha de nacimiento (FechaNac), su dirección (Dirección) y el número del departamento en el que trabaja (NúmeroDpto). El atributo NúmeroDpto es una *foreign key* que representa una *relación implícita* entre EMPLEADO y DEPARTAMENTO. Las semánticas de los esquemas DEPARTAMENTO y PROYECTO son también muy directas: cada tupla DEPARTAMENTO representa a una entidad departamento, y cada tupla PROYECTO es una entidad proyecto. El atributo DniDirector de DEPARTAMENTO relaciona un departamento con el empleado que es director del mismo, mientras que NumDptoProyecto de PROYECTO asocia un proyecto con el departamento que lo gestiona; ambos atributos son *foreign keys*. La facilidad con la que se pueda explicar el significado de los atributos de una relación es una *medida informal* de lo bien que está diseñada esa relación.

**Figura 10.1.** Una versión simplificada del esquema de base de datos relacional EMPRESA.



La semántica de los otros dos esquemas de relación de la Figura 10.1 es algo más compleja. Cada tupla de LOCALIZACIONES_DPTO consta de un número de departamento (NúmeroDpto) y *una* de las localizaciones del departamento (UbicaciónDpto). Cada tupla de TRABAJA_EN contiene el DNI del empleado (DniEmpleado), el número de uno de los proyectos en los que trabaja (NumProyecto) y el número de horas semanales que le dedica al mismo (Horas). Sin embargo, ambos esquemas tienen una interpretación bien definida y sin ambigüedad. El esquema LOCALIZACIONES_DPTO representa un atributo multivalor de DEPARTAMENTO, mientras que TRABAJA_EN es una relación M:N entre EMPLEADO y PROYECTO. Por consiguiente, todo el esquema de relaciones de la Figura 10.1 podría considerarse como fácil de explicar y, por tanto, bueno desde el punto de vista de contar con una semántica clara. De esta forma, podemos formular la siguiente directriz informal de diseño.

### Directriz 1

Diseñar un esquema de relación para que sea fácil explicar su significado. No combine atributos de varios tipos de entidad y de relación en una única relación. Intuitivamente, si un esquema de relación se corres-

**Figura 10.2.** Ejemplo del estado de la base de datos para el esquema de base de datos relacional de la Figura 10.1.

**EMPLEADO**

| NombreE | Dni | FechaNac | Dirección | NúmeroDpto |
|---|---|---|---|---|
| Pérez Pérez, José | 123456789 | 09-01-1965 | Eloy I, 98 | 5 |
| Campos Sastre, Alberto | 333445555 | 08-12-1955 | Avda. Ríos, 9 | 5 |
| Jiménez Celaya, Alicia | 999887777 | 19-07-1968 | Gran Vía, 38 | 4 |
| Sainz Oreja, Juana | 987654321 | 20-06-1941 | Cerquillas, 67 | 4 |
| Ojeda Ordóñez, Fernando. | 666884444 | 15-09-1962 | Portillo, s/n | 5 |
| Oliva Avezuela, Aurora | 453453453 | 31-07-1972 | Antón, 6 | 5 |
| Pajares Morera, Luis | 987987987 | 29-03-1969 | Enebros, 90 | 4 |
| Ochoa Paredes, Eduardo | 888665555 | 10-11-1937 | Las Peñas, 1 | 1 |

**DEPARTAMENTO**

| NombreDpto | NúmeroDpto | DniDirector |
|---|---|---|
| Investigación | 5 | 333445555 |
| Administración | 4 | 987654321 |
| Sede central | 1 | 888665555 |

**LOCALIZACIONES_DPTO**

| NúmeroDpto | UbicaciónDpto |
|---|---|
| 1 | Madrid |
| 4 | Gijón |
| 5 | Valencia |
| 5 | Sevilla |
| 5 | Madrid |

**PROYECTO**

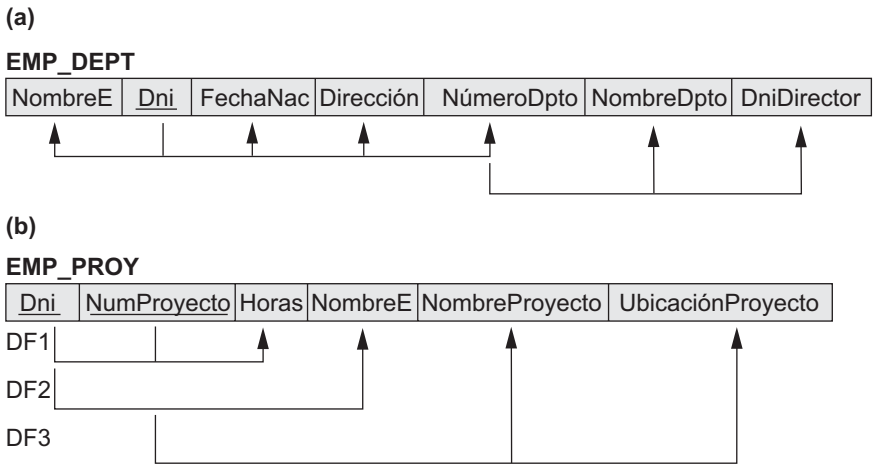
| NombreProyecto | NumProyecto | UbicaciónProyecto | NumDptoProyecto |
|---|---|---|---|
| ProductoX | 1 | Valencia | 5 |
| ProductoY | 2 | Sevilla | 5 |
| ProductoZ | 3 | Madrid | 5 |
| Computación | 10 | Gijón | 4 |
| Reorganización | 20 | Madrid | 1 |
| Comunicaciones | 30 | Gijón | 4 |

**Figura 10.2.** (*Continuación*).

**TRABAJA_EN**

| Dni | NumProyecto | Horas |
|---|---|---|
| 123456789 | 1 | 32.5 |
| 123456789 | 2 | 7.5 |
| 666884444 | 3 | 40.0 |
| 453453453 | 1 | 20.0 |
| 453453453 | 2 | 20.0 |
| 333445555 | 2 | 10.0 |
| 333445555 | 3 | 10.0 |
| 333445555 | 10 | 10.0 |
| 333445555 | 20 | 10.0 |
| 999887777 | 30 | 30.0 |
| 999887777 | 10 | 10.0 |
| 987987987 | 10 | 35.0 |
| 987987987 | 30 | 5.0 |
| 987654321 | 30 | 20.0 |
| 987654321 | 20 | 15.0 |
| 888665555 | 20 | Null |

**Figura 10.3.** Dos esquemas de relación con anomalías en la actualización. (a) EMP_DEPT y (b) EMP_PROY




ponde con un tipo de entidad o de relación, es correcto interpretar y explicar su significado. Por contra, si la relación está compuesta por una mezcla de múltiples entidades y relaciones, se producirá una ambigüedad semántica y la relación no podrá explicarse con claridad.

Los esquemas de relación de las Figuras 10.3(a) y 10.3(b) tienen también semánticas claras (el lector debe ignorar por el momento las líneas que aparecen bajo las relaciones; se utilizan para documentar la notación de dependencia funcional que explicamos en la Sección 10.2). Una tupla en el esquema de relación EMP_DEPT de la Figura 10.3(a) representa a un solo empleado, aunque incluye información adicional: el

nombre del departamento en el que trabaja (NombreDpto) y el DNI del director de ese departamento (DniDirector). En la relación EMP_PROY de la Figura 10.3(b), cada tupla relaciona un empleado con un proyecto, aunque incluye también el nombre del empleado (NombreE), el del proyecto (NombreProyecto) y la localización de éste (UbicaciónProyecto). Aunque desde el punto de vista lógico no existe nada erróneo en estas dos relaciones, se considera que tienen un diseño pobre porque violan la directriz 1 al mezclar atributos de dos entidades del mundo real; EMP_DEPT combina atributos de empleados y departamentos, mientras que EMP_PROY combina atributos de empleados y proyectos y la relación TRABAJA_EN. Deberían utilizarse como vistas, aunque esto provocaría problemas cuando se usasen como relaciones base, tal y como veremos en la siguiente sección.

### 10.1.2 Información redundante en tuplas y anomalías en la actualización

Uno de los objetivos de un esquema de diseño es reducir el espacio de almacenamiento utilizado por las relaciones base (y, por tanto, por los ficheros correspondientes). El agrupamiento de atributos en esquemas de relación tiene un efecto significativo sobre el espacio de almacenamiento. Por ejemplo, compare el espacio empleado por las dos relaciones base EMPLEADO y DEPARTAMENTO de la Figura 10.2 con el necesario para EMP_DEPT de la Figura 10.4, que es el resultado de aplicar la operación NATURAL JOIN a EMPLEADO y DEPARTAMENTO. En EMP_DEPT, los valores de atributo pertenecientes a un departamento particular (NúmeroDpto, NombreDpto, DniDirector) están repetidos para *cada empleado que trabaja en ese departamento.* Por contra, la información de cada departamento sólo aparece una vez en la relación DEPARTAMENTO de la Figura 10.2. Por cada empleado que trabaja en ese departamento, sólo se repite el número de departamento (NúmeroDpto) en la relación EMPLEADO como una *foreign key*. A la relación EMP_PROY pueden aplicársele comentarios similares (véase la Figura 10.4), que aumenta la relación TRABAJA_EN con atributos adicionales procedentes de EMPLEADO y PROYECTO.

Otro serio problema que aparece cuando se usan las relaciones de la Figura 10.4 como relaciones base son las **anomalías en la actualización**, las cuales pueden clasificarse en anomalías de inserción, de borrado y de modificación.[2]

**Anomalías de inserción.** Las anomalías de inserción pueden diferenciarse en dos tipos, que se ilustran con los siguientes ejemplos basados en la relación EMP_DEPT:

- Para insertar una nueva tupla en EMP_DEPT, debemos incluir los valores correspondientes al departamento en el que dicho empleado trabaja, o valores NULL en el caso de que no lo haga para ninguno. Por ejemplo, para insertar una nueva tupla para un empleado que trabaja en el departamento número 5, debemos introducir correctamente los valores de atributo del departamento 5, de modo que sean *coherentes* con los valores correspondientes del resto de tuplas de EMP_DEPT. En el diseño de la Figura 10.2 no tenemos que preocuparnos por el problema de la coherencia ya que sólo indicamos el número de departamento en la tupla de empleado; el resto de valores de atributo del departamento 5 sólo se graban una vez en la base de datos, como una única tupla de la relación DEPARTAMENTO.
- Es complicado insertar un nuevo departamento que aún no tenga ningún empleado en la relación EMP_DEPT. La única forma de hacerlo es colocando valores NULL en los atributos correspondiente al empleado. Esto genera un problema, ya que el DNI es la clave principal de EMP_DEPT, y se supone que cada tupla representa a una entidad empleado, no a una entidad departamento. Además, cuando se asigna el primer empleado a ese departamento, ya no necesitaremos nunca más esta tupla con valores NULL. Este problema no se da en el diseño de la Figura 10.2 porque un departamento se introduce en

[2] Estas anomalías fueron identificadas por Codd (1972a) para justificar la necesidad de normalización en las relaciones, como ya comentaremos en la Sección 10.3.

**Figura 10.4.** Ejemplo de los estados de EMP_DEPT y EMP_PROY resultantes de aplicar una NATURAL JOIN a las relaciones de la Figura 10.2. Éstas deberían almacenarse como relaciones base por motivos de rendimiento.

**EMP_DEPT**

Redundancia

| NombreE | Dni | FechaNac | Dirección | NúmeroDpto | NombreDpto | DniDirector |
|---|---|---|---|---|---|---|
| Pérez Pérez, José | 123456789 | 09-01-1965 | Eloy I, 98 | 5 | Investigación | 333445555 |
| Campos Sastre, Alberto | 333445555 | 08-12-1955 | Avda. Ríos, 9 | 5 | Investigación | 333445555 |
| Jiménez Celaya, Alicia | 999887777 | 19-07-1968 | Gran Vía, 38 | 4 | Administración | 987654321 |
| Sainz Oreja, Juana | 987654321 | 20-06-1941 | Cerquillas, 67 | 4 | Administración | 987654321 |
| Ojeda Ordóñez, Fernando. | 666884444 | 15-09-1962 | Portillo, s/n | 5 | Investigación | 333445555 |
| Oliva Avezuela, Aurora | 453453453 | 31-07-1972 | Antón, 6 | 5 | Investigación | 333445555 |
| Pajares Morera, Luis | 987987987 | 29-03-1969 | Enebros, 90 | 4 | Administración | 987654321 |
| Ochoa Paredes, Eduardo | 888665555 | 10-11-1937 | Las Peñas, 1 | 1 | Sede central | 888665555 |

**EMP_PROY**

Redundancia Redundancia

| Dni | NumProyecto | Horas | NombreE | NombreProyecto_ | UbicaciónProyecto |
|---|---|---|---|---|---|
| 123456789 | 1 | 32.5 | Pérez Pérez, José | ProductoX | Valencia |
| 123456789 | 2 | 7.5 | Pérez Pérez, José | ProductoY | Sevilla |
| 666884444 | 3 | 40.0 | Ojeda Ordóñez, Fernando. | ProductoZ | Madrid |
| 453453453 | 1 | 20.0 | Oliva Avezuela, Aurora | ProductoX | Valencia |
| 453453453 | 2 | 20.0 | Oliva Avezuela, Aurora | ProductoY | Sevilla |
| 333445555 | 2 | 10.0 | Campos Sastre, Alberto | ProductoY | Sevilla |
| 333445555 | 3 | 10.0 | Campos Sastre, Alberto | ProductoZ | Madrid |
| 333445555 | 10 | 10.0 | Campos Sastre, Alberto | Computación | Gijón |
| 333445555 | 20 | 10.0 | Campos Sastre, Alberto | Reorganización | Madrid |
| 999887777 | 30 | 30.0 | Jiménez Celaya, Alicia | Comunicaciones | Gijón |
| 999887777 | 10 | 10.0 | Jiménez Celaya, Alicia | Computación | Gijón |
| 987987987 | 10 | 35.0 | Pajares Morera, Luís | Computación | Gijón |
| 987987987 | 30 | 5.0 | Pajares Morera, Luís | Comunicaciones | Gijón |
| 987654321 | 30 | 20.0 | Sainz Oreja, Juana | Comunicaciones | Gijón |
| 987654321 | 20 | 15.0 | Sainz Oreja, Juana | Reorganización | Madrid |
| 888665555 | 20 | Null | Ochoa Paredes, Eduardo | Reorganización | Madrid |

la relación DEPARTAMENTO independientemente de que existan o no empleados trabajando en él, y siempre que un empleado se asigne a ese departamento se inserta una tupla en EMPLEADO.

**Anomalías de borrado.** El problema de las anomalías de borrado está relacionado con la segunda anomalía de inserción comentada anteriormente. Si eliminamos de EMP_DEPT una tupla empleado que representa al último que trabaja para un departamento dado, la información concerniente a ese departamento se pierde de la base de datos. Este problema no ocurre en la base de datos de la Figura 10.2 porque las tuplas DEPARTAMENTO se almacenan de forma separada.

**Anomalías de modificación.** En EMP_DEPT, si cambiamos el valor de uno de los atributos de un departamento particular (por ejemplo, el director del departamento 5), debemos actualizar las tuplas de todos los empleados que trabajan en ese departamento; en caso de no hacerlo, la base de datos se volverá inconsistente. Si falla la actualización de alguna tupla, el mismo departamento tendrá dos valores diferentes como director en distintas tuplas de empleado, lo que será incorrecto.[3]

Basándonos en las tres anomalías precedentes, podemos enunciar la siguiente directriz.

#### Directriz 2

Diseñar los esquemas de relación base de forma que no se presenten anomalías de inserción, borrado o actualización en las relaciones. En caso de que aparezca alguna de ellas, anótela claramente y asegúrese de que los programas que actualizan la base de datos operarán correctamente.

La segunda directriz es coherente, en cierto modo, con una reafirmación de la primera directriz. Podemos ver también la necesidad de una metodología más formal para evaluar si un diseño cumple estas directrices. Las Secciones de la 10.2 a la 10.4 muestran estas necesidades formales. Es importante indicar que estas directrices pueden, a veces, *tener que violarse* para *mejorar el rendimiento* de ciertas consultas. Por ejemplo, si una consulta importante recupera información relativa al departamento de un empleado junto con atributos de ese empleado, podría usarse el esquema EMP_DEPT como relación base. Sin embargo, deben indicarse y justificarse las anomalías de EMP_DEPT (por ejemplo, usando *triggers* o procedimientos almacenados que llevarían a cabo actualizaciones automáticas) de modo que, si se actualiza la relación base, no nos encontremos con incoherencias. En general, es aconsejable usar relaciones base que estén libres de anomalías y especificar vistas que incluyan las concatenaciones necesarias para recuperar los atributos que se referencian frecuentemente en las consultas. Esto reduce el número de términos JOIN especificados en la consulta, simplificando la escritura correcta de la consulta y, en muchos casos, mejorando el rendimiento.[4]

### 10.1.3 Valores NULL en las tuplas

En algunos diseños podemos agrupar muchos atributos en una relación "muy grande". Si muchos de los atributos no se aplican a todas las tuplas de la relación, nos encontraremos con muchos valores NULL en esas tuplas. Esto puede desperdiciar espacio de almacenamiento y puede inducir a problemas a la hora de entender el significado de los atributos con la especificación de operaciones JOIN a nivel lógico.[5] Otro problema con los NULL es cómo contabilizarlos cuando se aplican operaciones de agregación como COUNT o SUM. Las operaciones SELECT o JOIN implican comparaciones. Si hay presentes valores NULL, los resultados serán impredecibles.[6] Además, los NULL pueden tener múltiples interpretaciones:

- El atributo *no se aplica* a esta tupla.
- El valor de atributo de esta tupla es *desconocido*.
- El valor es *conocido pero está ausente*, es decir, aún no se ha grabado.

---

[3] Esto no es tan serio como otros problemas, ya que todas las tuplas pueden actualizarse con una sola sentencia SQL.

[4] El rendimiento de una consulta especificada en una vista que es la concatenación de varias relaciones base depende de cómo el DBMS implementa la vista. Muchos RDBMSs materializan las vistas usadas frecuentemente de forma que no se tengan que llevar a cabo las concatenaciones más habituales. El DBMS es responsable de la actualización de la vista materializada (ya sea inmediata o periódicamente) siempre que las relaciones base se modifiquen.

[5] Esto se debe a que las concatenaciones externas e internas producen resultados diferentes cuando existen valores NULL implicados en ellas. Los usuarios deben, por tanto, tener cuidado con los distintos significados de cada tipo de concatenación. Lo que resulta razonable para usuarios sofisticados, puede ser difícil para otros.

[6] En la Sección 8.5.1 presentamos varias comparaciones que implican valores NULL donde el resultado (en la lógica de tres valores) es TRUE, FALSE y UNKNOWN.

El tener la misma representación para todos los NULL compromete los diferentes significados que pueden tener. Por consiguiente, podemos establecer otra directriz.

#### Directriz 3

Hasta donde sea posible, evite situar en una relación base atributos cuyos valores sean NULL frecuentemente. En caso de no poderse evitar, asegúrese de que se aplican sólo en casos excepcionales y no los aplique a la mayor parte de las tuplas de la relación.

Utilizar el espacio eficientemente y evitar concatenaciones son los dos criterios principales que determinan si incluir las columnas que pueden tener valores NULL en una relación o tener una relación separada para esas columnas (con las columnas clave apropiadas). Por ejemplo, si sólo el 10 por ciento de los empleados tienen oficinas individuales, no es razón suficiente para la inclusión de un atributo NúmeroOficina en la relación EMPLEADO; en lugar de ello, se puede crear una relación OFICINAS_EMPS(DniEmpleado, NúmeroOficina) que incluya las tuplas de los empleados con oficinas individuales.

### 10.1.4 Generación de tuplas falsas

Considere los dos esquemas de relación EMP_LOCS y EMP_PROY1 de la Figura 10.5(a), la cual puede usarse en lugar de la relación simple EMP_PROY de la Figura 10.3(b). Una tupla en EMP_LOCS significa que el empleado cuyo nombre es NombreE trabaja en *algún proyecto* cuya localización es UbicaciónProyecto. Una tupla EMP_PROY1 se refiere al hecho de que el empleado cuyo Documento Nacional de Identidad es Dni trabaja un número de Horas por semana en el proyecto cuyo nombre, número y ubicación son NombreProyecto, NumProyecto y UbicaciónProyecto. La Figura 10.5(b) muestra el estado de relación de EMP_LOCS y EMP_PROY1 correspondiente a la relación EMP_PROY de la Figura 10.4, la cual se obtiene aplicando la operación PROYECCIÓN ($\pi$) adecuada a EMP_PROY [ignore por ahora las líneas discontinuas de la Figura 10.5(b)].

Supongamos que utilizamos EMP_PROY1 y EMP_LOCS como relaciones base en lugar de EMP_PROY. Esto produce un diseño de esquema incorrecto algo peculiar porque no podemos recuperar la información originaria de EMP_PROY desde EMP_PROY1 y EMP_LOCS. Si intentamos llevar a cabo una operación CONCATENACIÓN NATURAL en estas relaciones, el resultado produce muchas más tuplas que las existentes en el conjunto original de EMP_PROY. En la Figura 10.6, sólo se muestra la aplicación de la concatenación a las tuplas que están *por encima* de las líneas discontinuas de la Figura 10.5(b) (para reducir el tamaño de la relación resultante). Las tuplas adicionales que no se encontraban en EMP_PROY reciben el nombre de **tuplas falsas** (***spurious tuples***) porque representa información falsa que no es válida. Las tuplas falsas están marcadas con asteriscos ($*$) en la Figura 10.6.

No es aconsejable descomponer EMP_PROY en EMP_LOCS y EMP_PROY1 porque cuando deshacemos la CONCATENACIÓN usando una CONCATENACIÓN NATURAL, no obtenemos la información original correcta. Esto es así porque, en este caso, UbicaciónProyecto es el atributo que relaciona EMP_LOCS y EMP_PROY1, y no es ni una clave principal ni una *foreign key* en EMP_LOCS o EMP_PROY1. Ahora estamos en condiciones de definir otra directriz de diseño.

#### Directriz 4

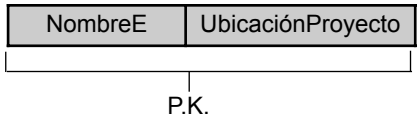
Diseñar los esquemas de relación de forma que puedan concatenarse con condiciones de igualdad en los atributos que son parejas de clave principal y *foreign key* de forma que se garantice que no se van a generar tuplas falsas. Evite las relaciones que contienen atributos coincidentes que no son combinaciones de *foreign key* y clave principal porque la concatenación de estos atributos puede producir tuplas falsas.

Esta directriz informal debe ser, obviamente, redefinida de una manera más adecuada. En el Capítulo 11 trataremos una condición formal llamada propiedad de reunión no aditiva que garantiza que ciertas concatenaciones no producen tuplas falsas.

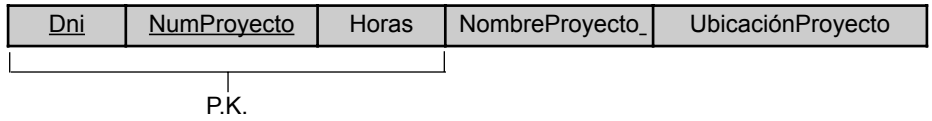
**Figura 10.5.** Diseño particularmente pobre de la relación EMP_PROY de la Figura 10.3(b). (a) Los dos esquemas de relación EMP_LOCS y EMP_PROY1. (b) El resultado de proyectar la extensión de EMP_PROY de la Figura 10.4 a las relaciones EMP_LOCS y EMP_PROY1.

**(a)**

**EMP_LOCS**




**EMP_PROJ1**




**(b)**

**EMP_LOCS**

| NombreE | Ubicación-Proyecto |
|---|---|
| Pérez Pérez, José | Valencia |
| Pérez Pérez, José | Surgarland |
| Ojeda Ordóñez, Fernando. | Madrid |
| Oliva Avezuela, Aurora | Valencia |
| Oliva Avezuela, Aurora | Surgarland |
| Campos Sastre, Alberto | Surgarland |
| Campos Sastre, Alberto | Madrid |
| Campos Sastre, Alberto | Gijón |
| Jiménez Celaya, Alicia | Gijón |
| Pajares Morera, Luis | Gijón |
| Sainz Oreja, Juana | Gijón |
| Sainz Oreja, Juana | Madrid |
| Ochoa Paredes, Eduardo | Madrid |

**EMP_PROJ1**

| Dni | NumProyecto | Horas | NombreProyecto | Ubicación-Proyecto |
|---|---|---|---|---|
| 123456789 | 1 | 32.5 | ProductoX | Valencia |
| 123456789 | 2 | 7.5 | ProductoY | Sevilla |
| 666884444 | 3 | 40.0 | ProductoZ | Madrid |
| 453453453 | 1 | 20.0 | ProductoX | Valencia |
| 453453453 | 2 | 20.0 | ProductoY | Sevilla |
| 333445555 | 2 | 10.0 | ProductoY | Sevilla |
| 333445555 | 3 | 10.0 | ProductoZ | Madrid |
| 333445555 | 10 | 10.0 | Computación | Gijón |
| 333445555 | 20 | 10.0 | Reorganización | Madrid |
| 999887777 | 30 | 30.0 | Comunicaciones | Gijón |
| 999887777 | 10 | 10.0 | Computación | Gijón |
| 987987987 | 10 | 35.0 | Computación | Gijón |
| 987987987 | 30 | 5.0 | Comunicaciones | Gijón |
| 987654321 | 30 | 20.0 | Comunicaciones | Gijón |
| 987654321 | 20 | 15.0 | Reorganización | Madrid |
| 888665555 | 20 | NULL | Reorganización | Madrid |

## 10.1.5 Resumen y explicación acerca de las directrices de diseño

En las Secciones de la 10.1.1 a la 10.1.4, hemos visto situaciones que provocan esquemas de relación problemáticos, y hemos propuesto unas directrices informales para definir un buen diseño relacional. Los problemas que hemos apuntado, que pueden detectarse sin la intervención de herramientas de análisis adicionales, son los siguientes:

- Anomalías que causan trabajo redundante durante la inserción y modificación de una relación, y que pueden causar pérdidas accidentales de información durante el borrado de la misma.
- Desaprovechamiento del espacio de almacenamiento debido a valores NULL y la dificultad de llevar a cabo operaciones de selección, agregación y concatenación debido a estos valores.

**Figura 10.6.** Resultado de aplicar una CONCATENACIÓN NATURAL a las tuplas que se encuentran por encima de las líneas discontinuas en EMP_PROY1 y EMP_LOCS de la Figura 10.5. Las tuplas falsas generadas aparecen marcadas con asteriscos.

| | Dni | NumProyecto | Horas | NombreProyecto | UbicaciónProyecto | NombreE |
|---|---|---|---|---|---|---|
| | 123456789 | 1 | 32.5 | ProductoX | Valencia | Pérez Pérez, José |
| * | 123456789 | 1 | 32.5 | ProductoX | Valencia | Oliva Avezuela, Aurora |
| | 123456789 | 2 | 7.5 | ProductoY | Sevilla | Pérez Pérez, José |
| * | 123456789 | 2 | 7.5 | ProductoY | Sevilla | Oliva Avezuela, Aurora |
| * | 123456789 | 2 | 7.5 | ProductoY | Sevilla | Campos Sastre, Alberto |
| | 666884444 | 3 | 40.0 | ProductoZ | Madrid | Ojeda Ordóñez, Fernando. |
| * | 666884444 | 3 | 40.0 | ProductoZ | Madrid | Campos Sastre, Alberto |
| * | 453453453 | 1 | 20.0 | ProductoX | Valencia | Pérez Pérez, José |
| | 453453453 | 1 | 20.0 | ProductoX | Valencia | Oliva Avezuela, Aurora |
| * | 453453453 | 2 | 20.0 | ProductoY | Sevilla | Pérez Pérez, José |
| | 453453453 | 2 | 20.0 | ProductoY | Sevilla | Oliva Avezuela, Aurora |
| * | 453453453 | 2 | 20.0 | ProductoY | Sevilla | Campos Sastre, Alberto |
| * | 333445555 | 2 | 10.0 | ProductoY | Sevilla | Pérez Pérez, José |
| * | 333445555 | 2 | 10.0 | ProductoY | Sevilla | Oliva Avezuela, Aurora |
| | 333445555 | 2 | 10.0 | ProductoY | Sevilla | Campos Sastre, Alberto |
| * | 333445555 | 3 | 10.0 | ProductoZ | Madrid | Ojeda Ordóñez, Fernando. |
| | 333445555 | 3 | 10.0 | ProductoZ | Madrid | Campos Sastre, Alberto |
| | 333445555 | 10 | 10.0 | Computación | Gijón | Campos Sastre, Alberto |
| * | 333445555 | 20 | 10.0 | Reorganización | Madrid | Ojeda Ordóñez, Fernando. |
| | 333445555 | 20 | 10.0 | Reorganización | Madrid | Campos Sastre, Alberto |

*
*
*

- Generación de datos incorrectos y falsos durante las concatenaciones en relaciones base incorrectamente relacionadas.

En el resto de este capítulo vamos a presentar conceptos y teorías formales que pueden utilizarse para definir de forma más precisa la *idoneidad* y la *mala calidad* de un esquema de relación *individual*. En primer lugar comentaremos la dependencia funcional como una herramienta de análisis. A continuación especificaremos las tres formas normales y la BCNF (Forma normal de Boyce-Codd, *Boyce-Codd Normal Form*) para un esquema de relación. En el Capítulo 11, definimos formas normales adicionales que están basadas en dependencias de tipos de datos adicionales llamadas dependencias multivalor y dependencias de concatenación.

# 10.2 Dependencias funcionales

El concepto básico más importante en la teoría de diseño de un esquema relacional es el de una dependencia funcional. En esta sección definiremos formalmente el concepto, mientras que en la Sección 10.3 veremos cómo usarlo para definir formas normales para los esquemas de relación.

## 10.2.1 Definición de dependencia funcional

Una dependencia funcional es una restricción que se establece entre dos conjuntos de atributos de la base de datos. Supongamos que nuestro esquema de base de datos relacional tiene $n$ atributos $A_1, A_2, \ldots, A_n$; pense-

mos que la base de datos completa está descrita por un único esquema de relación **universal** $R = \{A_1, A_2, \ldots, A_n\}$.[7] No sugerimos que vamos a almacenar la base de datos como una única tabla universal; usamos este concepto sólo en el desarrollo de la teoría formal de las dependencias de datos.[8]

> **Definición**. Una dependencia funcional, denotada por $X \rightarrow Y$, entre dos conjuntos de atributos $X$ e $Y$ que son subconjuntos de $R$, especifica una *restricción* en las posibles tuplas que pueden formar un estado de relación $r$ de $R$. La restricción dice que dos tuplas $t_1$ y $t_2$ en $r$ que cumplen que $t_1[X] = t_2[X]$, deben cumplir también que $t_1[Y] = t_2[Y]$.

Esto significa que los valores del componente $Y$ de una tupla de $r$ dependen de, o *están determinados por,* los valores del componente $X$; alternativamente, los valores del componente $X$ de una tupla únicamente (o **funcionalmente**) *determinan* los valores del componente $Y$. Decimos también que existe una dependencia funcional de $X$ hacia $Y$, o que $Y$ es **funcionalmente dependiente** de $X$. La abreviatura de dependencia funcional es **DF**, o **FD** o **f.d.** (del inglés, *functional dependency*). El conjunto de atributos $X$ recibe el nombre de **lado izquierdo** de la DF, mientras que $Y$ es el **lado derecho**.

Por tanto, $X$ determina funcionalmente $Y$ si para toda instancia r del esquema de relación R, no es posible que r tenga dos tuplas que coincidan en los atributos de X y no lo hagan en los atributos de Y. Observe lo siguiente:

- Si una restricción de $R$ indica que no puede haber más de una tupla con un valor $X$ concreto en cualquier instancia de relación $r(R)$, es decir, que $X$ es una **clave candidata** de $R$, se cumple que $X \rightarrow Y$ para cualquier subconjunto de atributos $Y$ de $R$ [ya que la restricción de clave implica que dos tuplas en cualquier estado legal $r(R)$ no tendrán el mismo valor de $X$].
- Si $X \rightarrow Y$ en $R$, esto no supone que $Y \rightarrow X$ en $R$.

Una dependencia funcional es una propiedad de la **semántica** o **significado de los atributos**. Los diseñadores de la base de datos utilizarán su comprensión de la semántica de los atributos de $R$ (esto es, cómo se relacionan unos con otros) para especificar las dependencias funcionales que deben mantenerse en *todos* los estados de relación (extensiones) $r$ de $R$. Siempre que la semántica de dos conjuntos de atributos de $R$ indique que debe mantenerse una dependencia funcional, la especificamos como una restricción. Las extensiones de relación $r(R)$ que satisfacen la restricción de dependencia funcional reciben el nombre de **estados de relación legales** (o **extensiones legales**) de $R$. Por tanto, el uso fundamental de las dependencias funcionales es describir más en profundidad un esquema de relación $R$ especificando restricciones de sus atributos que *siempre deben cumplirse.* Ciertas DF pueden especificarse sin hacer referencia a una relación específica. Por ejemplo, {Provincia, NumPermisoConducir} → Dni debe mantenerse para cualquier adulto que viva en España. También es posible que ciertas dependencias funcionales puedan dejar de existir en el mundo real si cambia la relación. Por ejemplo, en Estados Unidos la DF CódigoPostal → CodÁrea se utiliza como una relación entre los códigos postales y los códigos de los números telefónicos, pero con la proliferación de los códigos de área telefónica ya no es tan cierta.

Considere el esquema de relación EMP_PROY de la Figura 10.3(b); desde el punto de vista de la semántica de los atributos sabemos que deben mantenerse las siguientes dependencias funcionales:

**a.** Dni → NombreE

**b.** NumProyecto → {NombreProyecto, UbicaciónProyecto}

**c.** {Dni, NúmeroDpto} → Horas

---

[7] Este concepto de una relación universal es importante cuando se explican los algoritmos para el diseño de una base de datos relacional en el Capítulo 11.

[8] Esta presunción implica que cada atributo de la base de datos debe tener un nombre diferente. En el Capítulo 5 prefijamos nombres de atributo derivados de nombres de relación para lograr la unicidad siempre que los atributos de distintas relaciones tuvieran el mismo nombre.

**Figura 10.7.** Un estado de relación IMPARTIR con una *posible* dependencia funcional TEXTO → CURSO. Sin embargo, PROFESOR → CURSO no es posible.

**IMPARTIR**

| Profesor | Curso | Texto |
|---|---|---|
| Smith | Estructuras de datos | Bartram |
| Smith | Administración de datos | Martin |
| Hall | Compiladores | Hoffman |
| Brown | Estructuras de datos | Horowitz |

Estas dependencias funcionales especifican que (a) el valor del Documento Nacional de Identidad de un empleado (Dni) determina de forma inequívoca su nombre (NombreE), (b) el valor de un número de proyecto (NumProyecto) determina de forma única el nombre del mismo (NombreProyecto) y su ubicación (UbicaciónProyecto), y (c) una combinación de valores Dni y NumProyecto determina el número de horas semanales que el empleado ha trabajado en el proyecto (Horas). Alternativamente, decimos que NombreE está determinado funcionalmente por (o es funcionalmente dependiente de) Dni, o que *dado un Dni concreto, conocemos el valor de NombreE,* etc.

Una dependencia funcional es una *propiedad del esquema de relación R*, y no un estado de relación legal particular *r* de *R*. Por consiguiente, una DF *no puede* ser inferida automáticamente a partir de una extensión de relación *r*, sino que alguien que conozca la semántica de los atributos de *R* debe definirla explícitamente. Por ejemplo, la Figura 10.7 muestra un estado particular del esquema de relación IMPARTIR. Aunque a primera vista pudiéramos pensar que Texto → Curso, no podemos confirmarlo a menos que sepamos que se cumple *para todos los estados legales posibles* de IMPARTIR. Sin embargo, basta con demostrar un *único ejemplo en contra* para desautorizar una dependencia funcional. Por ejemplo, ya que 'Smith' enseña tanto 'Estructura de datos' como 'Administración de datos', podemos concluir que Profesor *no* determina funcionalmente a Curso.

La Figura 10.3 muestra una **notación diagramática** para visualizar las DFs: cada una de ellas aparece como una línea horizontal. Los atributos del lado izquierdo de la DF están conectados por líneas verticales a la línea que representa la DF, mientras que los del lado derecho lo están a los atributos mediante flechas que apuntan hacia ellos [véanse las Figuras 10.3(a) y 10.3(b)].

## 10.2.2 Reglas de inferencia para las dependencias funcionales

Decimos que *F* es el conjunto de dependencias funcionales especificadas en un esquema de relación *R*. Habitualmente, el diseñador del esquema especifica las dependencias funcionales que son *semánticamente obvias*; sin embargo, es habitual que otras muchas dependencias funcionales se encuentren en *todas* las instancias de relación legales entre los conjuntos de atributos que pueden derivarse y satisfacen las dependencias de *F*. Esas otras dependencias pueden *inferirse* o *deducirse* de las DF de *F*. En la vida real, es imposible especificar todas las dependencias funcionales posibles para una situación concreta. Por ejemplo, si cada departamento tiene un director, de manera que NúmeroDpto determina de forma única DniDirector (NúmeroDpto → DniDirector), y un director tiene un único número de teléfono TeléfonoDirector (DniDirector → TeléfonoDirector), entonces ambas dependencias juntas suponen que NúmeroDpto → TeléfonoDirector. Esto es una DF inferida y *no* tiene que declararse explícitamente. Por tanto, formalmente es útil definir un concepto llamado *clausura* (*closure*) que incluye todas las posibles dependencias que pueden inferirse de un conjunto *F* dado.

> **Definición.** Formalmente, el conjunto de todas las dependencias que incluyen *F*, junto con las dependencias que pueden inferirse de F, reciben el nombre de **clausuras** de *F*; está designada mediante $F^+$.

Por ejemplo, suponga que especificamos el siguiente conjunto *F* de dependencias funcionales obvias en el esquema de relación de la Figura 10.3(a):

F = {Dni → {NombreE, FechaNac, Dirección, NúmeroDpto},
NúmeroDpto → {NombreDpto, DniDirector} }

Las siguientes son algunas de las dependencias funcionales adicionales que se pueden *inferir* de *F*:

Dni → {NombreDpto, DniDirector}
Dni → Dni
NúmeroDpto → NombreDpto

Una DF $X \rightarrow Y$ es **inferida de** un conjunto de dependencias $F$ especificado en $R$ si $X \rightarrow Y$ se cumple en *todo* estado de relación legal $r$ de $R$; es decir, siempre que $r$ satisfaga todas las dependencias en $F$, $X \rightarrow Y$ también se cumple en $r$. La clausura $F^+$ de $F$ es el conjunto de todas las dependencias funcionales que pueden inferirse de $F$. Para determinar una manera sistemática de inferir dependencias, debemos descubrir un conjunto de **reglas de inferencia** que puedan usarse para deducir nuevas dependencias a partir de un conjunto de dependencias concreto. A continuación vamos a considerar algunas de estas reglas de inferencia. Usamos la notación $F \models X \rightarrow Y$ para indicar que la dependencia funcional $X \rightarrow Y$ se infiere del conjunto de dependencias funcionales $F$.

En la siguiente explicación, usaremos una notación abreviada para hablar de las dependencias funcionales. Por conveniencia, concatenamos las variables de atributo y eliminamos las comas. Por tanto, la DF $\{X,Y\} \rightarrow Z$ se expresa de forma abreviada como $XY \rightarrow Z$, mientras que la DF $\{X, Y, Z\} \rightarrow \{U, V\}$ se indica como $XYZ \rightarrow UV$. Las reglas de la RI1 a la RI6 son reglas de inferencia bien conocidas para las dependencias funcionales:

RI1 (regla reflexiva)[9]: Si $X \supseteq Y$, entonces $X \rightarrow Y$.

RI2 (regla de aumento)[10]: $\{X \rightarrow Y\} \models XZ \rightarrow YZ$.

RI3 (regla transitiva): $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$.

RI4 (regla de descomposición, o proyectiva): $\{X \rightarrow YZ\} \models X \rightarrow Y$.

RI5 (regla de unión, o aditiva): $\{X \rightarrow Y, X \rightarrow Z\} \models X \rightarrow YZ$.

RI6 (regla seudotransitiva): $\{X \rightarrow Y, WY \rightarrow Z\} \models WX \rightarrow Z$.

La regla reflexiva (RI1) especifica que un conjunto de atributos siempre se determina a sí mismo o cualquiera de sus subconjuntos, lo que es obvio. Ya que la RI1 genera dependencias que siempre son verdaderas, éstas reciben el nombre de *triviales*. Formalmente, una dependencia funcional $X \rightarrow Y$ es **trivial** si $X \supseteq Y$; en cualquier otro caso es **no trivial**. La regla de aumento (RI2) dice que añadir el mismo conjunto de atributos a ambos lados de una dependencia genera otra dependencia válida. Según la RI3, las dependencias funcionales son transitivas. La regla de descomposición (RI4) especifica que podemos eliminar atributos del lado derecho de una dependencia; si aplicamos esta regla repetidamente podemos descomponer la DF $X \rightarrow \{A_1, A_2, \ldots, A_n\}$ en el conjunto de dependencias $\{X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_n\}$. La regla de unión (RI5) nos permite realizar lo contrario: podemos combinar un conjunto de dependencias $\{X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_n\}$ en una única DF $X \rightarrow \{A_1, A_2, \ldots, A_n\}$.

Una *nota preventiva* acerca del uso de estas reglas. Aunque $X \rightarrow A$ y $X \rightarrow B$ implican $X \rightarrow AB$ por la regla de unión antes comentada, $X \rightarrow A$ e $Y \rightarrow B$ no *implican* que $XY \rightarrow AB$. Además, $XY \rightarrow A$ *no* implica necesariamente ni $X \rightarrow A$ ni $Y \rightarrow A$.

Cada una de las reglas de inferencia anteriores puede probarse a partir de la definición de dependencia funcional, bien por comprobación directa o bien **por contradicción**. Una comprobación por contradicción asume

[9] La regla reflexiva puede expresarse también como $X \rightarrow X$, es decir, cualquier conjunto de atributos se determina, funcionalmente, a sí mismo.

[10] La regla de aumento puede definirse también como $\{X \rightarrow Y\} \models XZ \rightarrow Y$, es decir, incrementar los atributos del lado izquierdo de una DF produce otra DF correcta.

que la regla no se cumple y muestra que ésta no es posible. Vamos a demostrar ahora que las tres primeras reglas son válidas. La segunda comprobación es por contradicción.

**Comprobación de la RI1.** Suponga que $X \supseteq Y$, y que dos tuplas $t_1$ y $t_2$ existen en alguna instancia de relación $r$ de $R$ como $t_1[X] = t_2[X]$. Por tanto, $t_1[Y] = t_2[Y]$ porque $X \supseteq Y$; por tanto, $X \rightarrow$ Y debe cumplirse en $r$.

**Comprobación de la RI2 por contradicción.** Asumimos que $X \rightarrow Y$ se cumple en una instancia de relación $r$ de $R$, pero no así $XZ \rightarrow YZ$. Entonces, deben existir dos tuplas $t_1$ y $t_2$ en $r$ tales que (1) $t_1[X] = t_2[X]$, (2) $t_1[Y] = t_2[Y]$, (3) $t_1[XZ] = t_2[XZ]$ y (4) $t_1[YZ]$ ? $t_2[YZ]$. Esto no es posible porque desde los puntos (1) y (3) deducimos el (5) $t_1[Z] = t_2[Z]$, y desde los puntos (2) y (5) deducimos el (6) $t_1[YZ] = t_2[YZ]$, contradiciendo el (4).

**Comprobación de la RI3.** Asumimos que (1) $X \rightarrow Y$ y (2) $Y \rightarrow Z$ se cumplen en una relación $r$. Entonces, por cada dos tuplas $t_1$ y $t_2$ en $r$ tales que $t_1[X] = t_2[X]$, debemos tener (3) $t_1[Y] = t_2[Y]$, desde la asunción del punto (1); por consiguiente, también debemos tener un punto (4) $t_1[Z] = t_2[Z]$, desde el (3) y asumiendo el (2); por tanto, $X \rightarrow Z$ debe cumplirse en $r$.

Usando argumentos de comprobación similares, podemos probar las reglas de inferencia de la RI4 a la RI6 y cualquier otra regla de inferencia válida. Sin embargo, una forma más simple de demostrar que una regla de inferencia es válida para las dependencias funcionales es probarla usando algunas de las que ya hemos visto que lo son. Por ejemplo, podemos comprobar las reglas RI4 a RI6 usando las reglas RI1, RI2 y RI3 de la siguiente forma.

**Comprobación de RI4 (usando RI1, RI2 y RI3).**

1. $X \rightarrow YZ$ (dada).
2. $YZ \rightarrow Y$ (usando RI1 y sabiendo que $YZ \supseteq Y$).
3. $X \rightarrow Y$ (usando RI3 en 1 y 2).

**Comprobación de RI5 (usando RI1, RI2 y RI3).**

1. $X \rightarrow Y$ (dada).
2. $X \rightarrow Z$ (dada).
3. $X \rightarrow XY$ (usando RI2 en 1 aumentando con $X$; observe que $XX = X$).
4. $XY \rightarrow YZ$ (usando RI2 en 2 aumentando con $Y$).
5. $X \rightarrow YZ$ (usando RI3 en 3 y 4).

**Comprobación de RI6 (usando las RI1, RI2 y RI3).**

1. $X \rightarrow Y$ (dada).
2. $WY \rightarrow Z$ (dada).
3. $WX \rightarrow WY$ (usando RI2 en 1 aumentando con $W$).
4. $WX \rightarrow Z$ (usando RI3 en 3 y 2).

Armstrong(1974) demostró que las reglas de inferencia de la RI1 a la RI3 son sólidas y completas. Por **sólida** queremos decir que, dado un conjunto de dependencias funcionales $F$ especificado en un esquema de relación $R$, cualquier dependencia que podamos inferir de $F$ usando RI1, RI2 y RI3 se cumple en cada estado de relación $r$ de $R$ que *satisfaga* las *dependencias* de $F$. Por **completa** decimos que, usando las tres primeras reglas repetidamente para inferir dependencias hasta que ya no se pueda determinar ninguna más, se genera el conjunto completo de *todas las dependencias posibles* que pueden inferirse a partir de $F$. En otras palabras, el conjunto de dependencias $F^+$, que hemos llamado **clausura** de $F$, pueden determinarse a partir de $F$ usando

sólo las reglas de inferencia de la RI1 a la RI3. Estas tres reglas se conocen como **reglas de inferencia de Armstrong**.[11]

Habitualmente, los diseñadores de bases de datos especifican, en primer lugar, el conjunto de dependencias funcionales $F$ que pueden determinarse fácilmente a partir de la semántica de los atributos de $R$; por tanto, se usan las reglas RI1, RI2 y RI3 para inferir dependencias funcionales adicionales que también se almacenarán en $R$. Una forma semántica para determinar estas dependencias funcionales adicionales es determinar, en primer lugar, cada conjunto de atributos $X$ que aparece en la parte izquierda de alguna dependencia funcional de $F$ para, a continuación, determinar el conjunto de *todos los atributos* que son dependientes en $X$.

**Definición**. Para cada conjunto de atributos $X$ como éste, determinamos el conjunto $X^+$ de atributos que están funcionalmente determinados por $X$ basados en $F$; $X^+$ recibe el nombre de **clausura de $X$ bajo $F$**. Puede usarse el Algoritmo 10.1 para calcular $X^+$.

**Algoritmo 10.1.** Determinación de $X^+$, la clausura de $X$ bajo $F$:

```
X+ := X;
    repetir
        antiguaX+ := X+;
        por cada dependencia funcional Y → Z en F ejecutar
            si X+ ⊇ Y entonces X+ := X+ ∪ Z;
hasta que (X+ = antiguaX+);
```

El Algoritmo 10.1 empieza asignado a $X^+$ todos los atributos de $X$. Según RI1, sabemos que todos esos atributos son funcionalmente dependientes de $X$. Usando las reglas de inferencia RI3 y RI4, añadimos atributos a $X^+$, usando cada dependencia funcional en $F$. Continuamos por todas las dependencias de $F$ (el bucle *repetir*) hasta que no se añadan más atributos a $X^+$ *durante un ciclo completo* (del bucle *por cada*) a través de las dependencias de $F$. Por ejemplo, consideremos el esquema de relación EMP_PROY de la Figura 10.3(b); según la semántica de los atributos, especificamos el siguiente conjunto $F$ de dependencias funcionales que deben cumplirse en EMP_PROY:

$$F = \{\text{Dni} \rightarrow \text{NombreE},\ \text{NumProyecto} \rightarrow \{\text{NombreProyecto}, \text{UbicaciónProyecto}\},\ \{\text{Dni}, \text{NumProyecto}\} \rightarrow \text{Horas}\}$$

Usando el Algoritmo 10.1, calculamos los siguientes conjuntos de clausura con respecto a $F$:

```
{Dni} += {Dni, NombreE}
{NumProyecto} += {NumProyecto, NombreProyecto, UbicaciónProyecto}
{Dni, NumProyecto} += {Dni, NumProyecto, NombreE, NombreProyecto, UbicaciónProyecto, Horas}
```

Intuitivamente, el conjunto de atributos del lado derecho de cada una de las líneas anteriores representa a todos los atributos que son funcionalmente dependientes del conjunto de atributos del lado izquierdo en base al conjunto $F$ dado.

### 10.2.3 Equivalencia de los conjuntos de dependencias funcionales

En esta sección vamos a tratar la equivalencia de dos conjuntos de dependencias funcionales. En primer lugar vamos a ver algunas definiciones preliminares.

[11] En la actualidad se conocen como **axiomas de Armstrong**. En un sentido estrictamente matemático, los *axiomas* (hechos dados) son las dependencias funcionales de $F$, ya que asumimos que son correctos, mientras que las reglas RI1, RI2 y RI3 son las *reglas de inferencia* para determinar nuevas dependencias funcionales (nuevos hechos).

**Definición.** Un conjunto de dependencias funcionales $F$ se dice que **cubre** a otro conjunto de dependencias funcionales $E$ si toda DF de $E$ está también en $F^+$, es decir, si toda dependencia de $E$ puede ser inferida a partir de $F$; alternativamente, podemos decir que $E$ está **cubierto por** $F$.

**Definición.** Dos conjuntos de dependencias funcionales $E$ y $F$ son **equivalentes** si $E^+ = F^+$. Por consiguiente, equivalencia significa que cada DF de $E$ puede inferirse a partir de $F$, y viceversa; es decir, $E$ es equivalente a $F$ si se cumplen las condiciones $E$ cubre a $F$ *y* $F$ cubre a $E$.

Podemos determinar si $F$ cubre a $E$ calculando $X^+$ *respecto a* $F$ para cada DF $X \rightarrow Y$ *en* $E$, y comprobando después si $X^+$ incluye todos los atributos de $Y$. Si es el caso para *toda* DF en $E$, entonces $F$ cubre a $E$. Determinamos si $E$ y $F$ son equivalentes comprobando que $E$ cubre a $F$ y que $F$ cubre a $E$.

### 10.2.4 Conjuntos mínimos de dependencias funcionales

Informalmente, una **cobertura mínima** de un conjunto de dependencias funcionales $E$ es un conjunto de dependencias funcionales $F$ que satisface la propiedad de que cada dependencia de $E$ está en la clausura $F^+$ de $F$. Además, esta propiedad se pierde si se elimina cualquier dependencia del conjunto $F$; $F$ no debe tener redundancias, y las dependencias de $F$ están en una forma estándar. Podemos definir formalmente que un conjunto de dependencias funcionales $F$ es **mínimo** si satisface las siguientes condiciones:

1. Toda dependencia en $F$ tiene un único atributo en su lado derecho.
2. No podemos reemplazar ninguna dependencia $X \rightarrow A$ de $F$ por otra dependencia $Y \rightarrow A$, donde $Y$ es un subconjunto propio de $X$, y seguir teniendo un conjunto de dependencias equivalente a $F$.
3. No podemos eliminar ninguna dependencia de $F$ y seguir teniendo un conjunto de dependencias equivalente a $F$.

Podemos concebir un conjunto mínimo de dependencias como un conjunto de dependencias de una *forma estándar* o *canónica* y *sin redundancias.* La condición 1 sólo representa cada dependencia en forma canónica con un único atributo en el lado derecho.[12] Las condiciones 2 y 3 garantizan que no habrá redundancia en las dependencias ya sea por tener atributos redundantes en el lado izquierdo de una dependencia (condición 2) o por tener una dependencia que puede inferirse a partir del renombrado de las DF en $F$ (condición 3).

**Definición**. Una cobertura mínima de un conjunto de dependencias funcionales $E$ es un conjunto mínimo de dependencias (en forma canónica estándar y sin redundancia) equivalente a $E$. Con el Algoritmo 10.2, siempre podemos buscar, *al menos*, *una* cobertura mínima $F$ por cada conjunto de dependencias $E$.

Si hay varios conjuntos de DF cualificados como coberturas mínimas de $E$ por la definición anterior, es costumbre utilizar criterios adicionales para minimizar. Por ejemplo, podemos elegir el conjunto mínimo con el *menor número de dependencias* o con la *longitud total* más pequeña (la longitud total de un conjunto de dependencias se calcula concatenando las dependencias y tratándolas como una cadena de caracteres larga).

**Algoritmo 10.2**. Localizar una cobertura mínima $F$ para un conjunto de dependencias funcionales $E$.

1. Establecer $F := E$.
2. Reemplazar cada dependencia funcional $X \rightarrow \{A_1, A_2, \ldots, A_n\}$ en $F$ por las $n$ dependencias funcionales $X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_n$.
3. Por cada dependencia funcional $X \rightarrow A$ en $F$
   por cada atributo $B$ que es un elemento de $X$
   si $\{ \{F - \{X \rightarrow A\}\} \cup \{ (X - \{B\}) \rightarrow \mathrm{A}\} \}$ es equivalente a $F$,
   entonces reemplazar $X \rightarrow A$ por $(X - \{B\}) \rightarrow A$ en $F$.

[12] Es una forma estándar de simplificar las condiciones y los algoritmos que garantizan la no existencia de redundancia en $F$. Con la regla de inferencia RI4, podemos convertir una dependencia simple con múltiples atributos en su parte derecha en un conjunto de dependencias con atributos simples en la parte derecha.

**4.** Por cada dependencia funcional $X \rightarrow A$ sobrante en $F$
si $\{F - \{X \rightarrow A\}\}$ es equivalente a $F$,
entonces eliminar $X \rightarrow A$ de $F$.

Para ilustrar el algoritmo anterior, podemos utilizar lo siguiente:

Partiendo del siguiente conjunto de DF, $E : \{B \rightarrow A, D \rightarrow A, AB \rightarrow D\}$, tenemos que localizar su cobertura mínima.

- Todas las dependencias anteriores están en forma canónica; ya hemos completado el paso 1 del Algoritmo 10.2 y podemos proceder con el paso 2. En este paso necesitamos determinar si $AB \rightarrow D$ tiene algún atributo redundante en el lado izquierdo, es decir, ¿puede sustituirse por $B \rightarrow D$ o $A \rightarrow D$?
- Ya que B $\rightarrow$ A, aumentando con $B$ en ambos lados (RI2), tenemos que $BB \rightarrow AB$, o $B \rightarrow AB$ (i). Sin embargo, $AB \rightarrow D$ como se ha dado (ii).
- Por la regla transitiva (RI3), obtenemos de (i) y (ii), $B \rightarrow D$. Por consiguiente $AB \rightarrow D$ podría sustituirse por $B \rightarrow D$.
- Ahora tenemos un conjunto equivalente al $E$ original, el $E'$ : $\{B \rightarrow A, D \rightarrow A, B \rightarrow D\}$. Ya no son posibles más reducciones en el paso 2, ya que todas las DF tienen un único atributo en el lado izquierdo.
- En el paso 3 buscamos una DF redundante en $E'$. Usando la regla transitiva en $B \rightarrow D$ y $D \rightarrow A$, derivamos $B \rightarrow A$. Por tanto, $B \rightarrow A$ es redundante en $E'$ y puede eliminarse.
- Así pues, la cobertura mínima de $E$ es $\{B \rightarrow D, D \rightarrow A\}$.

En el Capítulo 11 veremos cómo pueden sintetizarse las relaciones de un conjunto de dependencias $E$ localizando primero la cobertura mínima $F$ para $E$.

## 10.3 Formas normales basadas en claves principales

Una vez estudiadas las dependencias funcionales y algunas de sus propiedades, estamos preparados para usarlas a la hora de especificar algunos aspectos de la semántica de un esquema de relaciones. Asumimos que contamos con un conjunto de dependencias funcionales para cada relación, y que cada una de estas relaciones dispone de una clave principal; esta información combinada con las verificaciones (condiciones) de las formas normales conduce al *proceso de normalización* del diseño del esquema relacional. Los proyectos de diseño relacional más prácticos siguen una de estas aproximaciones:

- Realizar un diseño de esquema conceptual usando un modelo conceptual como el ER o el EER y asignar el diseño conceptual a un conjunto de relaciones.
- Diseñar las relaciones en base a un conocimiento externo derivado de una implementación existente de ficheros o formularios o informes.

Siguiendo alguno de estos métodos, es útil evaluar las relaciones de idoneidad y descomponerlas todo lo necesario hasta obtener formas normales elevadas usando la teoría de normalización presentada en este capítulo y en el siguiente. En esta sección nos centraremos en las tres primeras formas normales de un esquema de relaciones y en la intuición que se esconde tras ellas, y comentaremos el modo en que fueron desarrolladas históricamente. En la Sección 10.4 veremos las definiciones más generales de estas formas normales, las cuales tienen en cuenta todas las claves candidatas de una relación en lugar de considerar sólo la clave principal.

Empezaremos tratando de manera informal las formas normales y la motivación que se esconde tras su desarrollo, sin olvidarnos de revisar algunas de las definiciones propuestas en el Capítulo 5 y que son necesarias aquí. A continuación trataremos la primera forma normal (1FN) en la Sección 10.3.4, y presentaremos las definiciones de la segunda (2FN) y tercera (3FN) formas normales, que están basadas en claves principales, en las Secciones 10.3.5 y 10.3.6, respectivamente.

### 10.3.1 Normalización de relaciones

El proceso de normalización, tal y como fue propuesto en un principio por Codd (1972a), hace pasar un esquema de relación por una serie de comprobaciones para *certificar* que satisface una determinada **forma normal**. El proceso, que sigue un método descendente evaluando cada relación contra el criterio de las formas normales y descomponiendo las relaciones según sea necesario, puede considerarse como un *diseño relacional por análisis*. Inicialmente, Codd propuso tres formas normales: la primera, la segunda y la tercera. Una definición más estricta de la 3FN, llamada BCNF (Forma normal Boyce-Codd, *Boyce-Codd Normal Form*) fue propuesta posteriormente por Boyce y Codd. Todas estas formas normales estaban basadas en una única herramienta analítica: las dependencias funcionales entre los atributos de una relación. Más adelante se propusieron una cuarta (4FN) y una quinta (5FN) formas normales basadas en los conceptos de dependencias multivalor y dependencias de concatenación, respectivamente; ambas se explican en el Capítulo 11. Al final de dicho capítulo, comentaremos también el modo en que las relaciones 3FN pueden sintetizarse a partir de un conjunto de DFs dado. Este acercamiento recibe el nombre de *diseño relacional por síntesis*.

La **normalización de datos** puede considerarse como un proceso de análisis de un esquema de relación, basado en sus DF y sus claves principales, para obtener las propiedades deseables de (1) minimizar la redundancia y (2) minimizar las anomalías de inserción, borrado y actualización comentadas en la Sección 10.1.2. Los esquemas de relación no satisfactorios que no cumplen ciertas condiciones (las **pruebas de forma normal**) se decomponen en esquemas de relación más pequeños que cumplen esas pruebas y que, por consiguiente, cuentan con las propiedades deseables. De este modo, el procedimiento de normalización ofrece a los diseñadores de bases de datos lo siguiente:

- Un marco formal para el análisis de los esquemas de relación basado en sus claves y en las dependencias funcionales entre sus atributos.
- Una serie de pruebas de forma normal que pueden efectuarse sobre esquemas de relación individuales, de modo que la base de datos relacional pueda **normalizarse** hasta el grado deseado.

  **Definición.** La **forma normal** de una relación hace referencia a la forma normal más alta que cumple, e indica por tanto el grado al que ha sido normalizada.

Las formas normales, consideradas *aisladas* de otros factores, no garantizan un buen diseño de base de datos. Generalmente, no basta con comprobar por separado que cada esquema de relación de la base de datos está, digamos, en BCNF o en 3FN. En lugar de ello, el proceso de normalización por descomposición debe confirmar también la existencia de las propiedades adicionales que los esquemas relacionales, en conjunto, deben poseer. Dos de estas propiedades son las siguientes:

- La **propiedad de reunión sin pérdida** o **reunión no aditiva**, que garantiza que no se presentará el problema de las tuplas falsas comentado en la Sección 10.1.4, respecto a los esquema de relación creados después de la descomposición.
- La **propiedad de conservación de las dependencias**, que asegura que todas las dependencias funcionales están representadas en alguna relación individual resultante tras la descomposición.

La propiedad de reunión no aditiva es extremadamente crítica y **debe cumplirse a cualquier precio**, mientras que la de conservación de las dependencias, aunque deseable, puede sacrificarse a veces, tal y como veremos en la Sección 11.1.2. Aplazaremos al Capítulo 11 la presentación de los conceptos formales y las técnicas que garantizan las dos propiedades anteriores.

### 10.3.2 Uso práctico de las formas normales

Los proyectos de diseño más prácticos obtienen información de diseños de bases de datos previos, de modelos heredados o de ficheros existentes. La normalización se lleva a la práctica de forma que los diseños resultantes sean de máxima calidad y cumplan las propiedades deseables antes comentadas. Aunque se han

definido varias formas normales superiores, como la 4FN y la 5FF que veremos en el Capítulo 11, su utilidad práctica es cuestionable cuando las restricciones en las que se basan son difíciles de comprender o detectar por parte de los diseñadores de la base de datos y son los usuarios los que deben descubrirlas. Así, el diseño de base de datos, tal y como se realiza en la actualidad en la industria, presta especial atención a la normalización hasta la 3FN, la BCNF o la 4FN.

Otro punto que conviene resaltar es que los diseñadores de bases de datos *no necesitan* normalizar hasta la forma normal más alta posible. Por razones de rendimiento, las relaciones podrían dejarse en un estado de normalización menor, como el 2FN, como se ha comentado al final de la Sección 10.1.2.

> **Definición.** El proceso para almacenar la concatenación de relaciones de forma normal superiores como relación base (que se encuentra en una forma normal inferior) recibe el nombre de **desnormalización**.

### 10.3.3 Definiciones de claves y atributos que participan en las claves

Antes de avanzar más, vamos a revisar de nuevo las definiciones de las claves de un esquema de relación mostradas en el Capítulo 5.

> **Definición.** Una **superclave** de un esquema de relación $R = \{A_1, A_2, \ldots, A_n\}$ es un conjunto de atributos $S \subseteq R$ con la propiedad de que no habrá un par de tuplas $t_1$ y $t_2$ en ningún estado de relación permitido r de $R$ tal que $t_1[S] = t_2[S]$. Una **clave** $K$ es una superclave con la propiedad adicional de que la eliminación de cualquier atributo de $K$ provocará que $K$ deje de ser una superclave.

La diferencia entre una clave y una superclave es que la primera tiene que ser mínima, es decir, si tenemos una clave $K = \{A_1, A_2, \ldots, A_k\}$ de $R$, entonces $K - \{A_i\}$ no es una clave de $R$ para ningún $A_i$, 1 ? $i$ ? $k$. En la Figura 10.1, {Dni} es una clave de EMPLEADO, mientras que {Dni}, {Dni, NombreE}, {Dni, NombreE, FechaNac} y cualquier otro conjunto de atributos que incluya Dni, son superclaves.

Si un esquema de relación tiene más de una clave, cada una de ellas se denomina **clave candidata**. Una de ellas se elige *arbitrariamente* como **clave principal**, mientras que el resto son claves secundarias. Todo esquema de relación debe contar con una clave principal. En la Figura 10.1, {Dni} es la única clave candidata de EMPLEADO, por lo que también será la clave principal.

> **Definición.** Un atributo del esquema de relación $R$ recibe el nombre de **atributo primo** de $R$ si es miembro de *alguna de las claves candidatas* de $R$. Un atributo es **no primo** si no es miembro de ninguna clave candidata.

En la Figura 10.1, tanto Dni como NúmeroDpto son atributos primos de TRABAJA_EN, mientras que el resto de atributos de TRABAJA_EN no lo son.

Ahora vamos a presentar las tres primeras formas normales: 1FN, 2FN y 3FN. Todas ellas fueron propuestas por Codd (1972a) como una secuencia para alcanzar el estado deseable de relaciones 3FN tras pasar por los estados intermedios 1FN y 2FN en caso de ser necesario. Como veremos, la 2FN y la 3FN atacan diferentes problemas. Sin embargo, por motivos históricos, es habitual seguirlas en ese orden; así pues, por definición, una relación 3FN *también satisface* la 2FN.

### 10.3.4 Primera forma normal

La **primera forma normal (1FN)** está considerada como una parte de la definición formal de una relación en el modelo relacional básico;[13] históricamente, fue definida para prohibir los atributos multivalor, los

[13] Esta condición desaparece en el modelo relacional anidado y en los ORDBMS (*Sistemas de objetos relacionales, Object-Relational Systems*), los cuales permiten relaciones no normalizadas (consulte el Capítulo 22).

atributos compuestos y sus combinaciones. Afirma que el dominio de un atributo sólo debe incluir valores *atómicos* (simples, indivisibles) y que el valor de cualquier atributo en una tupla debe ser un *valor simple* del dominio de ese atributo. Por tanto, 1FN prohíbe tener un conjunto de valores, una tupla de valores o una combinación de ambos como valor de un atributo para una *tupla individual*. En otras palabras, 1FN prohíbe las *relaciones dentro de las relaciones* o las *relaciones como valores de atributo dentro de las tuplas*. Los únicos valores de atributo permitidos por 1FN son los **atómicos** (o **indivisibles**).

Considere el esquema de relación DEPARTAMENTO de la Figura 10.1, cuya clave principal es NúmeroDpto, y suponga que lo ampliamos incluyendo el atributo UbicacionesDpto [véase la Figura 10.8(a)]. Asumimos que cada departamento puede tener *un número de* localizaciones. En la Figura 10.8 podemos ver el esquema DEPARTAMENTO y un estado de relación de ejemplo. Como puede apreciarse, esto no está en la 1FN porque UbicacionesDpto no es un atributo atómico, como queda demostrado por la primera tupla de la Figura 10.8(b). Hay dos formas de ver el atributo UbicacionesDpto:

- El dominio de UbicacionesDpto contiene valores atómicos, pero algunas tuplas pueden tener un conjunto de estos valores. En este caso, UbicacionesDpto no es funcionalmente dependiente de la clave principal NúmeroDpto.
- El dominio de UbicacionesDpto contiene conjuntos de valores y, por tanto, no es atómico. En este caso, NúmeroDpto → UbicacionesDpto porque cada conjunto está considerado como un miembro único de un dominio de atributo.[14]

En cualquier caso, la relación DEPARTAMENTO de la Figura 10.8 no está en 1FN; de hecho, no está calificada ni como relación según nuestra definición de relación de la Sección 5.1. Existen tres formas principales de alcanzar la primera forma normal para una relación como ésta:

1. Eliminar el atributo UbicacionesDpto que viola la 1FN y colocarlo en una relación aparte LOCALIZACIONES_DPTO junto con la clave principal NúmeroDpto de DEPARTAMENTO. La clave principal de esta relación es la combinación {NúmeroDpto, UbicaciónDpto}, como puede verse en la Figura 10.2. En LOCALIZACIONES_DPTO existe una tupla distinta por *cada localización* de un departamento. Esto descompone la no relación 1FN en dos relaciones 1FN.
2. Expandir la clave de forma que exista una tupla separada en la relación DEPARTAMENTO original por cada localización de un DEPARTAMENTO, como puede verse en la Figura 10.8(c). En este caso, la clave principal resulta de la combinación {NúmeroDpto, UbicaciónDpto}. Esta solución tiene la desventaja de introducir *redundancia* en la relación.
3. Si se conoce un *número máximo de valores* para el atributo (por ejemplo, si se sabe que *pueden existir, al menos, tres localizaciones* para un departamento), sustituir el atributo UbicacionesDpto por tres atributos atómicos: UbicaciónDpto1, UbicaciónDpto2 y UbicaciónDpto3. Esta solución tiene el inconveniente de introducir *valores NULL* en el caso de que algún departamento tenga menos de tres localizaciones. Introduce además una semántica confusa acerca de la ordenación entre los valores de localización, lo que no era lo que originalmente se pretendía. La consulta sobre este atributo se hace más complicada; por ejemplo, considere cómo escribir la siguiente consulta: *enumere los departamentos que tienen 'Valencia' como una de sus localizaciones*.

De las tres posibles soluciones, la primera es la que se considera como la mejor porque no introduce redundancia y es completamente general, sin estar limitada por un número máximo de valores. De hecho, si elegimos la segunda solución, el proceso posterior de normalización nos conducirá a la primera solución.

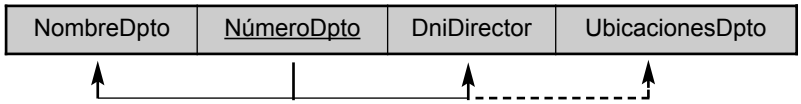
La primera forma normal también inhabilita los atributos multivalor que son compuestos. Reciben el nombre de **relaciones anidadas** porque cada tupla puede tener una relación *dentro de ella.* La Figura 10.9 muestra a

[14] En este caso, podemos considerar que el dominio de UbicacionesDpto es el **conjunto potencia** del conjunto de localizaciones individuales, es decir, el dominio está compuesto por todos los posibles subconjuntos del conjunto de ubicaciones individuales.

**Figura 10.8.** Normalización en 1FN. (a) Un esquema de relación que no está en 1FN. (b) Ejemplo de un estado de relación DEPARTAMENTO. (c) Versión 1FN de la misma relación con redundancia.

**(a)**

**DEPARTAMENTO**




**(b)**

**DEPARTAMENTO**

| NombreDpto | NúmeroDpto | DniDirector | UbicacionesDpto |
|---|---|---|---|
| Investigación | 5 | 333445555 | {Valencia, Sevilla, Madrid} |
| Administración | 4 | 987654321 | {Gijón} |
| Sede central | 1 | 888665555 | {Madrid} |

**(c)**

**DEPARTAMENTO**

| NombreDpto | NúmeroDpto | DniDirector | UbicaciónDpto |
|---|---|---|---|
| Investigación | 5 | 333445555 | Valencia |
| Investigación | 5 | 333445555 | Sevilla |
| Investigación | 5 | 333445555 | Madrid |
| Administración | 4 | 987654321 | Gijón |
| Sede central | 1 | 888665555 | Madrid |

qué podría parecerse la relación EMP_PROY en caso de permitirse la anidación. Cada tupla representa una entidad empleado, y una relación PROYS(NumProyecto, Horas) *dentro de cada tupla* implica los proyectos de cada empleado y las horas semanales dedicadas a cada proyecto. El esquema de esta relación EMP_PROY puede representarse de la siguiente manera:

EMP_PROY(Dni, NombreE, {PROYS(NumProyecto, Horas)})

Las llaves { } identifican el atributo PROYS como de tipo multivalor, y enumeramos los atributos componente que forman PROYS entre paréntesis ( ). Las últimas tendencias para el soporte de objetos complejos (consulte el Capítulo 20) y datos XML (consulte el Capítulo 27) intentan permitir y formalizar relaciones anidadas dentro de sistemas de bases de datos relacionales, las cuales fueron prohibidas en principio por la 1FN.

Observe que Dni es la clave principal de la relación EMP_PROY en las Figuras 10.9(a) y (b), mientras que NumProyecto es la clave **parcial** de la relación anidada, esto es, dentro de cada tupla, la relación anidada debe contar con valores únicos de NumProyecto. Para normalizar esto en 1FN, llevamos los atributos de la relación anidada a una nueva relación y *propagamos la clave principal*; la clave principal de la nueva relación combinará la clave parcial con la clave principal de la relación original. La descomposición y la propagación de la clave principal producen los esquemas EMP_PROY1 y EMP_PROY2 [véase la Figura 10.9(c)].

Este procedimiento puede aplicarse recursivamente a una relación con varios niveles de anidamiento para **desanidar** la relación en un conjunto de relaciones 1FN. Esto resulta útil para convertir un esquema de relación no normalizado con varios niveles de anidamiento en relaciones 1FN. La existencia de más de un atributo multivalor en una relación debe ser manipulado con cuidado. Como ejemplo, considere la siguiente relación no 1FN:

PERSONA (Dn#, {NumPermisoConducir#} , {Teléfono#})

**Figura 10.9.** Normalización de relaciones anidadas en 1FN. (a) Esquema de la relación EMP_PROY con un atributo de *relación anidada* PROYS. (b) Ejemplo de extensión de la relación EMP_PROY mostrando relaciones anidadas dentro de cada tupla. (c) Descomposición de EMP_PROY en las relaciones EMP_PROY1 y EMP_PROY2 por la propagación de la clave principal.

**(a)**

**EMP_PROY**

| | | Proyectos | |
|---|---|---|---|
| Dni | NombreE | NumProyecto | Horas |

**(b)**

**EMP_PROY**

| Dni | NombreE | NumProyecto | Horas |
|---|---|---|---|
| 123456789 | Pérez Pérez, José | 1<br>2 | 32.5<br>7.5 |
| 666884444 | Ojeda Ordóñez, Fernando. | 3 | 40.0 |
| 453453453 | Oliva Avezuela, Aurora | 1<br>2 | 20.0<br>20.0 |
| 333445555 | Campos Sastre, Alberto | 2<br>3<br>10<br>20 | 10.0<br>10.0<br>10.0<br>10.0 |
| 999887777 | Zelaya, Alicia J. | 30<br>10 | 30.0<br>10.0 |
| 987987987 | Pajares Morera, Luis | 10<br>30 | 35.0<br>5.0 |
| 987654321 | Sainz Oreja, Juana | 20<br>30 | 20.0<br>15.0 |
| 888665555 | Ochoa Paredes, Eduardo | 20 | NULL |

**(c)**

**EMP_PROY1**

| Dni | NombreE |
|---|---|

**EMP_PROY2**

| Dni | NumProyecto | Horas |
|---|---|---|

Esta relación representa el hecho de que una persona tiene varios vehículos y teléfonos. Si se sigue la segunda opción antes comentada, obtendremos una relación repleta de claves:

PERSONA_EN_1FN (Dn#, NumPermisoConducir#, Teléfono#)

Para evitar introducir relaciones extrañas entre el NumPermisoConducir# y Teléfono#, están representadas todas las posibles combinaciones de valores por cada Dn#, dando lugar a redundancias. Esto conduce a los problemas manipulados por las dependencias multivalor y 4FN, de las que hablaremos en el Capítulo 11. La forma correcta de tratar con los dos atributos multivalor de PERSONA es descomponerlos en dos relaciones separadas, usando la primera estrategia comentada anteriormente: P1(Dn#, NumPermisoConducir#) y P2(Dn#, Teléfono#).

### 10.3.5 Segunda forma normal

La **segunda forma normal** (**2FN**) está basada en el concepto de *dependencia funcional total*. Una dependencia funcional $X \rightarrow Y$ es **total** si la eliminación de cualquier atributo $A$ de $X$ implica que la dependencia deje de ser válida, es decir, para cualquier atributo $A \in X$, $(X - \{A\})$ *no* determina funcionalmente a $Y$. Una dependencia funcional $X \rightarrow Y$ es **parcial** si al eliminarse algún atributo $A \in X$ de $X$ la dependencia sigue siendo válida, es decir, para algún $A \in X$, $(X - \{A\}) \rightarrow Y$. En la Figura 10.3(b), {Dni, NumProyecto} → Horas es una dependencia completa (ni Dni → Horas ni NumProyecto → Horas son válidas). Sin embargo, la dependencia {Dni, NumProyecto} → NombreE es parcial porque se cumple Dni → NombreE.

> **Definición.** Un esquema de relación $R$ está en 2FN si todo atributo no primo $A$ en $R$ es *completa y funcionalmente dependiente* de la clave principal de $R$.

La comprobación para 2FN implica la verificación de las dependencias funcionales cuyos atributos del lado izquierdo forman parte de la clave principal. Si ésta contiene un único atributo, no es necesario aplicar la verificación. La relación EMP_PROY de la Figura 10.3(b) está en 1FN pero no en 2FN. El atributo no primo NombreE viola la 2FN debido a DF2, lo mismo que hacen los atributos no primos NombreProyecto y UbicaciónProyecto debido a DF3. Las dependencias funcionales DF2 y DF3 hacen que NombreE, NombreProyecto y UbicaciónProyecto sean parcialmente dependientes de la clave principal {Dni, NumProyecto} de EMP_PROY, violando la comprobación de 2FN.

Si un esquema de relación está en 2FN, puede ser *normalizado en segundo lugar* o *normalizado 2FN* en un número de relaciones 2FN en las que los atributos no primos sólo están asociados con la parte de la clave principal de la que son completa y funcionalmente dependientes. Por consiguiente, las dependencias funcionales FD1, FD2 y FD3 de la Figura 10.3(b) inducen a la descomposición de EMP_PROY en los tres esquemas de relación EP1, EP2 y EP3 mostrados en la Figura 10.10(a), cada uno de los cuales está es 2FN.

### 10.3.6 Tercera forma normal

La **tercera forma normal** (**3FN**) se basa en el concepto de *dependencia transitiva*. Una dependencia funcional $X \rightarrow Y$ en un esquema de relación $R$ es una **dependencia transitiva** si existe un conjunto de atributos $Z$ que ni es clave candidata ni un subconjunto de ninguna clave de $R$,[15] y se cumple tanto $X \rightarrow Z$ como $Z \rightarrow Y$. La dependencia Dni → DniDirector es transitiva a través de NúmeroDpto en EMP_DEPT en la Figura 10.3(a) porque se cumplen las dependencias Dni → NúmeroDpto y NúmeroDpto → DniDirector *y* NúmeroDpto no es una clave por sí misma ni un subconjunto de la clave de EMP_DEPT. Intuitivamente, podemos ver que la dependencia de DniDirector en NúmeroDpto no es deseable en EMP_DEPT ya que NúmeroDpto no es una clave de EMP_DEPT.

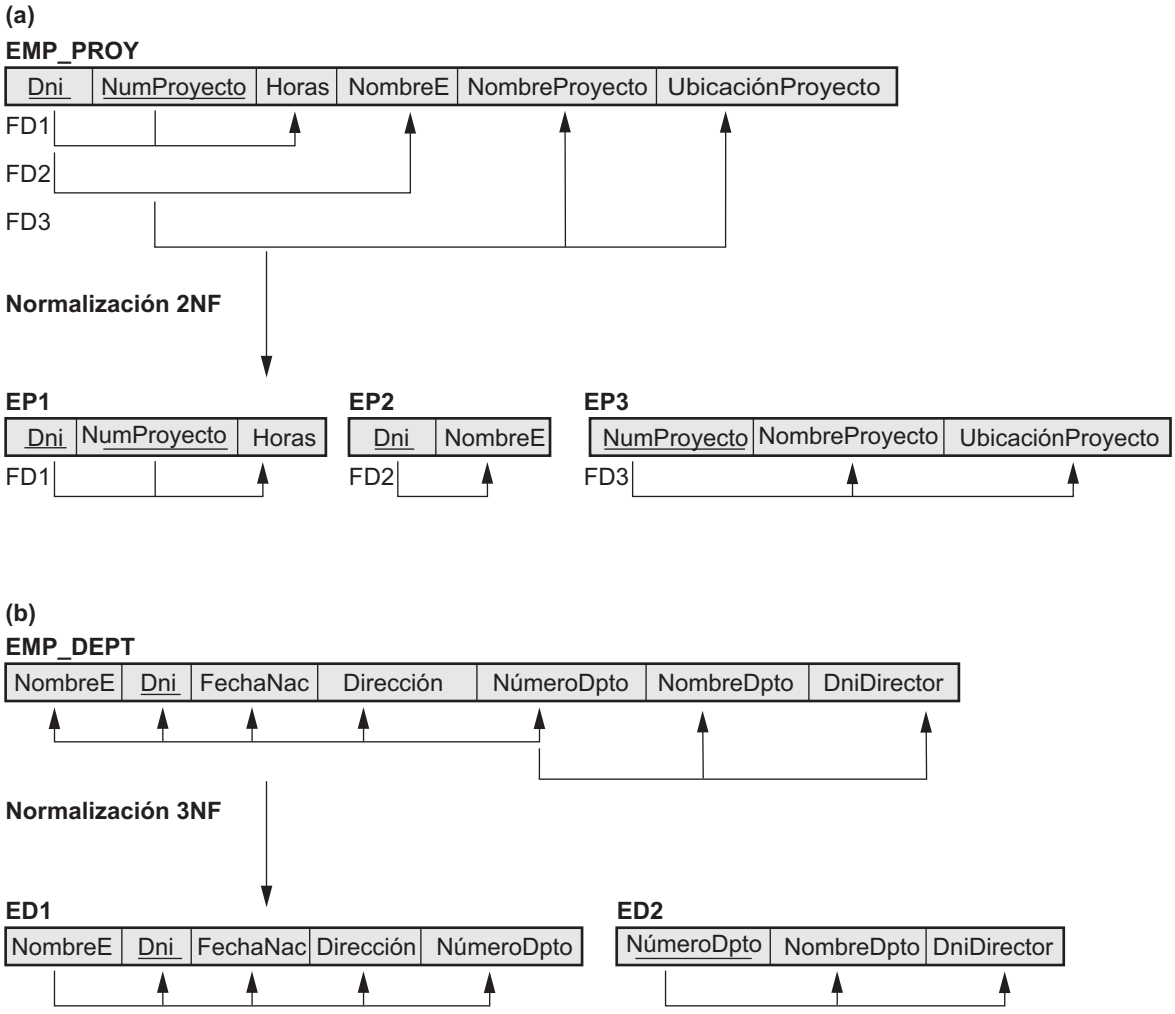
> **Definición.** Según la definición original de Codd, un esquema de relación $R$ está en **3FN** si satisface 2FN *y* ningún atributo no primo de $R$ es transitivamente dependiente en la clave principal.

El esquema de relación EMP_DEPT de la Figura 10.3(a) está en 2FN, ya que no existen dependencias no parciales en una clave. Sin embargo, EMP_DEPT no está en 3FN debido a la dependencia transitiva de DniDirector (y también de NombreDpto) en Dni a través de NúmeroDpto. Podemos normalizar EMP_DEPT descomponiéndola en los dos esquemas de relación 3FN ED1 y ED2 mostrados en la Figura 10.10(b). Intuitivamente, vemos que ED1 y ED2 representan entidades independientes de empleados y departamentos. Una operación CONCATENACIÓN NATURAL en ED1 y ED2 recuperará la relación EMP_DEPT original sin generar tuplas falsas.

Intuitivamente podemos ver que cualquier dependencia funcional en la que el lado izquierdo es parte de la clave principal (subconjunto propio), o cualquier dependencia funcional en la que el lado izquierdo es un

[15] Ésta es la definición general de una dependencia transitiva. Ya que en esta sección sólo nos interesan las claves principales, permitimos las dependencias transitivas donde $X$ es la clave principal pero $Z$ podría ser (un subconjunto de) una clave candidata.

**Figura 10.10.** Normalización en 2FN y 3FN. (a) Normalizando EMP_PROY en relaciones 2FN. (b) Normalizando EMP_DEPT en relaciones 3FN.




atributo no clave, implica una DF *problemática*. La normalización 2FN y 3FN elimina estas DFs descomponiendo la relación original en nuevas relaciones. En términos del proceso de normalización, no es necesario quitar las dependencias parciales antes que las dependencias transitivas, pero históricamente, la 3FN se ha definido con la presunción de que una relación se ha verificado antes para 2FN que para 3FN. La Tabla 10.1 resume de manera informal las tres formas normales basándose en sus claves principales, las pruebas empleadas en cada caso y el *remedio* correspondiente, o la normalización realizada para conseguir la forma normal.

## 10.4 Definiciones generales de la segunda y tercera formas normales

En general, diseñaremos nuestro esquema de relaciones de forma que no tengan ni dependencias parciales ni transitivas porque causan las anomalías de actualización comentadas en la Sección 10.1.2. Los pasos para la normalización en relaciones 3FN que hemos visto hasta ahora imposibilitan las dependencias parciales y

**Tabla 10.1.** Resumen de las formas normales en función a sus claves principales y la normalización correspondiente.

| Forma normal | Prueba | Remedio (normalización) |
|---|---|---|
| Primera (1FN) | La relación no debe tener atributos multivalor o relaciones anidadas. | Generar nuevas relaciones para cada atributo multivalor o relación anidada. |
| Segunda (2FN) | Para relaciones en las que la clave principal contiene varios atributos, un atributo no clave debe ser funcionalmente dependiente en una parte de la clave principal. | Descomponer y configurar una nueva relación por cada clave parcial con su(s) atributo(s) dependiente(s). Asegurarse de mantener una relación con la clave principal original y cualquier atributo que sea completa y funcionalmente dependiente de ella. |
| Tercera (3FN) | La relación no debe tener un atributo no clave que esté funcionalmente determinado por otro atributo no clave (o por un conjunto de atributos no clave). Esto es, debe ser una dependencia transitiva de un atributo no clave de la clave principal. | Descomponer y configurar una relación que incluya el(los) atributo(s) no clave que determine(n) funcionalmente otro(s) atributo(s) no clave. |

transitivas en la *clave principal.* Estas definiciones, sin embargo, no tienen en cuenta otras claves candidatas de una relación, en caso de que existan. En esta sección vamos a ver unas definiciones más generales de las formas 2FN y 3FN que sí consideran *todas* esas claves candidatas. Observe que esto no afecta a la definición de la 1FN ya que es independiente de claves y dependencias funcionales. Un **atributo primo** es, de forma general, un atributo que forma parte de *cualquier clave candidata*. Ahora consideraremos las dependencias funcionales y transitivas *respecto a todas las claves candidatas* de una relación.

## 10.4.1 Definición general de la segunda forma normal

> **Definición.** Un esquema de relación *R* está en **segunda forma normal** (**2FN**) si cada atributo no primo *A* en *R* no es parcialmente dependiente de *ninguna* clave de *R*.[16]

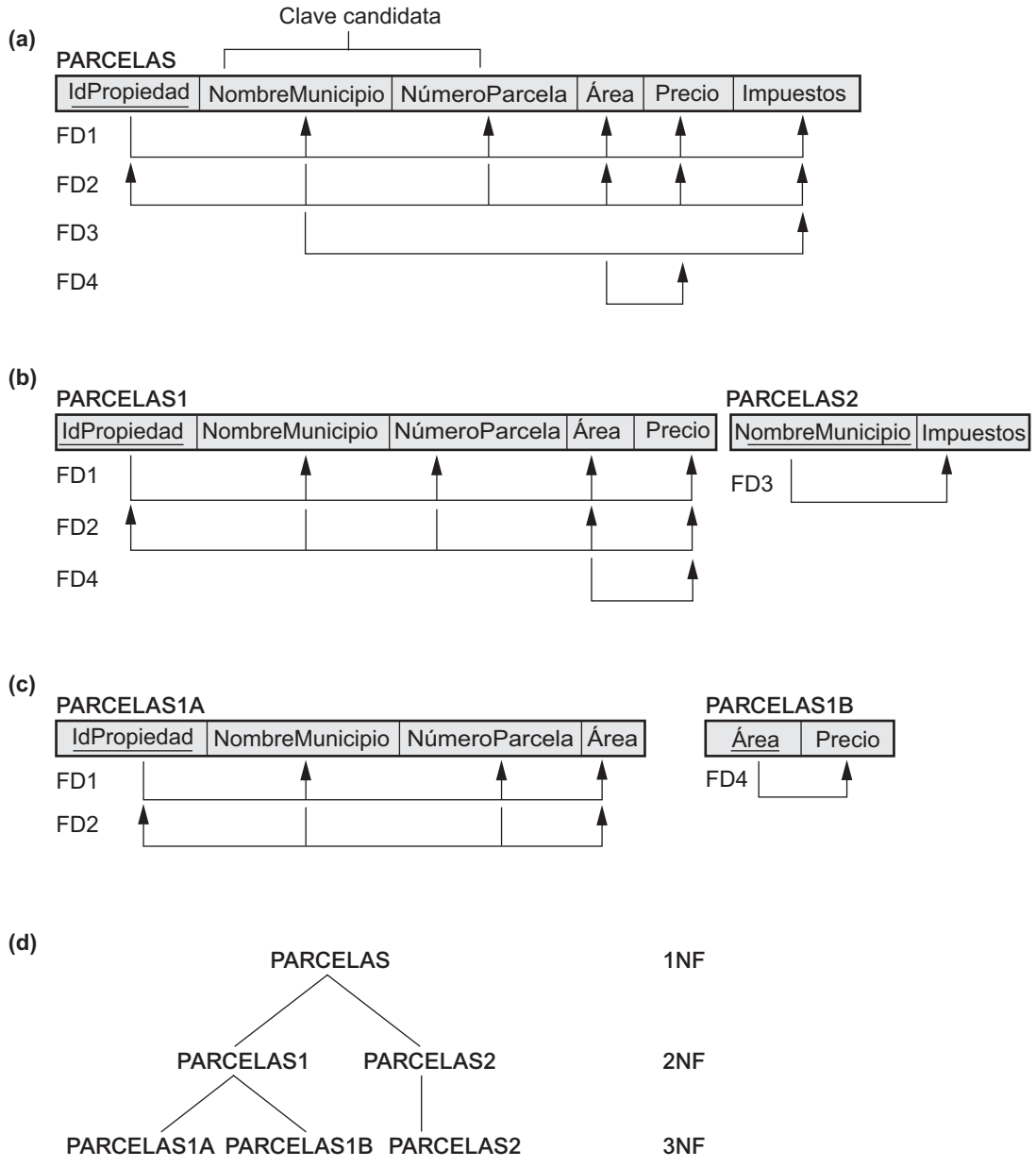
La prueba para 2FN implica la verificación de las dependencias funcionales cuyos atributos de la izquierda forman *parte de* la clave principal. Si la clave principal contiene un único atributo, no es preciso aplicar la comprobación a todo. Considere el esquema de relación PARCELAS de la Figura 10.11(a), que describe los terrenos en venta en distintas provincias. Suponga que existen dos claves candidatas: IdPropiedad y {NombreMunicipio, NúmeroParcela}, es decir, los números de parcela son únicos en cada provincia, mientras que los IdPropiedad son únicos para toda una provincia.

En base a las dos claves candidatas IdPropiedad y {NombreMunicipio, NúmeroParcela}, sabemos que se cumplen las dependencias funcionales FD1 y FD2 de la Figura 10.11(a). Elegimos IdPropiedad como clave principal, razón por la que aparece subrayada en la Figura 10.11(a), aunque no es preciso tomar ninguna consideración especial con esta clave en relación con la otra clave candidata. Supongamos que también se cumplen las dos siguientes dependencias funcionales en PARCELAS:

FD3: NombreProvincia → Impuestos
FD4: Área → Precio

[16] Esta definición puede redefinirse de la siguiente forma: un esquema de relación *R* está en 2FN si cada atributo no primo *A* en *R* es completa y funcionalmente dependiente en *cada* clave de *R*.

**Figura 10.11.** Normalización en 2FN y 3FN. (a) La relación PARCELAS con sus dependencias funcionales de la FD1 a la FD4. (b) Descomposición en la relaciones 2FN PARCELAS1 y PARCELAS2. (c) Descomposición de PARCELAS1 en las relaciones 3FN PARCELAS1A y PARCELAS1B. (d) Resumen de la normalización progresiva de PARCELAS.




La dependencia FD3 dice que los impuestos son fijos para cada municipio (no varían de una parcela a otra en el mismo municipio), mientras que FD4 dice que el precio de una parcela lo determina su área sin tener en cuenta el municipio en el que se encuentre (asumimos que éste es el precio de la parcela por motivos fiscales).

El esquema de relación PARCELAS viola la definición general de 2FN porque Impuestos es parcialmente dependiente de la clave candidata {NombreMunicipio, NúmeroParcela}, debido a FD3. Para normalizar

PARCELAS a 2FN, la descomponemos en las dos relaciones PARCELAS1 y PARCELAS2, mostradas en la Figura 10.11(b). Construimos PARCELAS1 eliminando de PARCELAS el atributo Impuestos que viola 2FN y colocándolo junto a NombreProvincia (el lado izquierdo de FD3 que provoca la dependencia parcial) en otra relación PARCELAS2. Tanto PARCELAS1 como PARCELAS2 están en 2FN. Observe que FD4 no viola 2FN y persiste en PARCELAS1.

### 10.4.2 Definición general de la tercera forma normal

> **Definición.** Un esquema de relación $R$ está en **tercera forma normal** (**3FN**) si, siempre que una dependencia funcional *no trivial* $X \rightarrow A$ se cumple en $R$, ya sea (a) $X$ una superclave de $R$, o (b) $A$ un atributo primo de $R$.

Según esta definición, PARCELAS2 (Figura 10.11(b)) está en 3FN. Sin embargo, FD4 de PARCELAS1 viola 3FN porque Área no es una superclave y Precio no es un atributo primo en PARCELAS1. Para normalizar PARCELAS1 a 3FN, la descomponemos en los esquemas de relación PARCELAS1A y PARCELAS1B de la Figura 10.11(c). Construimos PARCELAS1A eliminando el atributo Precio de PARCELAS1, que viola 3FN, y colocándolo junto con Área (el lado izquierdo de FD4 que provoca la dependencia transitiva) en otra relación PARCELAS1B. Tanto PARCELAS1A como PARCELAS1B están en 3FN.

Dos son los puntos a destacar en este ejemplo y en la definición general de la 3FN:

- PARCELAS1 viola 3FN porque Precio depende transitivamente de cada una de las claves candidatas de PARCELAS1 a través del atributo no primo Área.
- Esta definición general puede aplicarse *directamente* para comprobar si un esquema de relación está en 3FN; esto no implica pasar primero por 2FN. Si aplicamos la definición anterior de 3FN a PARCELAS con las dependencias FD1 a FD4, encontramos que *tanto* FD3 como FD4 violan 3FN. Por consiguiente, podríamos descomponer PARCELAS en PARCELAS1A, PARCELAS1B y PARCELAS2 directamente. Por consiguiente, las dependencias transitiva y parcial que violan 3FN pueden eliminarse *en cualquier orden*.

### 10.4.3 Interpretando la definición general de la tercera forma normal

Un esquema de relación $R$ viola la definición general de 3FN si una dependencia funcional $X \rightarrow A$ se cumple en $R$ y viola las *dos* condiciones (a) y (b) de la 3FN. La violación del apartado (b) significa que $A$ es un atributo no primo, mientras que la vulneración del (a) implica que $X$ no es una superclave de ninguna clave de $R$; por consiguiente, $X$ podría ser no primo o ser un subconjunto propio de una clave de $R$. Si $X$ es no primo, lo que tenemos es una dependencia transitiva que viola 3FN, mientras que si $X$ es un subconjunto propio de una clave de $R$, lo que aparece es una dependencia parcial que viola 3FN (y también 2FN). Por tanto, podemos declarar una **definición alternativa general de 3FN** de la siguiente manera:

> **Definición alternativa.** Un esquema de relación $R$ está en 3FN si cada atributo no primo de $R$ cumple las siguientes condiciones:
>
> - Es completa y funcionalmente dependiente de cada clave de $R$.
> - No depende transitivamente de cada clave de $R$.

## 10.5 Forma normal de Boyce-Codd

La **BCNF (Forma normal de Boyce-Codd, *Boyce-Codd Normal Form*)** se propuso como una forma más simple de la 3FN, aunque es más estricta que ésta. Es decir, toda relación que esté en BCNF lo está también

en 3FN; sin embargo, una relación 3FN *no está necesariamente* en BCNF. Intuitivamente, podemos ver la necesidad de una forma normal más estricta que la 3FN si volvemos al esquema de relación PARCELAS de la Figura 10.11(a), la cual tiene cuatro dependencias funcionales: de la FD1 a la FD4. Supongamos que tenemos cientos de parcelas en la relación, pero que sólo están en dos municipios: Getafe y Alcorcón. Supongamos también que el tamaño de las parcelas de Alcorcón es de sólo 0,5; 0,6; 0,7; 0,8; 0,9 y 1 hectárea, mientras que las de Getafe están restringidas a 1,1 y 2 hectáreas. En una situación como ésta deberemos contar con una dependencia funcional adicional FD5: Área → NombreMunicipio. Si queremos añadirla al resto de dependencias, el esquema de relación PARCELAS1A seguirá estando en 3FN porque NombreMunicipio es un atributo primo.

El área de una parcela que determina el municipio, según se especifica en la FD5, puede representarse mediante 16 tuplas en una relación separada *R*(Área, NombreMunicipio), ya que sólo existen 16 posibles valores de Área. Esta representación reduce la redundancia de tener que repetir la misma información en los miles de tuplas PARCELAS1A. La BCNF es una *forma normal más estricta* que prohibiría PARCELAS1A y sugeriría la necesidad de descomponerla.

> **Definición**. Un esquema de relación *R* está en **BCNF** si siempre que una dependencia funcional *no trivial* $X \rightarrow A$ se cumple en *R*, entonces *X* es una superclave de *R*.

La definición formal de BCNF difiere ligeramente de la de 3FN. La única diferencia entre ellas es la condición (b) de 3FN, la cual permite que *A* sea primo, lo que no se consiente en BCNF. En nuestro ejemplo, FD5 viola BCNF en PARCELAS1A porque ÁREA no es una superclave de PARCELAS1A. Observe que FD5 satisface la 3FN en PARCELAS1A porque NombreMunicipio es un atributo primo (condición b), pero esta condición no existe en la definición de BCNF. Podemos descomponer PARCELAS1A en las dos relaciones BCNF PARCELAS1AX y PARCELAS1AY mostradas en la Figura 10.12(a). Esta descomposición pierde la dependencia funcional FD2 porque sus atributos no coexisten en la misma relación tras la descomposición.

En la práctica, casi todos los esquemas de relación que están en 3FN lo están también en BCNF. Sólo si se cumple $X \rightarrow A$ en un esquema de relación *R,* no siendo *X* una superclave *y* siendo *A* un atributo primo,

**Figura 10.12.** Forma normal de Boyce-Codd. (a) Normalización BCNF de PARCELAS1A en la que se pierde la dependencia funcional FD2 en la descomposición. (b) Una relación esquemática con FDs; está en 3FN pero no en BCNF.

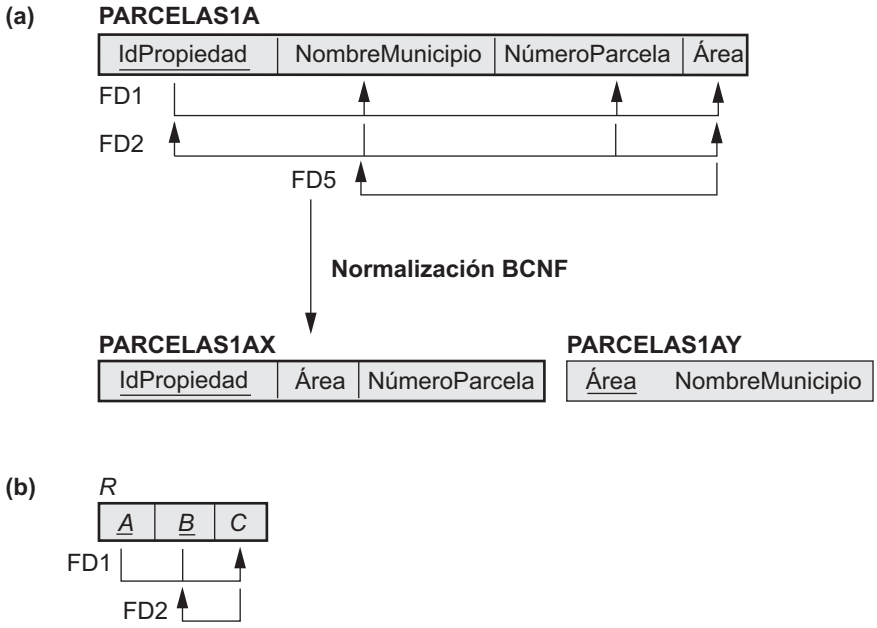


**Figura 10.13.** Una relación ENSEÑAR que está en 3FN pero no en BCNF.

| Estudiante | Curso | Profesor |
|---|---|---|
| Campos | Bases de datos | Marcos |
| Pérez | Bases de datos | María |
| Pérez | Sistemas operativos | Amanda |
| Pérez | Teoría | Sergio |
| Ochoa | Bases de datos | Marcos |
| Ochoa | Sistemas operativos | Aurora |
| Morera | Bases de datos | Eduardo |
| Celaya | Bases de datos | María |
| Campos | Sistemas operativos | Amanda |

estará en 3FN pero no en BCNF. El esquema de relación *R* mostrado en la Figura 10.12(b) ilustra el caso general de una relación como ésta. De forma ideal, el diseño de una base de datos relacional debe afanarse por cumplir la BCNF o la 3FN en cada esquema de relación. Alcanzar sólo la normalización 1FN o 2FN no se considera adecuado, ya que fueron desarrolladas históricamente como una pasarela hacia la 3FN y la BCNF.

La Figura 10.13 es otro ejemplo que muestra una relación ENSEÑAR con las siguientes dependencias:

FD1: {Estudiante, Curso} → Profesor
FD2:[17] Profesor → Curso

Observe que {Estudiante, Curso} es una clave candidata para esta relación y que las dependencias mostradas siguen el patrón de la Figura 10.12(b), con Estudiante como *A*, Curso como *B* y Profesor como *C*. Por consiguiente, esta relación está en 3FN pero no en BCNF. La descomposición de este esquema de relación en dos esquemas no es muy correcta porque puede dividirse en uno de los tres siguientes pares:

1. {Estudiante, Profesor} y {Estudiante, Curso}.
2. {Curso, Profesor} y {Curso, Estudiante}.
3. {Profesor, Curso} y {Profesor, Estudiante}.

Las tres descomposiciones *pierden* la dependencia funcional FD1. De ellas, la *descomposición deseable* es la número 3 porque no generará tuplas falsas tras una concatenación.

Una prueba para determinar si una descomposición es no aditiva (sin pérdida) se explica en la Sección 11.1.4 bajo la Propiedad NJB. En general, una relación que no está en BCNF debería descomponerse para cumplir esta propiedad, a la vez que se renuncia a preservar todas las dependencias funcionales en las relaciones resultantes, como ocurre en este ejemplo. El Algoritmo 11.3 hace esto y debería emplearse antes que la descomposición 3 de ENSEÑAR, la cual produce dos relaciones en BCNF como éstas:

(Profesor, Curso) y (Profesor, Estudiante)

Observe que si designamos (Profesor, Estudiante) como clave principal de la relación ENSEÑAR, la DF Profesor → Curso provoca una dependencia parcial (no completamente funcional) de Curso en una parte de esta clave. Esta DF podría eliminarse como parte de una segunda normalización generando exactamente las

[17] Esta dependencia significa que *cada profesor que enseña un curso* es una restricción para esta aplicación.

dos mismas relaciones. Esto es un ejemplo de cómo alcanzar el mismo diseño BCNF a través de rutas de normalización alternativas.

# 10.6 Resumen

En este Capítulo hemos tratado varios de los peligros a los que nos podemos enfrentar a la hora de diseñar bases de datos relacionales usando argumentos intuitivos. Identificamos de manera informal algunas de las medidas para indicar si un esquema de relación es *idóneo* o *inadecuado*, y facilitamos algunas líneas maestras informales para un buen diseño. Estas directrices están basadas en la realización de un cuidadoso diseño conceptual en el modelo ER y EER, siguiendo correctamente el procedimiento de asignación del Capítulo 7 para asociar entidades y relaciones. Una ejecución correcta de estas directrices, y la ausencia de redundancia, evitará las anomalías en la inserción/borrado/actualización, y la generación de datos falsos. Recomendamos limitar los valores NULL que causan problemas durante las operaciones SELECCIÓN, CONCATENACIÓN y de agregación. A continuación presentamos algunos conceptos formales que nos permiten realizar diseños relacionales de forma descendente mediante el análisis individual de la relaciones. Definimos este proceso de diseño mediante el análisis y la descomposición introduciendo el proceso de normalización.

Definimos el concepto de dependencia funcional y comentamos algunas de sus propiedades. Las dependencias funcionales especifican restricciones semánticas entre los atributos de un esquema de relación. Mostramos cómo se pueden inferir dependencias adicionales de un conjunto de dependencias funcionales usando reglas de inferencia. Definimos los conceptos de clausura y cubierta en relación con las dependencias funcionales. Después mostramos qué es la cubierta mínima de un conjunto de dependencias y ofrecimos un algoritmo para calcularla. También enseñamos el modo de verificar si dos conjuntos de dependencias funcionales son equivalentes.

A continuación describimos el proceso de normalización para lograr buenos diseños mediante la verificación de las relaciones. Ofrecimos un tratamiento de normalización sucesiva basado en una clave principal predefinida en cada relación, y después "aflojamos" este requisito y mostramos una serie de definiciones más generales de la segunda (2FN) y tercera (3FN) formas normales que tienen en cuenta todas las claves candidatas de una relación. Presentamos ejemplos que ilustran cómo, utilizando la definición general de la 3FN, una relación concreta puede analizarse y descomponerse para, eventualmente, producir un conjunto de relaciones en 3FN.

Finalmente, presentamos la BCNF (Forma normal de Boyce-Codd, *Boyce-Codd Normal Form*) y explicamos que se trata de una forma más estricta que la 3FN. Ilustramos también cómo la descomposición de una relación que no está en BCNF debe realizarse considerando los requisitos de descomposición no aditiva. El Capítulo 11 presenta los algoritmos de síntesis y descomposición para un diseño de bases de datos relacional basado en dependencias funcionales. En relación con la descomposición, abordamos los conceptos de reunión no aditiva (sin pérdida) y *conservación de las dependencias*, los cuales están implementados por alguno de estos algoritmos. Otros temas incluidos en el Capítulo 11 son las dependencias multivalor, las de concatenación y la cuarta y quinta formas normales, las cuales consideran estas dependencias.

## Preguntas de repaso

**10.1.** Argumente la semántica de atributo como una medida informal de la idoneidad de un esquema de relación.

**10.2.** Comente las anomalías de inserción, borrado y modificación. ¿Por qué están consideradas como malas? Ilustre sus comentarios con ejemplos.

**10.3.** ¿Por qué deben evitarse en la medida de lo posible lo valores NULL en una relación? Comente el problema de las tuplas falsas y cómo pueden prevenirse.

**10.4.** Enuncie las directrices informales para un esquema de relación que hemos comentado. Ilustre cómo la violación de estas líneas maestras podría ser dañina.

**10.5.** ¿Qué es una dependencia funcional? ¿Cuáles son las posibles fuentes de información que definen las dependencias funcionales que se cumplen entre los atributos de un esquema de relación?

**10.6.** ¿Por qué no podemos inferir automáticamente una dependencia funcional de un estado de relación particular?

**10.7.** ¿Cuál es el papel de las reglas de inferencia de Armstrong (las que van de la RI1 a la RI3) en el desarrollo de la teoría del diseño relacional?

**10.8.** ¿Qué implican la integridad y la solidez de las reglas de inferencia de Armstrong?

**10.9.** ¿Cuál es el significado de la clausura de un conjunto de dependencias funcionales? Ilústrelo con un ejemplo.

**10.10.** ¿Cuándo son equivalentes dos dependencias funcionales? ¿Cómo podemos determinar esta equivalencia?

**10.11.** ¿Qué es un conjunto mínimo de dependencias funcionales? ¿Debe tener cada conjunto de dependencias un conjunto equivalente mínimo? ¿Es siempre único?

**10.12.** ¿A qué hace referencia el término *relación no normalizada*? ¿Cómo se desarrollaron históricamente las formas normales desde la primera hasta la de Boyce-Codd?

**10.13.** Defina las tres primeras formas normales cuando sólo se consideran las claves principales. ¿En qué difieren las definiciones generales de la 2FN y la 3FN, las cuales consideran todas las claves de una relación, de las que sólo consideran las claves principales?

**10.14.** ¿Qué dependencias no deseables se evitan cuando una relación está en 2FN?

**10.15.** ¿Qué dependencias no deseables se evitan cuando una relación está en 3FN?

**10.16.** Defina la forma normal de Boyce-Codd. ¿En qué difiere de la 3FN? ¿Por qué se la considera una forma más estricta de la 3FN?

## Ejercicios

**10.17.** Suponga que tenemos los siguientes requisitos para la base de datos de una universidad que se utiliza para controlar los certificados de estudios de los estudiantes:

- **a.** La universidad controla, por cada estudiante, su NombreEstudiante, su NúmeroEstudiante, su Dni, su DirecciónActualEstudiante y su TeléfonoActualEstudiante, su DirecciónPermanenteEstudiante y su TeléfonoPermanenteEstudiante, su FechaNac, su Sexo, su Curso (‘primer año’. ‘segundo año’, . . . , ‘graduado’) y su Especialidad. Tanto el Dni como el NúmeroEstudiante tienen valores únicos para cada estudiante.
- **b.** Cada departamento está descrito mediante un NombreDpto, un CódigoDpto, un NúmeroOficina, un TeléfonoOficina y un Colegio. Tanto el nombre como el código tienen valores únicos para cada departamento.
- **c.** Cada curso tienen un NombreCurso, una DescripciónCurso, un NúmeroCurso, el NúmeroHorasSemestre, el Nivel y el DepartamentoImparte. El número de curso es único por cada uno de ellos.
- **d.** Cada Sección tiene un profesor (NombreProfesor), un Semestre, un Año, un CursoSección y un NumSección. El número de sección diferencia cada una de las secciones del mismo curso que se imparten durante el mismo semestre/año; sus valores son 1, 2, 3, ... , hasta alcanzar el número de secciones impartidas durante cada semestre.
- **e.** Un registro de nota hace referencia a un Estudiante (Dni), una sección particular y una Nota.

Diseñar un esquema de base de datos relacional para esta aplicación. Primero, muestre todas las dependencias funcionales que deben cumplirse entre los atributos. A continuación, diseñe el esquema de relaciones para la base de datos que están en 3FN o BCNF. Especifique los atributos clave

de cada relación. Anote cualquier requisito no especificado y tome las decisiones necesarias para suministrar la especificación completa.

**10.18.** Confirme o rechace las siguientes reglas de inferencia para dependencias funcionales. Puede realizarse una comprobación mediante un argumento de prueba o usando las reglas de inferencia de la RI1 a la RI3. Los rechazos deben llevarse a cabo probando una instancia de relación que satisfaga las condiciones y dependencias funcionales en el lado izquierdo de la regla de inferencia pero que no satisfaga las dependencias del lado derecho.

**a.** $\{W \rightarrow Y, X \rightarrow Z\} \models \{WX \rightarrow Y\}$

**b.** $\{X \rightarrow Y\}$ y $Y \supseteq Z \models \{X \rightarrow Z\}$

**c.** $\{X \rightarrow Y, X \rightarrow W, WY \rightarrow Z\} \models \{X \rightarrow Z\}$

**d.** $\{XY \rightarrow Z, Y \rightarrow W\} \models \{XW \rightarrow Z\}$

**e.** $\{X \rightarrow Z, Y \rightarrow Z\} \models \{X \rightarrow Y\}$

**f.** $\{X \rightarrow Y, XY \rightarrow Z\} \models \{X \rightarrow Z\}$

**g.** $\{X \rightarrow Y, Z \rightarrow W\} \models \{XZ \rightarrow YW\}$

**h.** $\{XY \rightarrow Z, Z \rightarrow X\} \models \{Z \rightarrow Y\}$

**i.** $\{X \rightarrow Y, Y \rightarrow Z\} \models \{X \rightarrow YZ\}$

**j.** $\{XY \rightarrow Z, Z \rightarrow W\} \models \{X \rightarrow W\}$

**10.19.** Considere los dos siguientes conjuntos de dependencias funcionales: $F = \{A \rightarrow C, AC \rightarrow D, E \rightarrow AD, E \rightarrow H\}$ y $G = \{A \rightarrow CD, E \rightarrow AH\}$. Compruebe si son equivalentes.

**10.20.** Considere el esquema de relación EMP_DEPT de la Figura 10.3(a) y el siguiente conjunto $G$ de dependencias funcionales en EMP_DEPT: $G$ = {Dni → {NombreE, FechaNac, Dirección, NúmeroDpto}, NúmeroDpto → {NombreDpto, DniDirector} }. Calcule las clausuras $\{\text{Dni}\}^+$ y $\{\text{NúmeroDpto}\}^+$ con respecto a $G$.

**10.21.** ¿Es mínimo el conjunto de dependencias funcionales $G$ del Ejercicio 10.20? En caso negativo, intente localizar uno que sea equivalente a $G$ y demuestre que ambos lo son.

**10.22.** ¿Qué anomalías de actualización ocurren en las relaciones EMP_PROY y EMP_DEPT de las Figuras 10.3 y 10.4?

**10.23.** ¿En qué forma normal está el esquema de relación PARCELAS de la Figura 10.11(a) con respecto a las interpretaciones que *sólo tienen en cuenta la clave principal*? ¿Estaría en la misma forma normal si se utilizaran sus definiciones generales?

**10.24.** Demuestre que cualquier esquema de relación con dos atributos está en BCNF.

**10.25.** ¿Por qué pueden producirse dos tuplas falsas en el resultado de la concatenación de las relaciones EMP_PROY1 y EMP_ LOCS de la Figura 10.5 (el resultado se muestra en la Figura 10.6)?

**10.26.** Considere la relación universal $R = \{A, B, C, D, E, F, G, H, I, J\}$ y el conjunto de dependencias funcionales $F = \{ \{A, B\} \rightarrow \{C\}, \{A\} \rightarrow \{D, E\}, \{B\} \rightarrow \{F\}, \{F\} \rightarrow \{G,H\}, \{D\} \rightarrow \{I, J\} \}$. ¿Cuál es la clave para $R$? Descomponga $R$ en relaciones 2FN y después en 3FN.

**10.27.** Repita el Ejercicio 10.26 para el siguiente conjunto diferente de dependencias funcionales $G = \{ \{A, B\} \rightarrow \{C\}, \{B, D\} \rightarrow \{E, F\}, \{A, D\} \rightarrow \{G, H\}, \{A\} \rightarrow \{I\}, \{H\} \rightarrow \{J\} \}$.

**10.28.** Considere la relación dada en la tabla de la página siguiente:

**a.** Dada la extensión previa (estado), ¿cuáles de las siguientes dependencias *podrían cumplirse* en la relación anterior? Si la dependencia no se cumple, explique por qué *especificando las tuplas que provocan la violación*.

i. A → B, ii. B → C, iii. C → B, iv. B → A, v. C → A

| A | B | C | TUPLA# |
|---|---|---|---|
| 10 | b1 | c1 | #1 |
| 10 | b2 | c2 | #2 |
| 11 | b4 | c1 | #3 |
| 12 | b3 | c4 | #4 |
| 13 | b1 | c1 | #5 |
| 14 | b3 | c4 | #6 |

**b.** ¿Tiene la relación anterior alguna clave candidata? En caso afirmativo, ¿cuál es? En caso negativo, ¿por qué no la tiene?

**10.29.** Considere una relación $R(A, B, C, D, E)$ con las siguientes dependencias:

$AB \rightarrow C$, $CD \rightarrow E$, $DE \rightarrow B$

¿Es $AB$ una clave candidata de esta relación? En caso negativo, ¿lo es $ABD$? Razone su respuesta.

**10.30.** Considere la relación $R$, que tiene atributos que guardan programaciones de cursos y secciones en una universidad; $R$ = {NúmeroCurso, NumSección, DeptOfertante, HorasCrédito, NivelCurso, DniProfesor, Semestre, Año, HorasDía, NúmeroSala, NúmeroDeEstudiantes}. Supongamos que en $R$ se mantienen las siguientes dependencias funcionales:

{NúmeroCurso} → {DeptOfertante, HorasCrédito, NivelCurso}
{NúmeroCurso, NumSección, Semestre, Año} → {HorasDía, NúmeroSala, NúmeroDeEstudiantes, DniProfesor}
{NúmeroSala, HorasDía, Semestre, Año} → {DniProfesor, NúmeroCurso, NumSección}

Intente determinar qué conjuntos de atributos forman las claves de $R$. ¿Cómo se podría normalizar esta relación?

**10.31.** Considere las siguientes relaciones para una aplicación de base de datos para el procesamiento de pedidos de la empresa ABC, Inc.

PEDIDO (NúmeroPedido, FechaPedido, NúmeroCliente, CosteTotal)
LÍNEA_PEDIDO(NúmeroPedido, CódigoObjeto, CantidadSolicitada, PrecioTotal, PorcentajeDto)

Asumimos que cada línea de pedido tiene un descuento diferente. El PrecioTotal se refiere a una línea, FechaPedido es la fecha en la que el pedido se realizó y el CosteTotal es el coste total del mismo. Si aplicamos una concatenación natural en la relaciones LÍNEA_PEDIDO y PEDIDO, ¿qué aspecto tendría el esquema de relación resultante? ¿Cuál sería su clave? Muestre las DF resultantes de esta relación. ¿Está en 2FN? ¿Está en 3FN? Razone sus respuestas.

**10.32.** Considere la siguiente relación:

VENTA_COCHE(NúmeroCoche, FechaVenta, NúmeroVendedor, PorcentajeComisión, Descuento)

Asumimos que un coche lo pueden vender varios vendedores, por lo que {NúmeroCoche, NúmeroVendedor} es la clave principal. Las siguientes son otras dependencias adicionales

FechaVenta → Descuento y
NúmeroVendedor → PorcentajeComisión

Basándonos en la clave principal anterior, ¿está la relación en 1FN, 2FN o 3FN? Razone su respuesta. ¿Cómo podría normalizarla completamente?

Considere la siguiente relación para registrar los libros publicados:

LIBRO(TítuloLibro, Autor, TipoLibro, ListaPrecio, AfiliaciónAutor, Editorial)

AfiliaciónAutor hace referencia a la afiliación del autor. Suponga que existen las siguientes dependencias:

TítuloLibro → Editorial, TipoLibro

TipoLibro → ListaPrecio

Autor → AfiliaciónAutor

**a.** ¿En qué forma normal está la relación? Razone su respuesta.

**b.** Aplique una normalización hasta que la relación no pueda descomponerse más. Explique las razones que se esconden tras cada descomposición.

**10.34.** Este ejercicio le pide que convierta sentencias de empresa en dependencias. Considere la relación DISCO (NúmeroSerie, Fabricante, Modelo, Versión, Capacidad, Distribuidor). Cada tupla de esta relación contiene información acerca de un disco con un NúmeroSerie único, fabricado por un fabricante, que tiene un modelo particular, distribuido con un número de versión concreto, que tiene una determinada capacidad y que es vendido por un distribuidor concreto. Por ejemplo, la tupla Disco ('1978619', 'WesternDigital', 'A2235X', '765234', 500, 'CompUSA') especifica que WesternDigital fabricó un disco con el número de serie 1978619, el modelo A2235X y la versión 765234; su capacidad es de 500GB y lo vende CompUSA.

Escriba cada una de las siguientes dependencias como una DF:

**a.** El fabricante y el número de serie identifican el disco de forma única.

**b.** Un modelo está registrado por un fabricante y, por consiguiente, no puede utilizarlo ningún otro.

**c.** Todos los discos de una versión particular son del mismo modelo.

**d.** Todos los discos de un determinado modelo de un fabricante particular tienen exactamente la misma capacidad.

**10.35.** Demuestre que $AB \rightarrow D$ está en la clausura de:

$$\{AB \rightarrow C, CE \rightarrow D, A \rightarrow E\}$$

**10.36.** Considere la relación siguiente:

*R* (NúmeroDoctor, NúmeroPaciente, Fecha, Diagnóstico, CódigoTratamiento, Coste)

En la relación anterior, cada tupla describe la visita de un paciente a un médico junto con el código del tratamiento diagnosticado y un coste diario. Asumimos que el diagnóstico está determinado (únicamente) para cada paciente por un médico. Asumimos también que cada código de tratamiento tiene un coste fijo (independientemente del paciente). ¿Esta relación está en 2FN? Justifique su respuesta y descompóngala si fuera necesario. A continuación, argumente si es necesaria una normalización a 3FN y, en caso afirmativo, realícela.

**10.37.** Considere la siguiente relación:

VENTA_COCHE (IdCoche, Extra, PrecioExtra, FechaVenta, PrecioDescontado)

Esta relación hace referencia a los extras instalados en un coche (por ejemplo, el control de velocidad) y que se vendieron a un distribuidor, y la lista y los precios descontados de los extras.

Si IdCoche → FechaVenta y Extra → PrecioExtra
e IdCoche, Extra → PrecioDescontado,

argumente usando la definición generalizada de la 3FN que esta relación no está en 3FN. A continuación, y desde sus conocimientos de la 2FN, indique por qué tampoco está en 2FN.

**10.38.** Considere una versión descompuesta de la relación:

PRECIO_ACTUAL_EXTRA (IdCoche, Extra, PrecioDescontado)

COCHE(IdCoche, FechaVenta)
EXTRA(Extra, PrecioExtra)

Usando el algoritmo de comprobación de descomposición sin pérdida (Algoritmo 11.1), determine si esta descomposición es realmente sin pérdidas.

## Ejercicios de práctica

*Nota*. Los siguientes ejercicios utilizan el sistema DBD (Diseñador de base de datos, *Data Base Designer*) descrito en el manual de prácticas. El esquema relacional *R* y el conjunto de dependencias funcionales *F* tienen que codificarse como listas. Como ejemplo, *R* y *F* para el problema 10.26 están codificados de este modo:

$R =$ [*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*]
$F =$ [[[*a*, *b*],[*c*]],
[[*a*],[*d*, *e*]],
[[*b*],[*f*]],
[[*f*],[*g*, *h*]],
[[*d*],[*i*, *j*]]]

Ya que DBD está implementado en Prolog, el uso de términos en mayúsculas está reservado a las variables del lenguaje y, por consiguiente, las constantes en minúsculas se emplean para codificar los atributos. Para obtener más detalles acerca del uso del sistema DBD, consulte el manual de prácticas.

**10.39.** Usando el sistema DBD, verifique sus respuestas a los siguientes ejercicios:

**a.** 10.19
**b.** 10.20
**c.** 10.21
**d.** 10.26 (sólo 3FN)
**e.** 10.27
**f.** 10.29
**g.** 10.30

## Bibliografía seleccionada

Las dependencias funcionales fueron presentadas originalmente por Codd (1970). Las definiciones originales de la primera, segunda y tercera formas normales fueron definidas en Codd (1972a), donde puede encontrarse también un comentario acerca de las anomalías de actualización. La forma normal de Boyce-Codd fue definida en Codd (1974). La definición alternativa de la tercera forma normal fue dada en Ullman (1988), así como la definición de BCNF que hemos mostrado aquí. Ullman (1988), Maier (1983) y Atzeni y De Antonellis (1993) contienen muchos de los teoremas y demostraciones referentes a las dependencias funcionales.

Armstrong (1974) muestra la solidez y la integridad de las reglas de inferencia de la RI1 a la RI3. El Capítulo 11 contiene referencias adicionales sobre la teoría del diseño relacional.

CAPÍTULO 11

# Algoritmos de diseño de bases de datos relacionales y dependencias adicionales

El Capítulo 10 presentó una técnica de **diseño relacional descendente** y una serie de conceptos usados extensamente en el diseño de bases de datos comerciales de hoy en día. El procedimiento supone la generación de un esquema conceptual ER o EER y su mapeo posterior al modelo relacional mediante alguno de los procedimientos descritos en el Capítulo 7. Las claves principales se asignan a cada relación en base a dependencias funcionales conocidas. El proceso consecuente podría recibir el nombre de **diseño relacional por análisis**, en el que las relaciones diseñadas inicialmente a partir del procedimiento anterior (o aquéllas heredadas de ficheros, formularios y otras fuentes) se analizan para detectar dependencias funcionales no deseadas. Estas dependencias son eliminadas por el procedimiento de normalización sucesivo descrito en la Sección 10.3 junto con las definiciones de las formas normales relacionadas, las cuales son los mejores estados de diseño de las relaciones individuales. En la Sección 10.3 asumimos que las claves primarias fueron asignadas a relaciones individuales; en la Sección 10.4 se presentó un tratamiento de normalización más general en el que se tenían en cuenta todas las claves candidatas de cada relación.

En este Capítulo usamos la teoría de las formas normales y las dependencias funcionales desarrolladas en el último capítulo mientras se mantienen tres diferentes planteamientos. En primer lugar, se describen las propiedades deseables de las concatenaciones no aditivas (sin pérdida) y la conservación de dependencias funcionales. Se presenta también un algoritmo general para comprobar la no aditividad de las concatenaciones entre un conjunto de relaciones. En segundo lugar, se muestra un acercamiento al **diseño relacional por síntesis** de dependencias funcionales, un **acercamiento de diseño ascendente** que presupone que se ha tomado como entrada el conocimiento de las dependencias funcionales a través de los conjuntos de atributos en el UoD (Universo de discurso, *Universe of Discourse*). Presentamos algoritmos para obtener las formas normales deseables (la 3FN y la BCNF) y para conseguir una, o ambas, propiedades de no aditividad de las concatenaciones y la conservación de la dependencia funcional. Aunque la aproximación por síntesis es teóricamente interesante como un acercamiento formal, no debe usarse en la práctica con diseños de bases de datos de gran tamaño debido a la dificultad de proporcionar todas las posibles dependencias funcionales directas antes de intentar el diseño. Sin embargo, con el desarrollo presentado en el Capítulo 10, las descomposiciones sucesivas y los refinamientos del diseño se hacen más manejables y pueden evolucionar con el tiempo. El último planteamiento de este capítulo es comentar los nuevos tipos de dependencias, como la MVD (Dependencia multivalor, *Multi-Valued Dependency*) y la cuarta forma normal basada en la eliminación de las MVD; por último, se ofrece una introducción breve a la dependencia de concatenación (*Join Dependency*) y a la quinta forma normal relacionada con ella.

En la Sección 11.1, primero se tratan las dos **propiedades de descomposición** deseables, la de conservación de la dependencia y la de concatenación sin pérdida (o no aditiva), que se utilizan en el diseño de algoritmos para obtener descomposiciones correctas. Es importante indicar que *no basta* con comprobar los esquemas de relación *independientemente unos de otros* por cumplir las formas normales superiores como la 2FN, la 3FN y la BCNF. Las relaciones resultantes deben satisfacer colectivamente estas dos propiedades adicionales para calificar un diseño como bueno. La Sección 11.2 está dedicada al desarrollo de algoritmos de diseño relacional que comienzan con un esquema de relación gigante llamado **relación universal**, que es una relación hipotética que contiene todos los atributos. Esta relación se descompone (o, en otras palabras, las dependencias funcionales dadas se sintetizan) en relaciones que satisfacen una determinada forma normal (por ejemplo, la 3FN o la BCNF) y que también cumple alguna de las propiedades deseables.

En la Sección 11.3 trataremos la MVD, y a continuación la JD (Dependencia de concatenación, *Join Dependency*) en la Sección 11.4. Estos dos tipos de dependencias representan restricciones que *no pueden* capturarse por las dependencias funcionales. Su eliminación produce relaciones en cuarta (4FN, carentes de MVDs no deseadas) y quinta formas normales (5FN, carentes de JDs no deseadas).

Completamos la explicación de las dependencias entre datos introduciendo las dependencias de inclusión y las dependencias de plantilla. Las primeras pueden representar restricciones de integridad referencial y de clase/subclase a través de las relaciones. Las segundas son una forma de representar cualquier restricción generalizada en los atributos. Describiremos también algunas situaciones en las que es necesaria la definición de un procedimiento o una función para declarar y verificar una dependencia funcional entre los atributos.

Por último, veremos la DKNF (Forma normal de dominio clave, *Domain-Key Normal Form*), que está considerada como la forma normal más general.

En un curso de introducción a las bases de datos, es posible saltarse alguna, o todas, las Secciones desde la 11.4 a la 11.6.

# 11.1 Propiedades de las descomposiciones relacionales

En la Sección 11.1.1 ofrecemos ejemplos para evidenciar que examinar una relación *individual* para verificar si está en una forma normal superior no es garantía, por sí misma, de un buen diseño; antes bien, un *conjunto de relaciones* que, juntas, forman el esquema de una base de datos relacional debe poseer ciertas propiedades adicionales para garantizar un buen diseño. En las Secciones 11.1.2 y 11.1.3 comentaremos dos de estas propiedades: la conservación (o preservación) de la dependencia y la concatenación no aditiva. La Sección 11.1.4 está dedicada a las descomposiciones binarias, mientras que la Sección 11.1.5 se centra en las de concatenación no aditiva.

## 11.1.1 Descomposición de una relación e insuficiencia de formas normales

Los algoritmos de diseño de una base de datos relacional que se presentan en la Sección 11.2 se inician a partir de un único **esquema de relación universal** $R = \{A_1, A_2, \ldots, A_n\}$ que incluye *todos* los atributos de la base de datos. Implícitamente hacemos la conjetura de **relación universal**, que especifica que cada nombre de atributo es único. El conjunto $F$ de dependencias funcionales que se debe cumplir en los atributos de $R$ está especificado por los diseñadores de la base de datos y disponible a través de los algoritmos de diseño. Al usar dependencias funcionales, los algoritmos descomponen el esquema de relación universal $R$ en un conjunto de esquemas de relación $D = \{R_1, R_2, \ldots, R_m\}$ que se convertirán en el esquema de la base de datos relacional; $D$ recibe el nombre de **descomposición** de $R$.

Debemos asegurarnos de que en la descomposición, cada atributo de $R$ aparezca en, al menos, una relación $R_i$ de forma que no se *pierdan* atributos; formalmente, tenemos:

$$\bigcup_{i=1}^{m} R_i = R$$

Esto es lo que se conoce como condición de **conservación de atributos** de una descomposición.

Otro de los objetivos es conseguir que cada relación $R_i$ individual de la descomposición $D$ esté en BCNF o 3FN. Sin embargo, esta condición no es suficiente por sí misma para garantizar un buen diseño de la base de datos. Debemos considerar la descomposición de la relación universal de una forma general, además de buscar en las relaciones individuales. Para ilustrar este punto, consideremos la relación EMP_LOCS(NombreE, UbicaciónProyecto) de la Figura 10.5, la cual está en 3FN y en BCNF. De hecho, cualquier esquema de relación con sólo dos atributos está automáticamente en BCNF[1]. Aunque EMP_LOCS está en BCNF, sigue generando tuplas falsas cuando se concatena con EMP_PROY (Dni, NumProyecto, Horas, NombreProyecto, UbicaciónProyecto), la cual no está en BCNF (compruebe el resultado de la concatenación natural en la Figura 10.6). Por tanto, EMP_LOCS representa un esquema de relación malo debido a su enrevesada semántica por la que UbicaciónProyecto da la localización de *uno de los proyectos* en los que trabaja un empleado. Al concatenar EMP_LOCS con PROYECTO(NombreProyecto, NumProyecto, UbicaciónProyecto, NumDpto-Proyecto) de la Figura 10.2 (que *está* en BCNF) también se generan tuplas falsas. Esto acentúa la necesidad de encontrar otros criterios que, junto con las condiciones de la 3FN o la BCNF, prevengan estos malos diseños. En las siguientes tres subsecciones comentaremos algunas de estas condiciones adicionales que deben cumplirse en una descomposición $D$.

### 11.1.2 Propiedad de conservación de la dependencia de una descomposición

Resultaría útil si cada dependencia funcional $X \rightarrow Y$ especificada en $F$ apareciera directamente en uno de los esquemas de relación $R_i$ de la descomposición $D$, o pudiera inferirse a partir de las dependencias que aparecen en alguna $R_i$. Informalmente, esto es lo que se conoce como *condición de conservación de la dependencia.* Lo que queremos es mantener las dependencias porque cada una de ellas en $F$ representa una restricción de la base de datos. Si alguna de estas dependencias no está representada en alguna relación individual $R_i$ de la descomposición, no podemos imponer esta restricción en referencia a una relación concreta. Puede que tengamos que concatenar varias relaciones para incluir todos los atributos implicados en esta dependencia.

No es necesario que las dependencias exactas especificadas en $F$ aparezcan en las relaciones individuales de la descomposición $D$. Basta con que la unión de las dependencias que se mantienen en las relaciones individuales de $D$ sea equivalente a $F$. Ahora estamos en condiciones de definir estos conceptos de una manera más formal.

> **Definición.** Dado un conjunto de dependencias $F$ en $R$, la **proyección** de $F$ en $R_i$, especificada por $\pi_{Ri}(F)$ donde $R_i$ es un subconjunto de $R$, es el conjunto de dependencias $X \rightarrow Y$ en $F^+$ tales que los atributos en $X \cup Y$ se encuentran todos en $R_i$. Por consiguiente, la proyección de $F$ en cada esquema de relación $R_i$ en la descomposición $D$ es el conjunto de dependencias funcionales en $F^+$, la clausura de $F$, tal que todos sus atributos del lado izquierdo y derecho están en $R_i$. Decimos que una descomposición $D = \{R_1, R_2, \ldots, R_m\}$ de $R$ **conserva las dependencias** respecto a $F$ si la unión de las proyecciones de $F$ en cada $R_i$ en $D$ es equivalente a $F$; es decir, $((\pi_{R1}(F)) \cup \ldots \cup (\pi_{Rm}(F)))^+ = F^+$.

Si una descomposición no es de dependencia conservada, algunas de las dependencias se **pierden** en la descomposición. Para comprobar que una dependencia perdida se mantiene, debemos tomar la CONCATENACIÓN de dos o más relaciones en la descomposición para obtener una relación que incluya todos los atributos de la izquierda y de la derecha de la dependencia perdida, y después verificar que la dependencia persiste en el resultado de la CONCATENACIÓN (una opción que no es práctica).

---

[1] Como ejercicio, el lector deberá probar que esta sentencia es verdadera.

La Figura 10.12(a) muestra un ejemplo de descomposición que no conserva las dependencias, ya que se pierde la dependencia funcional DF2 cuando se descompone PARCELAS1A en {PARCELAS1AX, PARCELAS1AY}. Sin embargo, las descomposiciones de la Figura 10.11 conservan las dependencias. De forma similar, para el ejemplo de la Figura 10.13, no importa la descomposición que se elija para la relación ENSEÑAR(Estudiante, Curso, Profesor) de las tres que se ofrecen en el texto, se pierden una o ambas dependencias presentadas originalmente. A continuación, formulamos una afirmación que está relacionada con esta propiedad sin proporcionar ninguna prueba.

> **Afirmación 1.** Siempre es posible buscar una descomposición $D$ con las dependencias conservadas respecto a $F$ de modo que cada relación $Ri$ en $D$ esté en 3FN.

En la Sección 11.2.1 se describe el Algoritmo 11.2, que crea una descomposición con las dependencias conservadas $D = \{R_1, R_2, \ldots, R_m\}$ de una relación universal $R$ basada en un conjunto de dependencias funcionales $F$, de modo que cada $R_i$ en $D$ esté en 3FN.

### 11.1.3 Propiedad no aditiva (sin pérdida) de una descomposición

Otra propiedad que puede poseer una descomposición $D$ es la de concatenación no aditiva, la cual garantiza que no se generarán tuplas falsas cuando se aplica una operación de CONCATENACIÓN NATURAL (NATURAL JOIN) a las relaciones de la descomposición. En la Sección 10.1.4 ya ilustramos este problema con el ejemplo de las Figuras 10.5 y 10.6. Ya que ésta es una propiedad de una descomposición de esquemas de relación, la condición de que no existan tuplas falsas debe mantenerse en *cada estado de relación legal,* es decir, en cada relación que satisface las dependencias funcionales en $F$. Por consiguiente, la propiedad de concatenación sin pérdida está siempre definida respecto a un conjunto específico $F$ de dependencias.

> **Definición**. Formalmente, una descomposición $D = \{R_1, R_2, \ldots, R_m\}$ de $R$ tiene **la propiedad de concatenación sin pérdida (no aditiva)** respecto al conjunto de dependencias $F$ en $R$ si, por *cada* estado de relación $r$ de $R$ que satisface $F$, se mantiene lo siguiente, donde $*$ es la CONCATENACIÓN NATURAL de todas las relaciones en $D$: $*(\pi_{R1}(r), \ldots, \pi_{Rm}(r)) = r$.

La palabra "pérdida" en *sin pérdida* hace referencia a una *pérdida de información*, no de tuplas. Si una descomposición no tiene esta propiedad, podríamos obtener tuplas falsas adicionales una vez aplicadas las operaciones PROYECCIÓN ($\pi$) y CONCATENACIÓN NATURAL ($*$); estas tuplas adicionales representan información errónea o incorrecta. Preferimos el término concatenación no aditiva porque describe la situación de una forma más exacta. Aunque el término concatenación sin pérdida ha sido muy popular en la literatura, nosotros usaremos de ahora en adelante el término "concatenación no aditiva" porque es más explicativo y menos ambiguo. Esta propiedad garantiza que no se producirán tuplas falsas tras la aplicación de las operaciones PROYECCIÓN y CONCATENACIÓN. Sin embargo, puede que haya ocasiones en las que empleemos el concepto diseño con pérdida para referirnos a un diseño que representa una pérdida de información (consulte el ejemplo que aparece al final del Algoritmo 11.2).

La descomposición de EMP_PROY(Dni, NumProyecto, Horas, NombreE, NombreProyecto, UbicaciónProyecto) de la Figura 10.3 en EMP_LOCS(NombreE, UbicaciónProyecto) y EMP_PROY1(Dni, NumProyecto, Horas, NombreProyecto, UbicaciónProyecto) de la Figura 10.5 obviamente no cuenta con la propiedad de concatenación no aditiva, tal y como se muestra en la Figura 10.6. Usaremos un procedimiento general para comprobar si cualquier descomposición $D$ de una relación en $n$ relaciones es no aditiva respecto a un conjunto dado de dependencias funcionales $F$ en la relación; es el Algoritmo 11.1 que se muestra más abajo. Es posible aplicar una verificación más simple para determinar si la descomposición es no aditiva para las descomposiciones binarias; esta verificación se describe en la Sección 11.1.4.

> **Algoritmo 11.1.** Verificación de la propiedad de concatenación no aditiva:
>
> **Entrada**: Una relación universal $R$, una descomposición $D = \{R_1, R_2, \ldots, R_m\}$ de $R$ y un conjunto $F$ de dependencias funcionales.

*Nota*: Al final de algunos de los pasos podrá encontrar algunos comentarios explicativos que siguen el formato: (∗ *comentario* ∗)

1. Cree una matriz inicial $S$ con una fila $i$ por cada relación $R_i$ en $D$, y una columna $j$ por cada atributo $A_j$ en $R$.
2. Asigne $S(i, j) := b_{ij}$ en todas las entradas de la matriz. (∗ cada $b_{ij}$ es un símbolo distinto asociado a índices $(i, j)$ ∗)
3. Por cada fila $i$ que representa un esquema de relación $R_i$
   {por cada columna $j$ que representa un atributo $A_j$
   {si la (relación $R_i$ incluye un atributo $A_j$) entonces asignar $S(i, j) := a_j$;};};
   (∗ cada $a_j$ es un símbolo distinto asociado a un índice $(j)$ ∗)
4. Repetir el siguiente bucle hasta que una *ejecución completa del mismo* no genere cambios en $S$
   {por cada dependencia funcional $X \rightarrow Y$ en $F$
   {para todas las filas de $S$ *que tengan los mismos símbolos* en las columnas correspondientes a los atributos de $X$
   {hacer que los símbolos de cada columna que se corresponden con un atributo de $Y$ sean los mismos en todas esas filas siguiendo este patrón: si cualquiera de las filas tiene un símbolo $a$ para la columna, hacer que el resto de filas tengan el *mismo* símbolo $a$ en la columna. Si no existe un símbolo $a$ para el atributo en ninguna de las filas, elegir uno de los símbolos $b$ para el atributo que aparezcan en una de las filas y ajustar el resto de filas a ese valor;} ;} ;};
5. Si una fila está compuesta enteramente por símbolos $a$, entonces la descomposición tiene la propiedad de concatenación no aditiva; en caso contrario, no la tiene.

Dada una relación $R$ que está descompuesta en un número de relaciones $R_1, R_2, \ldots, R_m$, el Algoritmo 11.1 empieza con la matriz $S$ que consideramos que es algún estado de relación $r$ de $R$. La fila $i$ en $S$ representa una tupla $t_i$ (correspondiente a la relación $R_i$) que tiene símbolos $a$ en las columnas que se corresponden con los atributos de $R_i$ y símbolos $b$ en el resto. El algoritmo transforma entonces las filas de esta matriz (en el bucle del paso 4) de modo que representen tuplas que satisfagan todas las dependencias funcionales en $F$. Al final del paso 4, dos filas cualesquiera de S (que representan a dos tuplas de $r$) que coinciden en sus valores de atributos izquierdos $X$ de una dependencia funcional $X \rightarrow Y$ en $F$ coincidirán también en los valores de sus atributos derechos $Y$. Puede verse que tras aplicar el bucle del paso 4, si una fila de $S$ sólo cuenta con símbolos $a$, entonces la descomposición $D$ tiene la propiedad de concatenación no aditiva respecto a $F$.

Si, por otro lado, ninguna fila termina sólo con símbolos $a$, $D$ no satisface la propiedad de concatenación no aditiva. En este caso, la relación $r$ representada por $S$ al final del algoritmo será un ejemplo de estado de relación $r$ de $R$ que satisface las dependencias en $F$ pero que no cumple la condición de concatenación no aditiva. Por consiguiente, esta relación sirve como un **contraejemplo** que prueba que $D$ no tiene esta propiedad respecto a $F$. Observe que los símbolos $a$ y $b$ no tienen ningún significado especial al final del algoritmo.

La Figura 11.1(a) muestra cómo aplicar el Algoritmo 11.1 a la descomposición del esquema de relación EMP_PROY de la Figura 10.3(b) en los dos esquemas de relación EMP_PROY1 y EMP_LOCS de la Figura 10.5(a). El bucle del paso 4 no puede cambiar ninguna de las *bes* por *aes*; por tanto, la matriz $S$ resultante no cuenta con ninguna fila en la que sólo haya símbolos $a$, y por tanto la descomposición no tiene la propiedad de concatenación sin pérdida.

La Figura 11.1(b) es otro ejemplo de descomposición de EMP_PROY (en EMPLEADO, PROYECTO y TRABAJA_EN) que tiene la propiedad de concatenación no aditiva, mientras que la Figura 11.1(c) muestra la forma de aplicar el algoritmo para esta descomposición. Una vez que una fila sólo está compuesta por símbolos $a$, sabemos que la descomposición tiene la propiedad de concatenación no aditiva, y podemos dejar de aplicar las dependencias funcionales (paso 4 del algoritmo) a la matriz $S$.

**Figura 11.1.** Comprobación de la concatenación no aditiva para *n* descomposiciones. (a) Caso 1: la descomposición de EMP_PROY en EMP_PROY1 y EMP_LOCS hace que la comprobación falle. (b) Una descomposición de EMP_PROY que tiene la propiedad de concatenación sin pérdida. (c) Caso 2: la descomposición de EMP_PROY en EMPLEADO, PROYECTO y TRABAJA_EN satisface la comprobación.

**(a)** $R$ = {Dni, NombreE, NumProyecto, NombreProyecto, UbicaciónProyecto, Horas} $D = \{R_1, R_2\}$
$R_1$ = EMP_LOCS = {NombreE, UbicaciónProyecto}
$R_2$ = EMP_PROY1 = {Dni, NumProyecto, Horas, NombreProyecto, UbicaciónProyecto}

$F$ = {Dni ➔ NombreE; NumProyecto ➔ {NombreProyecto, UbicaciónProyecto}; {Dni, NumProyecto} ➔ Horas}

| | Dni | NombreE | NumProyecto | NombreProyecto | UbicaciónProyecto | Horas |
|---|---|---|---|---|---|---|
| $R_1$ | $b_{11}$ | $a_2$ | $b_{13}$ | $b_{14}$ | $a_5$ | $b_{16}$ |
| $R_2$ | $a_1$ | $b_{22}$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ |

(Ningún cambio en la matriz después de aplicar las dependencias funcionales.)

**(b)**

**EMP**

| Dni | NombreE |
|---|---|

**PROYECTO**

| NumProyecto | NombreProyecto | UbicaciónProyecto |
|---|---|---|

**TRABAJA_EN**

| Dni | NumProyecto | Horas |
|---|---|---|

**(c)** $R$ = {Dni, NombreE, NumProyecto, NombreProyecto, UbicaciónProyecto, Horas} $D = \{R_1, R_2, R_3\}$
$R_1$ = EMP = {Dni, NombreE}
$R_2$ = PROY = {NumProyecto, NombreProyecto, UbicaciónProyecto}
$R_3$ = TRABAJA_EN = {Dni, NumProyecto, Horas}

$F$ ={Dni ➔ NombreE; NumProyecto ➔ {NombreProyecto, UbicaciónProyecto};{Dni, NumProyecto} ➔ Horas}

| | Dni | NombreE | NumProyecto | NombreProyecto | UbicaciónProyecto | Horas |
|---|---|---|---|---|---|---|
| $R_1$ | $a_1$ | $a_2$ | $b_{13}$ | $b_{14}$ | $b_{15}$ | $b_{16}$ |
| $R_2$ | $b_{21}$ | $b_{22}$ | $a_3$ | $a_4$ | $a_5$ | $b_{26}$ |
| $R_3$ | $a_1$ | $b_{32}$ | $a_3$ | $b_{34}$ | $b_{35}$ | $a_6$ |

(Matriz S original al comienzo del algoritmo.)

| | Dni | NombreE | NumProyecto | NombreProyecto | UbicaciónProyecto | Horas |
|---|---|---|---|---|---|---|
| $R_1$ | $a_1$ | $a_2$ | $b_{13}$ | $b_{14}$ | $b_{15}$ | $b_{16}$ |
| $R_2$ | $b_{21}$ | $b_{22}$ | $a_3$ | $a_4$ | $a_5$ | $b_{26}$ |
| $R_3$ | $a_1$ | ~~$b_{32}$~~ $a_2$ | $a_3$ | ~~$b_{34}$~~ $a_4$ | ~~$b_{35}$~~ $a_5$ | $a_6$ |

(Matriz S después de aplicar las dos primeras dependencias funcionales;
la última fila sólo contiene símbolos “a”, por lo que paramos)

### 11.1.4 Comprobación de la propiedad de concatenación no aditiva en descomposiciones binarias

El Algoritmo 11.1 nos permite comprobar si una descomposición $D$ particular en $n$ relaciones cumple la propiedad de concatenación no aditiva respecto a un conjunto de dependencias funcionales $F$. Existe un caso especial llamado **descomposición binaria**: la descomposición de una relación $R$ en dos relaciones. Vamos a dar un método más simple que el Algoritmo 11.1 que, aunque es muy cómodo de utilizar, sólo puede aplicarse a las descomposiciones binarias.

> **Propiedad NJB (Comprobación de concatenación no aditiva para descomposiciones binarias).** Una descomposición $D = \{R_1, R_2\}$ de $R$ tiene la propiedad de concatenación sin pérdida (no aditiva) respecto a un conjunto de dependencias funcionales $F$ en $R$ *si y sólo si*:
>
> - La DF $((R_1 \cap R_2) \rightarrow (R_1 - R_2))$ está en $F^+$, o bien,
> - La DF $((R_1 \cap R_2) \rightarrow (R_2 - R_1))$ está en $F^+$.

Es preciso comprobar que esta propiedad se cumple con respecto a nuestros ejemplos informales de normalización sucesiva de las Secciones 10.3 y 10.4.

### 11.1.5 Descomposiciones de concatenación no aditiva sucesivas

En las Secciones 10.3 y 10.4 vimos una descomposición sucesiva de relaciones durante el proceso de la segunda y tercera normalización. Para comprobar que estas descomposiciones son no aditivas, necesitamos garantizar otra propiedad, lo que desemboca en la Afirmación 2.

> **Afirmación 2 (Conservación de la no aditividad en descomposiciones sucesivas).** Si una descomposición $D = \{R_1, R_2, \ldots, R_m\}$ de $R$ tiene la propiedad de concatenación no aditiva respecto a un conjunto de dependencias funcionales $F$ en $R$, y si una descomposición $D_i = \{Q_1, Q_2, \ldots, Q_k\}$ de $R_i$ también tiene la propiedad en relación a la proyección de $F$ en $R_i$, entonces la descomposición $D_2 = \{R_1, R_2, \ldots, R_{i-1}, Q_1, Q_2, \ldots, Q_k, R_{i+1}, \ldots, R_m\}$ de $R$ cuenta a su vez con la propiedad de concatenación no aditiva respecto a $F$.

## 11.2 Algoritmos para el diseño de un esquema de base de datos relacional

Ahora vamos a ver tres algoritmos para la creación de una descomposición relacional a partir de una relación universal. Cada algoritmo tiene propiedades específicas, como veremos más adelante.

### 11.2.1 Descomposición de la conservación de dependencias en esquemas 3FN

El Algoritmo 11.2 crea una descomposición de la conservación de dependencias $D = \{R_1, R_2, \ldots, R_m\}$ de una relación universal $R$ basada en un conjunto de dependencias funcionales $F$, de modo que cada $R_i$ en $D$ esté en 3FN. Sólo garantiza la propiedad de conservación de las dependencias, no la de concatenación no aditiva. El primer paso del Algoritmo 11.2 es localizar una cobertura mínima $G$ para $F$; para ello, puede utilizarse el Algoritmo 10.2. Pueden existir varias coberturas mínimas para un conjunto $F$ (como quedará demostrado en el ejemplo que sigue al Algoritmo 11.2). En estos casos, los algoritmos pueden producir potencialmente múltiples diseños alternativos.

> **Algoritmo 11.2.** Síntesis relacional en 3FN con conservación de las dependencias:
>
> **Entrada:** Una relación universal $R$ y un conjunto de dependencias funcionales $F$ en los atributos de $R$.

1. Localizar una cobertura mínima $G$ para $F$ (utilice el Algoritmo 10.2);
2. Por cada $X$ izquierdo de una dependencia funcional que aparezca en $G$, crear un esquema de relación en $D$ con los atributos $\{X \cup \{A_1\} \cup \{A_2\} \ldots \cup \{A_k\}\}$, donde $X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_k$ son las únicas dependencias en $G$ que tienen $X$ como parte izquierda ($X$ es la clave de esta relación);
3. Situar cualquier atributo que sobre (los que no se hayan podido colocar en ninguna relación) en un esquema de relación simple que asegure la propiedad de conservación de los atributos.

**Ejemplo del Algoritmo 11.2.** Considere la siguiente relación universal:

$U$(DniEmpleado, NumProy, SalarioEmpleado, TlfEmpleado, NúmeroDpto, NombreProyecto, UbicaciónProyecto)

DniEmpleado, SalarioEmpleado y TlfEmpleado hacen referencia al Documento Nacional de Identidad, el salario y el teléfono del empleado. NumProy, NombreProyecto y UbicaciónProyecto son el número, el nombre y la localización del proyecto. NúmeroDpto es el número de departamento.

Las siguientes dependencias están presentes:

DF1: DniEmpleado→SalarioEmpleado, TlfEmpleado, NúmeroDpto
DF2: NumProy→NombreProyecto, UbicaciónProyecto
DF3: DniEmpleado, NumProy→SalarioEmpleado, TlfEmpleado, NúmeroDpto, NombreProyecto, UbicaciónProyecto

En virtud de la DF3, el conjunto de atributos {DniEmpleado, NumProy} es una clave de la relación universal. Así pues, $F$, el conjunto de DFs dado, incluye {DniEmpleado → SalarioEmpleado, TlfEmpleado, NúmeroDpto; NumProy→NombreProyecto, UbicaciónProyecto; DniEmpleado, NumProy→SalarioEmpleado, TlfEmpleado, NúmeroDpto, NombreProyecto, UbicaciónProyecto}.

Aplicando el Algoritmo 10.2 de cobertura mínima, en el paso 3 vemos que NumProy es un atributo redundante en DniEmpleado, NumProy→SalarioEmpleado, TlfEmpleado, NúmeroDpto. Además, DniEmpleado es redundante en DniEmpleado, NumProy→NombreProyecto, UbicaciónProyecto. Por consiguiente, la cobertura mínima constará de DF1 y DF2 (DF3 es completamente redundante) de la siguiente forma (si agrupamos los atributos con la misma LHS en una DF):

Cobertura mínima $G$: {DniEmpleado→SalarioEmpleado, TlfEmpleado, NúmeroDpto; NumProy→NombreProyecto, UbicaciónProyecto}

Aplicando el Algoritmo 11.2 a $G$, obtenemos un diseño 3FN que consta de dos relaciones con las claves DniEmpleado y NumProy:

$R_1$ (DniEmpleado, SalarioEmpleado, TlfEmpleado, NúmeroDpto)

$R_2$ (NumProy, NombreProyecto, UbicaciónProyecto)

Un lector observador habrá visto fácilmente que estas dos relaciones han perdido la información original contenida en la clave de la relación universal U (digamos que existen ciertos empleados que trabajan en ciertos proyectos en una relación de muchos-a-muchos). Así, mientras que el algoritmo conserva las dependencias originales, no garantiza que se mantenga toda la información. Por tanto, el diseño resultante es *pobre*.

> **Afirmación 3.** Todo esquema de relación creado por el Algoritmo 11.2 está en 3FN (no daremos aquí una comprobación formal;[2] la comprobación está condicionada a que $G$ sea un conjunto mínimo de dependencias).

Es obvio que todas las dependencias en $G$ son conservadas por el algoritmo, ya que cada una de ellas aparece en una de las relaciones $R_i$ de la descomposición $D$. Ya que $G$ es equivalente a $F$, todas las dependencias

[2] Si desea una demostración, consulte Maier (1983) o Ullman (1982).

en $F$ también se conservan directamente en la descomposición, o puede derivarse de ellas usando las reglas de inferencia de la Sección 10.2.2 en las relaciones resultantes, lo que asegura la propiedad de conservación de las dependencias. El Algoritmo 11.2 recibe el nombre de **algoritmo de síntesis relacional**, porque cada esquema de relación $R_i$ en la descomposición se sintetiza (construye) a partir del conjunto de dependencias funcionales en $G$ con la misma $X$ izquierda.

### 11.2.2 Descomposición de concatenación no aditiva en esquemas BCNF

El siguiente algoritmo descompone una relación universal $R = \{A_1, A_2, \ldots, A_n\}$ en una descomposición $D = \{R_1, R_2, \ldots, R_m\}$ de forma que cada $R_i$ esté en BCNF *y* que la descomposición $D$ tenga la propiedad de concatenación sin pérdida respecto a $F$. El Algoritmo 11.3 utiliza la propiedad NJB y la Afirmación 2 (conservación de la no aditividad en descomposiciones sucesivas) para crear una descomposición de concatenación no aditiva $D = \{R_1, R_2, \ldots, R_m\}$ de una relación universal $R$ basada en un conjunto de dependencias funcionales $F$, de forma que cada $R_i$ en $D$ esté en BCNF.

**Algoritmo 11.3.** Descomposición relacional en BCNF con la propiedad de concatenación no aditiva:

**Entrada:** Una relación universal $R$ y un conjunto de dependencias funcionales $F$ en los atributos de $R$.

**1.** Establecer $D := \{R\}$ ;

**2.** Mientras que exista un esquema de relación $Q$ en $D$ que no sea BCNF hacer lo siguiente:

```
{
   elegir un esquema de relación Q en D que no esté en BCNF;
   localizar una dependencia funcional X→Y en Q que viole BCNF;
   reemplazar Q en D con dos esquemas de relación (Q – Y) y (X ∪ Y);
} ;
```

Cada pasada por el bucle del Algoritmo 11.3 descompone un esquema de relación $Q$ que no está en BCNF en dos esquemas de relación. Según la propiedad NJB para las descomposiciones binarias y la Afirmación 2, la descomposición $D$ tiene la propiedad de concatenación no aditiva. Al final del algoritmo, todos los esquemas de relación en $D$ estarán en BCNF. El lector puede comprobar que el ejemplo de normalización de las Figuras 10.11 y 10.12 siguen básicamente este algoritmo. Las dependencias funcionales DF3, DF4 y la tardía DF5 violan la BCNF, por lo que la relación PARCELAS se descompone apropiadamente en relaciones BCNF, lo que satisface la propiedad de concatenación no aditiva. De forma parecida, si aplicamos el algoritmo al esquema de relación ENSEÑAR de la Figura 10.13, éste se descompone en ENSEÑAR1(Profesor, Estudiante) y ENSEÑAR2(Profesor, Curso) porque la dependencia DF2: Profesor→Curso viola la BCNF.

En el segundo paso del Algoritmo 11.3, es necesario determinar si un esquema de relación $Q$ está en BCNF o no. Un método para lograrlo es comprobar, por cada dependencia funcional $X \rightarrow Y$ en $Q$, si $X^+$ falla al incluir todos los atributos en $Q$, para así determinar si $X$ es o no una (súper)clave en $Q$. Otra técnica está basada en una observación que dice que siempre que un esquema de relación $Q$ viola la BCNF existe un par de atributos $A$ y $B$ en $Q$ que hacen que $\{Q - \{A, B\}\} \rightarrow A$; procesando la clausura $\{Q - \{A, B\}\}^+$ para cada par de atributos $\{A, B\}$ de $Q$, y verificando si esa clausura incluye $A$ (o $B$), podemos determinar si $Q$ está en BCNF.

### 11.2.3 Conservación de las dependencias y descomposición de concatenación no aditiva (sin pérdida) en esquemas 3FN

Si queremos que una descomposición tenga la propiedad de concatenación no aditiva *y* que conserve las dependencias, tenemos que ajustar los esquemas de relación a 3FN en lugar de a BCNF. Una sencilla modificación del Algoritmo 11.2, que se muestra en el Algoritmo 11.4, consigue que una descomposición $D$ de $R$ cumpla lo siguiente:

- Conservar las dependencias.
- Tener la propiedad de concatenación no aditiva.
- Que cada esquema de relación resultante de la descomposición esté en 3FN.

Ya que el Algoritmo 11.4 logra las dos propiedades deseables, en lugar de sólo la de conservación de las dependencias funcionales garantizada por el Algoritmo 11.2, podemos considerarlo como un algoritmo preferente.

**Algoritmo 11.4.** Síntesis relacional en 3FN con conservación de las dependencias y propiedad de concatenación no aditiva:

**Entrada:** Una relación universal $R$ y un conjunto de dependencias funcionales $F$ en los atributos de $R$.

1. Localizar una cobertura mínima $G$ para $F$ (utilice el Algoritmo 10.2).
2. Por cada $X$ izquierdo de una dependencia funcional que aparezca en $G$, crear un esquema de relación en $D$ con los atributos $\{X \cup \{A_1\} \cup \{A_2\} \ldots \cup \{A_k\}\}$, donde $X \rightarrow A_1$, $X \rightarrow A_2$, . . . , $X \rightarrow A_k$ son las únicas dependencias en $G$ que tienen $X$ como parte izquierda ($X$ es la clave de esta relación).
3. Si ninguno de los esquemas de relación en $D$ contienen una clave de $R$, crear un esquema de relación más en $D$ que contenga los atributos que forman una clave de $R$.[3] (Puede utilizarse el Algoritmo 11.4(a) para localizar una clave.)
4. Eliminar las relaciones redundantes del conjunto de relaciones resultante en el esquema de base de datos relacional. Una relación $R$ se considera como redundante si es una proyección de otra relación $S$ en el esquema; alternativamente, $R$ está abarcada por $S$.[4]

El paso 3 del Algoritmo 11.4 implica la identificación de una clave $K$ de $R$. Puede usarse el Algoritmo 11.4(a) para identificarla basándonos en el conjunto dado de dependencias funcionales $F$. Empezamos asignando $K$ a todos los atributos de $R$; a continuación, eliminamos un atributo cada vez y comprobamos si los que quedan siguen formando una superclave. Observe que el conjunto de dependencias funcionales usado para determinar una clave en el Algoritmo 11.4(a) podría ser $F$ o $G$, ya que ambos son equivalentes. Observe también que el Algoritmo 11.4(a) sólo determina *una clave* de entre todas las posibles claves candidatas de $R$; la que se devuelva dependerá del orden en el que se eliminan los atributos de $R$ en el paso 2.

**Algoritmo 11.4(a).** Localización de una clave $K$ para $R$ dado un conjunto $F$ de dependencias funcionales:

**Entrada:** Una relación universal $R$ y un conjunto de dependencias funcionales $F$ en los atributos de $R$.

1. Establecer $K := R$.
2. Por cada atributo $A$ en $K$

   {procesar $(K - A)^+$ respecto a $F$;
   si $(K - A)^+$ contiene todos los atributos en $R$, entonces establecer $K := K - \{A\}$ };

**Ejemplo 1 del Algoritmo 11.4.** Vamos a retomar el ejemplo dado al final del Algoritmo 11.2. La cobertura mínima $G$ se mantiene como antes. El segundo paso produce las relaciones $R_1$ y $R_2$ igual que antes. Sin embargo, ahora en el paso 3 se generará una relación correspondiente a la clave {DniEmpleado, NumProy}. Por consiguiente, el diseño resultante contiene:

---

[3] El paso 3 del Algoritmo 11.2 no es necesario en el Algoritmo 11.4 para conservar los atributos, porque la clave incluirá todos los que no están colocados; se trata de los atributos que no participan en ninguna dependencia funcional.

[4] Observe que hay un tipo adicional de dependencia: $R$ es una proyección de la concatenación de dos o más relaciones en el esquema. Este tipo de redundancia está considerada como de concatenación, como se verá más adelante en la Sección 11.4. De este modo, técnicamente, podría seguir existiendo sin desestabilizar el estado 3FN del esquema.

$R_1$ (DniEmpleado, SalarioEmpleado, TlfEmpleado, NúmeroDpto)

$R_2$ (NumProy, NombreProyecto, UbicaciónProyecto)

$R_3$ (DniEmpleado, NumProy)

Este diseño consigue las dos propiedades deseables de conservación de las dependencias y concatenación no aditiva.

**Ejemplo 2 del Algoritmo 11.4 (caso *X*).** Considere el esquema de relación PARCELAS1A de la Figura 10.12(a). Asumimos que es una relación universal que tiene las siguientes dependencias funcionales:

DF1: IdPropiedad→NúmeroParcela, NombreMunicipio, Área

DF2: NúmeroParcela, NombreMunicipio→Área, IdPropiedad

DF3: Área→NombreMunicipio

Los nombres que recibieron en la Figura 10.12(a) eran DF1, DF2 y DF5. El significado de los atributos anteriores y la implicación de las dependencias funcionales se explicaron en la Sección 10.4. Para facilitar la referencia, abreviaremos los atributos como IdPropiedad = P, NúmeroParcela = L, NombreMunicipio = C y Área=A, y representaremos las dependencias funcionales como el conjunto:

$F$ : { P → LCA, LC → AP, A → C }.

Si aplicamos el Algoritmo 10.2 de cobertura mínima a $F$, (en el paso 2) representamos primero el conjunto $F$ como:

$F$ : {P → L, P → C, P → A, LC → A, LC → P, A → C}.

En el conjunto $F$, podemos inferir P→A de P→LC y LC→A; P→A se da por transitividad y es, por consiguiente, redundante. De este modo, una posible cobertura mínima sería:

Cobertura mínima $GX$: {P→LC, LC → AP, A → C}.

En el paso 2 del Algoritmo 11.4 se produce el diseño $X$ (antes de eliminar relaciones redundantes) como:

Diseño $X$: $R_1$ (P, L, C), $R_2$ (L, C, A, P) y $R_3$ (A, C).

En el paso 4 del algoritmo, vemos que $R_3$ está incluida en $R_2$, es decir, $R_3$ es siempre una proyección de $R_2$ al igual que $R_1$. Por tanto, ambas relaciones son redundantes. De este modo, el esquema 3FN que logra las dos propiedades deseables es (tras eliminar las relaciones redundantes):

Diseño $X$: $R_2$ (L, C, A, P).

o, en otras palabras, es idéntica a la relación PARCELAS1A (NúmeroParcela, NombreMunicipio, Área, IdPropiedad) que en la Sección 10.4.2 determinamos que estaba en 3FN.

**Ejemplo 2 del Algoritmo 11.4 (caso *Y* ).** Empezando con PARCELAS1A como la relación universal ,y dado el mismo conjunto de dependencias funcionales, el segundo paso del Algoritmo 10.2 de cobertura mínima produce, como antes:

$F$: {P → C, P → A, P → L, LC → A, LC → P, A → C}.

La DF LC → A podría considerarse como redundante porque LC → P y P → A implica LC → A por transitividad. Lo mismo podríamos decir de P→C, ya que P → A y A → C implican P → C también por transitividad. Esto proporciona una cobertura mínima diferente:

Cobertura mínima $GY$: { P → LA, LC → P, A → C }.

El diseño alternativo $Y$ producido por el algoritmo es ahora:

Diseño $Y$: $S_1$ (P, A, L), $S_2$ (L, C, P) y $S_3$ (A, C).

Observe que este diseño tiene tres relaciones 3FN, de la cuales ninguna puede considerarse como redundante por la condición del paso 4. Todas las DF del conjunto $F$ original se mantienen. El lector habrá observado que fuera de las tres relaciones anteriores, $S_1$ y $S_3$ se produjeron como el diseño BCNF según el procedimiento de la Sección 10.5 (lo que implica que $S_2$ es redundante en presencia de $S_1$ y $S_3$). Sin embargo, no podemos eliminar $S_2$ del conjunto de las tres relaciones 3FN ya que no es una proyección ni de $S_1$ ni $S_3$. El diseño $Y$ permanece, por consiguiente, como uno de los posibles resultados de la aplicación del Algoritmo 11.4 en la relación universal que genera las relaciones 3FN.

Es importante saber que la teoría de las descomposiciones de concatenación no aditiva se basan en la suposición de que *no se permiten valores NULL en los atributos de concatenación*. La siguiente sección trata algunos de los problemas que los NULL pueden causar en las descomposiciones relacionales.

### 11.2.4 Problemas con los valores NULL y las tuplas colgantes

A la hora de diseñar un esquema de bases de datos relacional debemos prestar mucha atención a los problemas derivados de los NULL. Hasta ahora, no existe ningún diseño relacional satisfactorio que incluya estos valores. Un problema aparece cuando alguna tupla tiene valores NULL en los atributos que se usarán para concatenar las relaciones individuales en la descomposición. Para demostrar esto, considere la base de datos de la Figura 11.2(a), donde se muestran las relaciones EMPLEADO y DEPARTAMENTO. Las dos últimas tuplas de empleado ('Palomo' y 'Benítez') representan a dos empleados recientemente contratados que aún no están asignados a un departamento (asumimos que esta situación no viola ninguna restricción de integridad). Ahora vamos a suponer que queremos recuperar una lista de valores (NombreE, NombreDpto) de todos los empleados. Si aplicamos la CONCATENACIÓN NATURAL en EMPLEADO y DEPARTAMENTO (Figura 11.2[b]), las dos tuplas antes mencionadas *no* aparecerán en el resultado. La operación CONCATENACIÓN EXTERNA, comentada en el Capítulo 6, puede resolver este problema. Recuerde que si tomamos la CONCATENACIÓN EXTERNA IZQUIERDA de EMPLEADO con DEPARTAMENTO, las tuplas de EMPLEADO que tengan NULL en los atributos de concatenación aparecerán en el resultado unidas con una tupla *imaginaria* de DEPARTAMENTO que tiene valores NULL en todos sus atributos. La Figura 11.2(c) muestra el resultado.

En general, siempre que el esquema de una base de datos relacional esté diseñado de forma que dos o más relaciones estén interrelacionadas mediante *foreign keys*, debemos prestar especial atención para localizar potenciales valores NULL en dichas claves, ya que esto puede provocar pérdidas inesperadas de información en las consultas que implican concatenación de ellas. Además, si aparecen NULLs en otros atributos, como el salario, debe evaluarse con cuidado su efecto en funciones predefinidas como SUM y AVERAGE.

Un problema relacionado es el de las *tuplas colgantes*, que se pueden generar si llevamos a cabo una descomposición demasiado lejos. Supongamos que descomponemos la relación EMPLEADO de la Figura 11.2(a) más allá de EMPLEADO_1 y EMPLEADO_2 (véanse las Figuras 11.3[a] y 11.3[b]).[5] Si aplicamos una operación CONCATENACIÓN NATURAL a ambas relaciones, obtenemos la relación EMPLEADO original. Sin embargo, podemos usar la representación alternativa de la Figura 11.3(c), donde *no se incluye una tupla* en EMPLEADO_3 si el empleado aún no ha sido asignado a ningún departamento (en lugar de generar una con valores NULL en NumDptoProyecto como ocurre en EMPLEADO_2). Si usamos EMPLEADO_3 en lugar de EMPLEADO_2 y aplicamos una CONCATENACIÓN NATURAL en EMPLEADO_1 y EMPLEADO_3, las tuplas correspondientes a Palomo y Benítez no aparecerán en el resultado; esto es lo que se conoce como **tuplas colgantes**, porque sólo están representadas en una de las dos relaciones que representan a empleados y que, por tanto, se perderían si se aplicara una operación CONCATENACIÓN (INTERNA).

[5] Esto ocurre a veces si se aplica una fragmentación vertical a la relación en el contexto de una base de datos distribuida (consulte el Capítulo 25).

**Figura 11.2.** Problemas que aparecen en las concatenaciones con valores NULL. (a) Alguna tupla EMPLEADO tiene NULLs en el atributo de concatenación NumDptoProyecto. (b) Resultado de aplicar una CONCATENACIÓN NATURAL a las relaciones EMPLEADO y DEPARTAMENTO. (c) Resultado de aplicar una CONCATENACIÓN EXTERNA IZQUIERDA a EMPLEADO y DEPARTAMENTO.

**(a)**

**EMPLEADO**

| NombreE | Dni | FechaNac | Dirección | NúmeroDpto |
|---|---|---|---|---|
| Pérez Pérez, José | 123456789 | 09-01-1965 | Eloy I, 98 | 5 |
| Campos Sastre, Alberto | 333445555 | 08-12-1955 | Avda. Ríos, 9 | 5 |
| Jiménez Celaya, Alicia | 999887777 | 19-07-1968 | Gran Vía, 38 | 4 |
| Sainz Oreja, Juana | 987654321 | 20-06-1941 | Cerquillas, 67 | 4 |
| Ojeda Ordóñez, Fernando | 666884444 | 15-09-1962 | Portillo, s/n | 5 |
| Oliva Avezuela, Aurora | 453453453 | 31-07-1972 | Antón, 6 | 5 |
| Pajares Morera, Luis | 987987987 | 29-03-1969 | Enebros, 90 | 4 |
| Ochoa Paredes, Eduardo | 888665555 | 10-11-1937 | Las Peñas, 1 | 1 |
| Palomo González, Andrés | 999775555 | 26-04-1965 | Alameda, 3 | NULL |
| Benítez Gallego, Carlos | 888664444 | 09-01-1963 | Paseo del río, 45 | NULL |

**DEPARTAMENTO**

| NombreDpto | NúmeroDpto | DniDirector |
|---|---|---|
| Investigación | 5 | 333445555 |
| Administración | 4 | 987654321 |
| Sede Central | 1 | 888665555 |

**(b)**

| NombreE | Dni | FechaNac | Dirección | NúmeroDpto | NombreDpto | DniDirector |
|---|---|---|---|---|---|---|
| Pérez Pérez, José | 123456789 | 09-01-1965 | Eloy I, 98 | 5 | Investigación | 333445555 |
| Campos Sastre, Alberto | 333445555 | 08-12-1955 | Avda. Ríos, 9 | 5 | Investigación | 333445555 |
| Jiménez Celaya, Alicia | 999887777 | 19-07-1968 | Gran Vía, 38 | 4 | Administración | 987654321 |
| Sainz Oreja, Juana | 987654321 | 20-06-1941 | Cerquillas, 67 | 4 | Administración | 987654321 |
| Ojeda Ordóñez, Fernando | 666884444 | 15-09-1962 | Portillo, s/n | 5 | Investigación | 333445555 |
| Oliva Avezuela, Aurora | 453453453 | 31-07-1972 | Antón, 6 | 5 | Investigación | 333445555 |
| Pajares Morera, Luis | 987987987 | 29-03-1969 | Enebros, 90 | 4 | Administración | 987654321 |
| Ochoa Paredes, Eduardo | 888665555 | 10-11-1937 | Las Peñas, 1 | 1 | Sede central | 888665555 |

**(c)**

| NombreE | Dni | FechaNac | Dirección | NúmeroDpto | NombreDpto | DniDirector |
|---|---|---|---|---|---|---|
| Pérez Pérez, José | 123456789 | 09-01-1965 | Eloy I, 98 | 5 | Investigación | 333445555 |
| Campos Sastre, Alberto | 333445555 | 08-12-1955 | Avda. Ríos, 9 | 5 | Investigación | 333445555 |
| Jiménez Celaya, Alicia | 999887777 | 19-07-1968 | Gran Vía, 38 | 4 | Administración | 987654321 |
| Sainz Oreja, Juana | 987654321 | 20-06-1941 | Cerquillas, 67 | 4 | Administración | 987654321 |
| Ojeda Ordóñez, Fernando | 666884444 | 15-09-1962 | Portillo, s/n | 5 | Investigación | 333445555 |
| Oliva Avezuela, Aurora | 453453453 | 31-07-1972 | Antón, 6 | 5 | Investigación | 333445555 |
| Pajares Morera, Luis | 987987987 | 29-03-1969 | Enebros, 90 | 4 | Administración | 987654321 |
| Ochoa Paredes, Eduardo | 888665555 | 10-11-1937 | Las Peñas, 1 | 1 | Sede Central | 888665555 |
| Palomo González, Andrés | 999775555 | 26-04-1965 | Alameda, 3 | NULL | NULL | NULL |
| Benítez Gallego, Carlos | 888665555 | 09-01-1963 | Paseo del río, 45 | NULL | NULL | NULL |

**Figura 11.3.** El problema de la tupla colgante. (a) La relación EMPLEADO_1 (incluye todos los atributos de EMPLEADO de la Figura 11.2[a], excepto NumDptoProyecto). (b) La relación EMPLEADO_2 (incluye los atributos NumDptoProyecto con valores NULL). (c) La relación EMPLEADO_3 (incluye NumDptoProyecto, pero no las tuplas en las que este atributo tiene valores NULL).

**(a) EMPLEADO_1**

| NombreE | Dni | FechaNac | Dirección |
|---|---|---|---|
| Pérez Pérez, José | 123456789 | 09-01-1965 | Eloy I, 98 |
| Campos Sastre, Alberto | 333445555 | 08-12-1955 | Avda. Ríos, 9 |
| Jiménez Celaya, Alicia | 999887777 | 19-07-1968 | Gran Vía, 38 |
| Sainz Oreja, Juana | 987654321 | 20-06-1941 | Cerquillas, 67 |
| Ojeda Ordóñez, Fernando | 666884444 | 15-09-1962 | Portillo, s/n |
| Oliva Avezuela, Aurora | 453453453 | 31-07-1972 | Antón, 6 |
| Pajares Morera, Luis | 987987987 | 29-03-1969 | Enebros, 90 |
| Ochoa Paredes, Eduardo | 888665555 | 10-11-1937 | Las Peñas, 1 |
| Palomo González, Andrés | 999775555 | 26-04-1965 | Alameda, 3 |
| Benítez Gallego, Carlos | 888665555 | 09-01-1963 | Paseo del río, 45 |

**(b) EMPLEADO_2**

| Dni | NúmeroDpto |
|---|---|
| 123456789 | 5 |
| 333445555 | 5 |
| 999887777 | 4 |
| 987654321 | 4 |
| 666884444 | 5 |
| 453453453 | 5 |
| 987987987 | 4 |
| 888665555 | 1 |
| 999775555 | NULL |
| 888664444 | NULL |

**(c) EMPLEADO_3**

| Dni | NúmeroDpto |
|---|---|
| 123456789 | 5 |
| 333445555 | 5 |
| 999887777 | 4 |
| 987654321 | 4 |
| 666884444 | 5 |
| 453453453 | 5 |
| 987987987 | 4 |
| 888665555 | 1 |

## 11.2.5 Normalización de algoritmos

Uno de los problemas que existen en los algoritmos de normalización que hemos descrito es que el diseñador de la base de datos debe especificar primero *todas* las dependencias funcionales relevantes entre los atributos de la base de datos. Esto no es una tarea simple en bases de datos grandes con cientos de atributos. Un error a la hora de especificar una o dos dependencias importantes puede desembocar en un diseño ilegible. Otro problema es que estos algoritmos son, en general, *no deterministas*. Por ejemplo, los *algoritmos de síntesis* (Algoritmos 11.2 y 11.4) precisan de la especificación de una cobertura mínima $G$ para un conjunto de dependencias funcionales $F$. Debido a que pueden existir muchas de estas coberturas mínimas para $F$, como quedó demostrado en el Ejemplo 2 del Algoritmo 11.4, éste puede generar diferentes diseños dependiendo de la cobertura usada. Algunos de estos diseños pueden no ser adecuados. El *Algoritmo de descomposición* (Algoritmo 11.3) depende del orden en el que se le suministran las dependencias funcionales para verificar la violación de la BCNF. De nuevo, es posible que se generen muchos diseños diferentes para el mismo conjunto de dependencias funcionales, en función del orden en el que son consideradas dichas dependencias en la

comprobación de la BCNF. Algunos de los diseños pueden ser bastante superiores, mientras que otros pueden no ser adecuados.

No siempre es posible encontrar una descomposición en los esquemas de relación que conserve las dependencias y permita que cada esquema de relación en la descomposición esté en BCNF (en lugar de en 3FN como ocurre en el Algoritmo 11.4). Podemos verificar individualmente los esquemas de relación 3FN en la descomposición para ver si cada una de ellas satisface la BCNF. Si algún esquema de relación $R_i$ no está en BCNF, podemos elegir entre efectuar más descomposiciones o dejarlo en 3FN (con el riesgo de posibles anomalías de actualización).

Para ilustrar los puntos anteriores, vamos a retomar la relación PARCELAS1A de la Figura 10.12(a). Esta relación está en 3FN, pero no en BCNF como quedó demostrado en la Sección 10.5. Vimos también que al empezar con las dependencias funcionales (DF1, DF2 y DF5 de la Figura 10.12[a]) usando el planteamiento ascendente para el diseño y la aplicación del Algoritmo 11.4, es posible la generación de cualquier relación PARCELAS1A como diseño 3FN (lo que llamamos previamente diseño *X*), u otro diseño alternativo *Y* compuesto por tres relaciones ($S_1$, $S_2$, $S_3$), cada una de ellas en 3FN. Observe que si verificamos el diseño Y para BCNF, $S_1$, $S_2$ y $S_3$ se giran para estar individualmente en BCNF. Sin embargo, cuando se comprueba el diseño *X* para BCNF, la verificación falla. Produce las dos relaciones $S_1$ y $S_3$ aplicando el Algoritmo 11.3 (debido a la violación de la dependencia funcional A → C). Por tanto, el procedimiento de diseño ascendente del Algoritmo 11.4 para diseñar relaciones 3FN, conseguir las dos propiedades deseables y después aplicar el 11.3 para obtener la BCNF con la propiedad de concatenación no aditiva (sacrificando la conservación de la dependencia funcional), produce $S_1$, $S_2$ y $S_3$ como diseño BCNF final por un camino (diseño *Y*) y $S_1$ y $S_3$ por otro (diseño *X*). Esto ocurre por las distintas coberturas mínimas del conjunto de dependencias funcionales original. Observe que $S_2$ es una relación redundante en el diseño *Y*; sin embargo, no viola la restricción de concatenación no aditiva. Es fácil ver que $S_2$ es una relación válida y significativa que tiene juntas las dos claves candidatas (L, C), y P. La Tabla 11.1 resume las propiedades de los algoritmos vistos hasta el momento en este capítulo.

**Tabla 11.1** Resumen de los algoritmos explicados en las Secciones 11.1 y 11.2.

| Algoritmo | Entrada | Salida | Propiedades/Objetivo | Comentarios |
|---|---|---|---|---|
| 11.1 | Una descomposición *D* de *R* y un conjunto *F* de dependencias funcionales | Resultado booleano: sí o no para la propiedad de concatenación no aditiva | Verificación de la descomposición de concatenación no aditiva | Consulte la verificación más simple para las descomposiciones binarias en la Sección 11.1.4 |
| 11.2 | Conjunto de dependencias funcionales *F* | Un conjunto de relaciones en 3FN | Conservación de la dependencia | No garantiza cumplir la propiedad de concatenación sin pérdida |
| 11.3 | Conjunto de dependencias funcionales *F* | Un conjunto de relaciones en BCNF | Descomposición de concatenación no aditiva | No garantiza la conservación de la dependencia |
| 11.4 | Conjunto de dependencias funcionales *F* | Un conjunto de relaciones en 3FN | Concatenación no aditiva *y* descomposición por conservación de la dependencia | Puede no alcanzarse la BCNF, pero sí se consiguen *todas* las propiedades y la 3FN |
| 11.4a | Esquema de relación *R* con un conjunto de dependencias funcionales *F* | Clave K de *R* | Localizar una clave K (que es un subconjunto de *R*) | Toda la relación *R* es siempre una superclave predeterminada |

# 11.3 Dependencias multivalor y cuarta forma normal

Hasta ahora sólo hemos tratado la dependencia funcional que es, con mucho, el tipo de dependencia más importante en la teoría del diseño de una base de datos relacional. Sin embargo, en muchos casos, las relaciones tienen restricciones que no pueden especificarse como dependencias funcionales. En esta sección vamos a tratar el concepto de MVD (dependencia multivalor) y a definir la *cuarta forma normal,* la cual se basa en esta dependencia. Las dependencias multivalor son una consecuencia de la 1NF (consulte la Sección 10.3.4), que prohíbe que una tupla tenga un *conjunto de valores*. Si tenemos dos o más atributos multivalor *independientes* en el mismo esquema de relación, tenemos el problema de tener que repetir cada valor de uno de los atributos con cada valor del otro atributo para mantener la consistencia del estado de la relación y la independencia entre los atributos implicados. Esta restricción está especificada por una dependencia multivalor.

Por ejemplo, consideremos la relación EMP de la Figura 11.4(a). Una tupla de esta relación representa el hecho de que un empleado cuyo nombre es NombreE trabaja en el proyecto cuyo nombre es NombreProyecto y tiene un subordinado llamado NombreSubordinado. Un empleado puede trabajar en varios proyectos y tener varios subordinados, y sus proyectos y subordinados son independientes unos de otros.[6] Para mantener la coherencia de esta relación, debemos tener una tupla separada para representar cada combinación de subordinado y proyecto del empleado. Esta restricción se especifica como una dependencia multivalor en EMP. De manera informal, siempre que dos relaciones 1:N *independientes* $A$:$B$ y $A$:$C$ se mezclen en la misma relación se alcanza $R(A, B, C)$ y una MVD.

## 11.3.1 Definición formal de una dependencia multivalor

**Definición.** Una dependencia multivalor $X \twoheadrightarrow Y$ especificada en un esquema de relación $R$, donde $X$ e $Y$ son subconjuntos de $R$, especifica las siguientes restricciones en cualquier relación $r$ de $R$: si dos tuplas $t_1$ y $t_2$ existen en $r$ de modo que $t_1[X] = t_2[X]$, entonces también deberían existir otras dos tuplas $t_3$ y $t_4$ en $r$ con las siguientes propiedades,[7] donde utilizamos $Z$ para indicar $(R - (X \cup Y))$:[8]

- $t_3[X] = t_4[X] = t_1[X] = t_2[X]$.
- $t_3[Y] = t_1[Y]$ y $t_4[Y] = t_2[Y]$.
- $t_3[Z] = t_2[Z]$ y $t_4[Z] = t_1[Z]$.

Siempre que se cumpla $X \twoheadrightarrow Y$, decimos que $X$ **multidetermina** $Y$. Debido a la simetría de la definición, cuando se cumpla $X \twoheadrightarrow Y$ en $R$, también lo hace en $X \twoheadrightarrow Z$. Por tanto, $X \twoheadrightarrow Y$ implica $X \twoheadrightarrow Z$, por lo que a veces se escribe como $X \twoheadrightarrow Y|Z$.

La definición formal especifica que dado un valor particular de $X$, el conjunto de valores de $Y$ determinados por este valor de $X$ está completamente determinado por $X$ y *no depende* de los valores de los atributos $Z$ restantes de $R$. De este modo, siempre que existan dos tuplas con distintos valores $Y$ pero el mismo valor $X$, los valores de $Y$ deben repetirse en tuplas separadas por cada *valor distinto de Z* que se produzca con ese mismo valor de $X$. Esto hace que $Y$ sea un atributo multivalor de las entidades representadas por tuplas en $R$.

En la Figura 11.4(a), las MVD NombreE $\twoheadrightarrow$ NombreProyecto y NombreE $\twoheadrightarrow$ NombreSubordinado (o NombreE $\twoheadrightarrow$ NombreProyecto | NombreSubordinado) se cumplen en la relación EMP. El empleado con el

[6] En un diagrama ER, cada una podría representarse como un atributo multivalor o como un tipo de entidad débil (consulte el Capítulo 3).

[7] Las tuplas $t_1$, $t_2$, $t_3$ y $t_4$ no son necesariamente distintas.

[8] $Z$ es un atajo para los atributos de $R$ una vez eliminados de $R$ los atributos en $(X \cup Y)$.

**Figura 11.4.** Cuarta y quinta formas normales.
(a) La relación EMP con dos MVD: NombreE →→NombreProyecto y NombreE →→NombreSubordinado.
(b) Descomposición de la relación EMP en dos relaciones 4FN, PROYECTOS_EMP y SUBORDINADOS_EMP.
(c) La relación SUMINISTRO, sin MVD, está en 4FN, pero no en 5FN si tiene la JD($R_1$, $R_2$, $R_3$).
(d) Descomposición de la relación SUMINISTRO en las relaciones 5FN $R_1$, $R_2$, $R_3$.

**(a) EMP**

| NombreE | NombreProyecto | NombreSubordinado |
|---|---|---|
| Pérez | X | Juan |
| Pérez | Y | Ana |
| Pérez | X | Ana |
| Pérez | Y | Juan |

**(b) PROYECTOS_EMP**

| NombreE | NombreProyecto |
|---|---|
| Pérez | X |
| Pérez | Y |

**SUBORDINADOS_EMP**

| NombreE | NombreSubordinado |
|---|---|
| Pérez | Juan |
| Pérez | Ana |

**(c) SUMINISTRO**

| NombreS | NombrePieza | NombreProy |
|---|---|---|
| Pérez | Cerrojo | ProyX |
| Pérez | Tuerca | ProyY |
| Alberto | Cerrojo | ProyY |
| María | Tuerca | ProyZ |
| Alberto | Clavo | ProyX |
| Alberto | Cerrojo | ProyX |
| Pérez | Cerrojo | ProyY |

**(d)** ***R*1**

| NombreS | NombrePieza |
|---|---|
| Pérez | Cerrojo |
| Pérez | Tuerca |
| Alberto | Cerrojo |
| María | Tuerca |
| Alberto | Clavo |

***R*2**

| NombreS | NombreProy |
|---|---|
| Pérez | ProyX |
| Pérez | ProyY |
| Alberto | ProyY |
| María | ProyZ |
| Alberto | ProyX |

***R*3**

| NombrePieza | NombreProy |
|---|---|
| Cerrojo | ProyX |
| Tuerca | ProyY |
| Cerrojo | ProyY |
| Tuerca | ProyZ |
| Clavo | ProyX |

NombreE ‘Pérez’ trabaja en los proyectos cuyo NombreProyecto es ‘X’ e ‘Y’ y tiene dos subordinados con un NombreSubordinado ‘Juan’ y ‘Ana’. Si sólo almacenamos las dos primeras tuplas de EMP (<‘Pérez’, ‘X’, ‘Juan’> y <‘Pérez’, ‘Y’, ‘Ana’>), podríamos mostrar de forma incorrecta asociaciones entre el proyecto ‘X’ y ‘Juan’ y entre el proyecto ‘Y’ y ‘Ana’; podría equivaler a una relación falsa entre proyecto y subordinado, ya que no se pretende algo parecido a esto en esta relación. Por tanto, debemos almacenar las otras dos tuplas (<‘Pérez’, ‘X’, ‘Ana’> y <‘ Pérez’, ‘Y’, ‘Juan’>) para mostrar que {‘X’, ‘Y’} y {‘Juan’, ‘Ana’} sólo están asociadas con ‘Pérez’, es decir, que no existe una asociación entre NombreProyecto y NombreSubordinado, lo que significa que los dos atributos son independientes.

Una MVD $X \twoheadrightarrow Y$ en $R$ recibe el nombre de **MVD trivial** si (a) $Y$ es un subconjunto de $X$, o (b) $X \cup Y = R$. Por ejemplo, la relación PROYECTOS_EMP de la Figura 11.4(b) tiene la MVD trivial NombreE $\twoheadrightarrow$ NombreSubordinado. Una MVD que no satisface ni (a) ni (b) se conoce como **MVD no trivial**. Una MVD trivial se cumplirá en *cualquier* estado de relación $r$ de $R$; recibe el nombre de trivial porque no especifica ninguna restricción significativa en $R$.

Si tenemos una MVD no trivial en una relación, puede que tengamos que repetir valores en las tuplas. En la relación EMP de la Figura 11.4(a), los valores 'X' e 'Y' de NombreProyecto están repetidos en cada valor de NombreDpto (o, por simetría, los valores 'Juan' y 'Ana' de NombreSubordinado están repetidos con cada uno de los valores de NombreProyecto). Esta redundancia es, claramente, indeseable. Sin embargo, el esquema EMP está en BCNF porque *no* se almacenan dependencias en él. Por consiguiente, tenemos que definir una cuarta forma normal que sea más estricta que la BCNF y prohíba esquemas de relación como EMP. Primero trataremos algunas de las propiedades de las MVD y consideraremos cómo están relacionadas con las dependencias funcionales. Observe que las relaciones que contienen MVDs no triviales tienden a ser **relaciones todo clave**, es decir, su clave está compuesta por todos sus atributos juntos. Adicionalmente, es raro que relaciones de este tipo con una ocurrencia combinatoria de valores repetidos se produzca en la práctica. Sin embargo, el reconocimiento de las MVD como una dependencia potencialmente problemática es esencial en el diseño relacional.

### 11.3.2 Reglas de inferencia para dependencias funcionales y multivalor

Al igual que ocurre con las dependencias funcionales (DF), se han desarrollado las reglas de inferencia para las MVD. Lo mejor es desarrollar un entorno unificado que incluya tanto las DF como las MVD, de forma que ambos tipos de restricciones pueden ser tratados en conjunto. Las reglas de inferencia de la RI1 a la RI8 componen un lógico y completo conjunto para inferir dependencias funcionales y multivalor a partir de un conjunto concreto de dependencias. Asumimos que todos los atributos están incluidos en una *relación universal* $R = \{A_1, A_2, ..., A_n\}$ y que $X$, $Y$, $Z$ y $W$ son subconjuntos de $R$.

RI1 (regla reflexiva para las DF): si $X \supseteq Y$, entonces $X \rightarrow Y$.

RI2 (regla de aumento para las DF): $\{X \rightarrow Y\} \models XZ \rightarrow YZ$.

RI3 (regla transitiva para las DF): $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$.

RI4 (regla de complementación para las MVD): $\{X \twoheadrightarrow Y\} \models \{X \twoheadrightarrow (R - (X \cup Y))\}$.

RI5 (regla de aumento para las MVD): si $X \twoheadrightarrow Y$ y $W \supseteq Z$, entonces $WX \twoheadrightarrow YZ$.

RI6 (regla transitiva para las MVD): $\{X \twoheadrightarrow Y, Y \twoheadrightarrow Z\} \models X \twoheadrightarrow (Z - Y)$.

RI7 (regla de duplicación de las DF a las MVD): $\{X \rightarrow Y\} \models X \twoheadrightarrow Y$.

RI8 (regla de coalescencia para las DF y las MVD): si $X \twoheadrightarrow Y$ y existe $W$ con las propiedades de que (a) $W \cap Y$ está vacío, (b) $W \rightarrow Z$ y (c) $Y \supseteq Z$, entonces $X \rightarrow Z$.

Las reglas de la RI1 a la RI3 son las reglas de inferencia de Armstrong sólo para las DF. De la RI4 a la RI6 pertenecen sólo a las MVD. Por último, la RI7 y la RI8 relacionan las DF y las MVD. En particular, la RI7 dice que una dependencia funcional es un *caso especial* de una dependencia multivalor, es decir, cada DF es también una MVD porque satisface la definición formal de una MVD. Sin embargo, esta equivalencia tiene un truco: una DF $X \rightarrow Y$ es una MVD $X \twoheadrightarrow Y$ con la *restricción adicional implícita* de que a lo sumo un valor de $Y$ está asociado con cada valor de $X$.[9] Dado un conjunto $F$ de dependencias funcionales y multivalor especificadas en $R = \{A_1, A_2, ..., A_n\}$, podemos usar las reglas de la RI1 a la RI8 para inferir el conjunto

[9] Es decir, el conjunto de valores de Y determinados por un valor de X está restringido a ser un conjunto simple con un único valor. Por tanto, en la práctica, nunca vemos una DF como una MVD.

(completo) de todas las dependencias (funcionales o multivalor) $F^+$ que se almacenarán en cada relación $r$ de $R$ que satisface $F$. De nuevo, llamamos a $F^+$ la **clausura** de $F$.

### 11.3.3 Cuarta forma normal

Ahora vamos a presentar la definición de la **cuarta forma normal** (**4FN**), la cual se viola cuando una relación tiene dependencias multivalor no deseadas y, por tanto, puede usarse para identificar y descomponer relaciones de este tipo.

> **Definición.** Un esquema de relación $R$ está en 4FN respecto a un conjunto de dependencias $F$ (que incluye dependencias funcionales y multivalor) si, por cada dependencia multivalor *no trivial* $X \twoheadrightarrow Y$ en $F^+$, $X$ es una superclave de $R$.

La relación EMP de la Figura 11.4(a) no tiene DF, ya que es una relación todo-clave. Ya que las restricciones BCNF están formuladas sólo en términos de DFs, todas las relaciones todo-clave están siempre en BCNF por defecto. Por consiguiente, EMP está en BCNF pero no en 4FN porque en las MVD no triviales NombreE $\twoheadrightarrow$ NombreProyecto y NombreE $\twoheadrightarrow$ NombreSubordinado, y NombreE no es una superclave de EMP. Descomponemos EMP en PROYECTOS_EMP y SUBORDINADOS_EMP (véase la Figura 11.4[b]). Ambas están en 4FN, ya que las MVD NombreE $\twoheadrightarrow$ NombreProyecto en PROYECTOS_EMP y NombreE $\twoheadrightarrow$ NombreSubordinado en SUBORDINADOS_EMP son triviales. No se almacena ninguna otra MVD no trivial ni en PROYECTOS_EMP ni en SUBORDINADOS_EMP. Tampoco existe ninguna DF en estos esquemas de relación.

Para ilustrar la importancia de la 4FN, la Figura 11.5(a) muestra la relación EMP en el que un nuevo empleado, 'López', cuenta con tres subordinados ('Jesús', 'Dolores' y 'Roberto') y trabaja en cuatro proyectos diferentes ('W', 'X', 'Y' y 'Z'). Existen 16 tuplas EMP en la Figura 11.5(a). Si descomponemos la relación en PROYECTOS_EMP y SUBORDINADOS_EMP, tal y como puede verse en la Figura 11.5(b), sólo tenemos que almacenar un total de 11 tuplas en ambas relaciones. La descomposición, no sólo ahorra espacio de almacenamiento, sino que también evita las anomalías de actualización asociadas a las dependencias multivalor. Por ejemplo, si 'López' empieza a trabajar en un nuevo proyecto 'P', debemos insertar *tres* tuplas en EMP (una por cada subordinado). Si olvidamos hacerlo con alguno de ellos, la relación viola la MVD y se vuelve inconsistente porque implica de forma incorrecta una relación entre un proyecto y un subordinado.

Si la relación tiene MVDs no triviales, entonces las operaciones de inserción, borrado y actualización en tuplas únicas pueden provocar la modificación de otras tuplas. Si la actualización se realiza incorrectamente, el significado de la relación podría cambiar. Sin embargo, tras la normalización en la 4FN, estas anomalías desaparecen. Por ejemplo, para incorporar la información de que 'López' será asignado al proyecto 'P', sólo insertaremos una tupla en la relación 4FN PROYECTOS_EMP.

La relación EMP de la Figura 11.4(a) no está en 4FN porque representa a dos relaciones 1:N *independientes*: una entre los empleados y los proyectos en los que trabajan, y otra entre los empleados y sus subordinados. Hay veces en las que tenemos una relación entre tres entidades que depende de esas tres entidades, como ocurre con la relación SUMINISTRO de la Figura 11.4(c) (considere por ahora sólo las tuplas de la Figura 11.4[c] que están *por encima* de la línea de puntos). En este caso, una tupla representa a un proveedor que suministra una pieza específica a un *proyecto particular,* por lo que no existen MVDs no triviales. De este modo, la relación todo-clave SUMINISTRO ya está en 4FN y no debe descomponerse.

### 11.3.4 Descomposición de concatenación no aditiva en relaciones 4FN

Siempre que se descompone un esquema de relación $R$ en $R_1 = (X \cup Y)$ y $R_2 = (R - Y)$ basándonos en una MVD $X \twoheadrightarrow Y$ contenida en $R$, la descomposición tiene la propiedad de la concatenación no aditiva. Puede

**Figura 11.5.** Descomposición de un estado de la relación EMP que no está en 4FN. (a) Relación EMP con tuplas adicionales. (b) Las dos relaciones 4FN correspondientes, PROYECTOS_EMP y SUBORDINADOS_EMP.

**(a) EMP**

| NombreE | NombreProyecto | NombreSubordinado |
|---|---|---|
| Pérez | X | Juan |
| Pérez | Y | Ana |
| Pérez | X | Ana |
| Pérez | Y | Juan |
| López | W | Jesús |
| López | X | Jesús |
| López | Y | Jesús |
| López | Z | Jesús |
| López | W | Dolores |
| López | X | Dolores |
| López | Y | Dolores |
| López | Z | Dolores |
| López | W | Roberto |
| López | X | Roberto |
| López | Y | Roberto |
| López | Z | Roberto |

**(b) PROYECTOS_EMP**

| NombreE | NombreProyecto | NombreSubordinado |
|---|---|---|
| Pérez | X | Ana |
| Pérez | Y | Juan |
| López | W | Jesús |
| López | X | Jesús |
| López | Y | Jesús |
| López | Z | Jesús |

**SUBORDINADOS_EMP**

| NombreE | NombreSubordinado |
|---|---|
| Pérez | Ana |
| Pérez | Juan |
| López | Jesús |
| López | Jesús |
| López | Jesús |
| López | Jesús |

demostrarse que esto es una condición necesaria y suficiente para descomponer un esquema en otros dos que tienen la propiedad de concatenación no aditiva, como hace la propiedad NJB', que es una generalización superior de la propiedad NJB mostrada anteriormente. La propiedad NJB sólo trata con las DFs, mientras que la NJB' trata tanto con las DFs como con las MVDs (recuerde que una DF es también una MVD).

**Propiedad NJB'.** Los esquemas de relación $R_1$ y $R_2$ forman una descomposición de concatenación no aditiva de $R$ respecto a un conjunto $F$ de dependencias funcionales *y* multivalor si, y solo si,

$(R_1 \cap R_2) \twoheadrightarrow (R_1 - R_2)$

o, por simetría, si y sólo si,

$(R_1 \cap R_2) \twoheadrightarrow (R_2 - R_1)$.

Podemos usar una pequeña modificación del Algoritmo 11.3 para desarrollar el 11.5, el cual crea una descomposición de concatenación no aditiva en esquemas de relación que están en 4FN (en lugar de en BCNF). Al igual que ocurre con el Algoritmo 11.3, el 11.5 *no* produce necesariamente una descomposición que conserve las DF.

**Algoritmo 11.5.** Descomposición relacional en relaciones 4FN con la propiedad de concatenación no aditiva:

**Entrada:** Una relación universal $R$ y un conjunto de dependencias funcionales y multivalor $F$.

1. Establecer $D := \{ R \}$;
2. Mientras exista un esquema de relación $Q$ en $D$ que no esté en 4FN, hacer lo siguiente:

   { elegir un esquema de relación $Q$ en $D$ que no esté en 4FN;
   localizar una MVD no trivial $X \twoheadrightarrow Y$ en $Q$ que viole la 4FN;
   sustituir $Q$ en $D$ por dos esquemas de relación $(Q - Y)$ y $(X \cup Y)$;
   };

# 11.4 Dependencias de concatenación y quinta forma normal

Hemos visto que LJ1 y LJ1' dan a un esquema de relación $R$ la condición de estar descompuesto en otros dos esquemas $R_1$ y $R_2$, donde la descomposición tiene la propiedad de concatenación no aditiva. Sin embargo, en algunos casos, puede que no se produzca una descomposición de concatenación no aditiva de $R$ en *dos* esquemas de relación, sino en *más de dos*. Además, puede que no exista una dependencia funcional en $R$ que viole ninguna forma normal hasta la BCNF, y puede que no haya ninguna MVD no trivial en $R$ que viole la 4FN. En este caso, hemos de recurrir a otra dependencia llamada *dependencia de concatenación* y, en caso de existir, llevar a cabo una *descomposición multivía* en la quinta forma normal (5FN). Es importante indicar que una dependencia de este tipo es una restricción con una semántica muy peculiar que resulta muy difícil de detectar en la práctica; por consiguiente, la normalización a 5FN es muy rara verla en la práctica.

**Definición.** Una **JD** (**Dependencia de concatenación**, *Join Dependency*), expresada por JD($R_1$, $R_2$, . . . , $R_n$), especificada en un esquema de relación $R$, indica una restricción en los estados $r$ de $R$ que dice que cada estado legal $r$ de $R$ debe tener una descomposición de concatenación no aditiva en $R1$, $R2$, . . . , $Rn$; es decir, por cada $r$ tenemos:

$$* (\pi_{R1}(r), \pi_{R2}(r), ..., \pi_{Rn}(r)) = r$$

Observe que una MVD es un caso especial de JD donde $n = 2$. Esto es, una JD indicada como JD($R_1$, $R_2$) implica una MVD $(R_1 \cap R_2) \twoheadrightarrow (R_1 - R_2)$ (o, por simetría, $(R_1 \cap R_2) \twoheadrightarrow (R_2 - R_1)$). Una dependencia de concatenación JD($R_1$, $R_2$,..., $R_n$), especificada en un esquema de relación $R$, es una JD **trivial** si uno de los esquemas de relación $R_i$ de la JD($R_1$, $R_2$,..., $R_n$) es igual a $R$. Una dependencia de este tipo se dice que es trivial porque tiene la propiedad de concatenación no aditiva en cualquier estado de relación $r$ de $R$ y, por consiguiente, no especifica ninguna restricción en $R$. Ahora podemos definir la quinta forma normal, la cual recibe también el nombre de forma normal de proyección-concatenación.

**Definición.** Un esquema de relación $R$ está en **quinta forma normal** (**5FN**) (o en **PJNF** [**Forma normal de proyección-concatenación**, ***Project-Join Normal Form***]) respecto a un conjunto $F$ de dependencias funcionales, multivalor y de concatenación si, por cada dependencia de concatenación no aditiva JD($R_1$, $R_2$,..., $R_n$) en $F^+$ (es decir, implicada por $F$), cada $R_i$ es una superclave de $R$.

Para ver un ejemplo de JD, consideremos de nuevo la relación todo-clave SUMINISTRO de la Figura 11.4(c). Supongamos que siempre se cumple la siguiente restricción adicional: siempre que un proveedor $s$ suminis-

tra una pieza *p*, *y* un proyecto *j* utiliza la pieza *p*, *y* el proveedor *s* suministra *al menos una* pieza del proyecto *j*, *entonces* el proveedor *s* será también suministrador de *p* al proyecto *j*. Esta restricción puede replantearse de otras formas y especificar una dependencia de concatenación JD($R_1$, $R_2$, $R_3$) entre las tres proyecciones $R_1$(NombreS, NombrePieza), $R_2$(NombreS, NombreProy) y $R_3$(NombrePieza, NombreProy) de SUMINISTRO. Si esta restricción se cumple, las tuplas por encima de la línea de puntos de la Figura 11.4(c) deben existir en cualquier estado legal de la relación SUMINISTRO que también contenga las tuplas por encima de esa línea. La Figura 11.4(d) muestra cómo la relación SUMINISTRO *con la dependencia de concatenación* se descompone en tres relaciones $R_1$, $R_2$ y $R_3$ que están en 5FN. Observe que la aplicación de una CONCATENACIÓN NATURAL a *dos* relaciones *cualquiera produce tuplas falsas,* pero no así la misma operación sobre *las tres relaciones*. El lector deberá verificar este hecho en la relación de ejemplo de la Figura 11.4(c) y sus proyecciones de la Figura 11.4(d). Esto es así porque sólo existe la JD, pero no se han especificado las MVD. Observe también que la JD($R_1$, $R_2$, $R_3$) está indicada en *todos* los estados de relación legales, no sólo en los mostrados en la Figura 11.4(c).

Descubrir todas las JD en bases de datos reales con cientos de atributos es casi imposible. Esto sólo puede conseguirse con un alto grado de intuición sobre los datos por parte del diseñador.

# 11.5 Dependencias de inclusión

Las dependencias de inclusion fueron definidas para formalizar los dos tipos de restricciones interrelacionales:

- La *foreign key* (o restricción de la integridad referencial) no puede especificarse como una dependencia funcional o multivalor porque relaciona atributos entre relaciones.
- La restricción entre dos relaciones que representan una relación clase/subclase (consulte el Capítulo 4 y la Sección 7.2) tiene también una definición no formal en términos de dependencias funcionales, multivalor o de concatenación.

  **Definición.** Una **dependencia de inclusión** $R.X < S.Y$ entre dos conjuntos de atributos ($X$ del esquema de relación $R$ e $Y$ del esquema de relación *S)* especifica la restricción de que, en cualquier momento en que $r$ es un estado de relación de $R$ y $s$ lo es de $S$, debemos tener:

  $$\pi_X(r(R)) \subseteq \pi_Y(s(S))$$

La relación $\subseteq$ (subconjunto) no tiene que ser necesariamente un subconjunto. Obviamente, los conjuntos de atributos en los que se especifica la dependencia de inclusión ($X$ de $R$ e $Y$ de $S$) deben tener el mismo número de atributos. Además, los dominios de los pares de atributos correspondientes deben ser compatibles. Por ejemplo, si $X = \{A_1, A_2, ..., A_n\}$ e $Y = \{B_1, B_2, ..., B_n\}$, una posible correspondencia es contar con dom($A_i$) *compatible con* dom($B_i$) para $1 \leq i \leq n$. En este caso, decimos que $A_i$ **se corresponde con** $B_i$.

Por ejemplo, podemos especificar las siguientes dependencias de inclusión en el esquema relacional de la Figura 10.1:

DEPARTAMENTO.DniDirector < EMPLEADO.Dni

TRABAJA_EN.Dni < EMPLEADO.Dni

EMPLEADO.NúmeroDpto < DEPARTAMENTO.NúmeroDpto

PROYECTO.NumDptoProyecto < DEPARTAMENTO.NúmeroDpto

TRABAJA_EN.NumProyecto < PROYECTO.NumProyecto

LOCALIZACIONES_DPTO.NúmeroDpto < DEPARTAMENTO.NúmeroDpto

Todas las dependencias de inclusión anteriores representan **restricciones de integridad referencial**. También podemos usar dependencias de inclusión para representar **relaciones clase/subclase**. Por ejemplo, en el esquema relacional de la Figura 7.6, podemos especificar las siguientes dependencias de inclusión:

EMPLEADO.Dni < PERSONA.Dni

ALUMNO.Dni < PERSONA.Dni

ESTUDIANTE.Dni < PERSONA.Dni

Al igual que ocurre con otros tipos de dependencias, tenemos las IDIR (Reglas de inferencia de las dependencias de inclusión, *Inclusion Dependency Inference Rules*). Las siguientes son tres ejemplos de ellas:

IDIR1 (reflexividad): $R.X < R.X$.

IDIR2 (correspondencia de atributo): si $R.X < S.Y$, donde $X = \{A_1, A_2, ..., A_n\}$ e $Y = \{B_1, B_2, ..., B_n\}$ y $A_i$ *se corresponde con* $B_i$, entonces $R.A_i < S.B_i$ para $1 \leq i \leq n$.

IDIR3 (transitividad): si $R.X < S.Y$ y $S.Y < T.Z$, entonces $R.X < T.Z$.

Las reglas de inferencia anteriores evidenciaron que eran correctas y completas para las dependencias de inclusión. Hasta ahora, no se han desarrollado formas normales basadas en ellas.

# 11.6 Otras dependencias y formas normales

## 11.6.1 Dependencias de plantilla

Las dependencias de plantilla ofrecen una técnica de representación de restricciones en las relaciones que, normalmente, no tienen definiciones formales ni sencillas. Sin importar cuántos tipos de dependencias desarrollemos, siempre surge alguna restricción peculiar basada en la semántica de los atributos de las relaciones que no puede ser representada por ninguna de ellas. La idea que se esconde tras las dependencias de plantilla es especificar una plantilla (o ejemplo) que defina cada restricción o dependencia.

Existen dos tipos de plantilla: la de generación de tuplas y la de generación de restricciones. Una plantilla consta de un número de **tuplas de hipótesis** que están pensadas para mostrar un ejemplo de los tipos de tuplas que pueden aparecer en una o más relaciones. La otra parte de la plantilla es la **conclusión**. En las plantillas de generación de tuplas, la conclusión es un *conjunto de tuplas* que deben existir también en las relaciones en las que estén presentes las tuplas de hipótesis. Para las plantillas de generación de restricciones, la conclusión de la plantilla es una *condición* que debe cumplirse en las tuplas de hipótesis.

La Figura 11.6 muestra cómo definir como plantillas las dependencias funcionales, multivalor y de inclusión. La Figura 11.7 indica la forma de especificar la restricción de que *el salario de un empleado no puede ser superior al de su supervisor directo* en el esquema de relación EMPLEADO de la Figura 5.5.

## 11.6.2 Dependencias funcionales basadas en funciones aritméticas y procedimientos

A veces, los atributos de una relación pueden estar relacionados mediante alguna función aritmética o alguna otra relación funcional más compleja. Siempre y cuando un valor único de $Y$ esté asociado con cada $X$, podemos considerar que existe la DF $X \rightarrow Y$. Por ejemplo, en la relación:

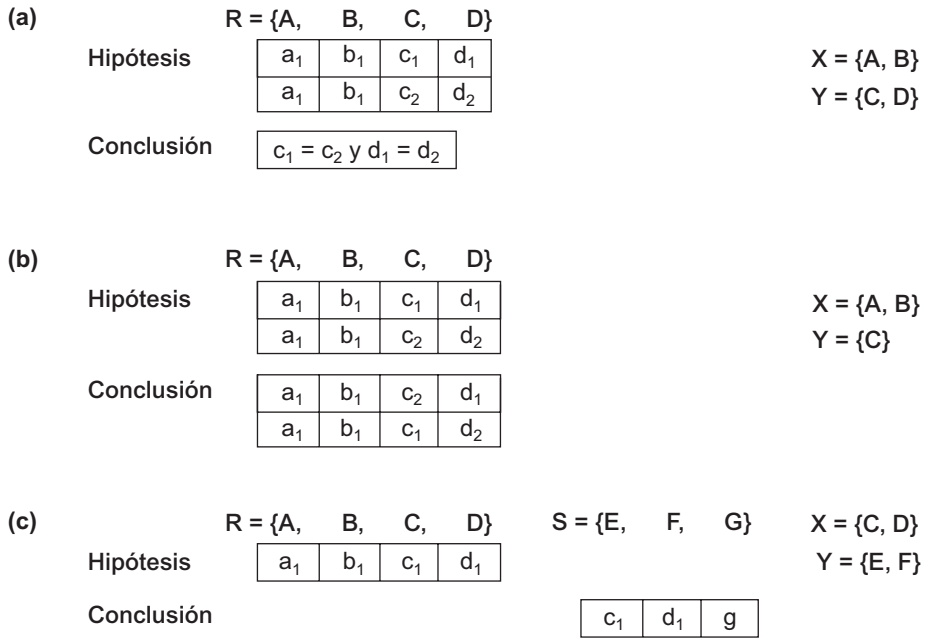
LÍNEA_PEDIDO(NúmeroPedido, NúmeroObjeto, Cantidad, PrecioUnitario, PrecioTotal, PrecioDescuento)

cada tupla representa un elemento con una cantidad y el precio por unidad de ese elemento. En esta relación, (Cantidad, PrecioUnitario) →PrecioTotal según la fórmula

PrecioTotal = PrecioUnitario * Cantidad.

Por tanto, existe un único valor PrecioTotal para cada pareja (Cantidad, PrecioUnitario), y esto se adapta a la definición de dependencia funcional.

**Figura 11.6.** Plantilla para algunos tipos de dependencias comunes.
(a) Plantilla para la dependencia funcional X→ Y.
(b) Plantilla para la dependencia multivalor X →→Y.
(c) Plantilla para la dependencia de inclusión R.X < S.Y.




Además, puede existir un procedimiento que tenga en cuenta los descuentos, el tipo de elemento, etcétera, y que procese un precio con descuento a partir de la cantidad total solicitada para ese elemento. Por consiguiente, podemos decir que:

(NúmeroObjeto, Cantidad, PrecioUnitario) →PrecioDescuento, o bien,
(NúmeroObjeto, Cantidad, PrecioTotal) →PrecioDescuento.

Para verificar la DF anterior, tenemos que hacer entrar en juego un procedimiento más complejo llamado CALCULAR_PRECIO_TOTAL. Aunque las DF anteriores están presentes en la mayoría de relaciones, no se les presta una atención especial durante la normalización.

## 11.6.3 Forma normal de dominio clave

No existe una regla estricta y rápida para definir formas normales sólo hasta la 5FN. Históricamente, el proceso de normalización y el de descubrimiento de dependencias no deseadas fueron llevados a cabo a través de la 5FN, pero ha sido posible definir formas normales más estrictas que tienen en cuenta tipos adicionales de dependencias y restricciones. La idea que se esconde tras la **DKFN** (**Forma normal de dominio clave**, ***Domain-Key Normal Form***) es la de especificar (al menos, teóricamente) la *última forma normal* que tiene en cuenta todos los posibles tipos de dependencias y restricciones. Un esquema de relación se dice que está en **DKNF** si todas las restricciones y dependencias que deben persistir en los estados de relación válidos pueden cumplirse simplemente haciendo cumplir las restricciones clave y de dominio de la relación. Para una relación en DKNF, le es muy sencillo cumplir todas las restricciones de la base de datos comprobando simplemente que cada atributo de una tupla está en el dominio apropiado y que se cumple cada restricción clave.

Sin embargo, debido a la dificultad para incluir restricciones complejas en una relación DKNF, su utilidad práctica está limitada, ya que puede resultar muy complicado especificar restricciones de integridad genera-

**Figura 11.7.** Plantillas para la restricción de que el salario de un empleado debe ser menor que el de su supervisor.

**EMPLEADO = {Nombre, Dni, . . . , Sueldo, DniSupervisor**

| | a | b | c | d |
|---|---|---|---|---|
| **Hipótesis** | e | d | f | g |
| **Conclusión** | | | c < f | |

les. Por ejemplo, consideremos las relaciones COCHE(Marca, Identificador) (donde Identificador es el número de identificación del vehículo) y FABRICANTE(Identificador, País). Una restricción general podría tener la siguiente forma: *si la Marca es 'Toyota' o 'Lexus', el primer carácter de* Identificador *es una 'J' si el país de fabricación es 'Japón'; si la Marca es 'Honda' o 'Acura', el segundo carácter de* Identificador *es una 'J' si el país de fabricación es 'Japón'.* No existe una forma simplificada para representar esta restricción a no ser con un procedimiento (o una afirmación general). El procedimiento CALCULAR_PRECIO_TOTAL anterior es un ejemplo de procedimiento necesario para forzar restricciones de integridad de este tipo.

# 11.7 Resumen

En este capítulo hemos visto varios algoritmos de normalización. Los a*lgoritmos de síntesis relacional* (como el 11.2 y el 11.4) crean relaciones 3FN a partir de un esquema de relación universal basado en un conjunto de dependencias funcionales dado especificado por el diseñador de la base de datos. Los algoritmos de descomposición relacional (como el 11.3 y el 11.5) crean relaciones BCNF (o 4FN) mediante una descomposición no aditiva sucesiva de relaciones no normalizadas en dos relaciones. Primero estudiamos dos propiedades importantes de las descomposiciones: la de concatenación no aditiva y la de conservación de las dependencias. Hemos mostrado un algoritmo para la descomposición no aditiva (Algoritmo 11.1) y hemos hablado de las descomposiciones binarias (NJB). Vimos que es posible sintetizar esquemas de relación 3FN que cumplen las dos propiedades anteriores; sin embargo, en el caso de la BCNF, es posible tener como única meta la no aditividad de las concatenaciones: la conservación de la dependencia *puede no* estar necesariamente garantizada. En este caso, la condición de concatenación no aditiva es un objetivo prioritario.

A continuación, definimos otros tipos de dependencias y formas normales. Las dependencias multivalor, que provienen de una combinación incorrecta de dos o más atributos multivalor independientes en la misma relación y que producen una expansión combinatoria de las tuplas, se utilizan para definir la cuarta forma normal (4FN). Las dependencias de concatenación, que indican una descomposición sin pérdida multivía de una relación, inducen la definición de la quinta forma normal (5FN), que también se conoce como PJNF. Vimos también las dependencias de inclusión, que se utilizan para especificar restricciones de integridad referencial y de clase/subclase, y las dependencias de plantilla, que pueden utilizarse para especificar tipos de restricciones arbitrarias. Para forzar ciertas restricciones de una dependencia funcional, apuntamos la necesidad de contar con funciones aritméticas o con procedimientos más complejos. Concluimos con un breve comentario acerca de la DKNF.

## Preguntas de repaso

**11.1.** ¿Cuál es el significado de la condición de conservación de los atributos en una descomposición?

**11.2.** ¿Por qué son insuficientes las formas normales como condición para un buen diseño de un esquema?

**11.3.** ¿Qué es la propiedad de conservación de la dependencia de una descomposición? ¿Por qué es importante?

**11.4.** ¿Por qué no podemos garantizar la producción de esquemas de relación BCNF a partir de descomposiciones de dependencia conservada de esquemas de relación que no sean BCNF? Ofrezca un contraejemplo que ilustre este punto.

**11.5.** ¿Qué es la propiedad de concatenación sin pérdida (o no aditiva)? ¿Por qué es importante?

**11.6.** Entre las propiedades de conservación de la dependencia y pérdida, ¿cuál debe satisfacerse definitivamente? ¿Por qué?

**11.7.** Comente los problemas relacionados con el valor NULL y las tuplas colgantes.

**11.8.** ¿Qué es una dependencia multivalor? ¿Qué tipo de restricción especifica? ¿Cuándo se origina?

**11.9.** Ilustre cómo el proceso de creación de relaciones en la primera forma normal puede inducir dependencias multivalor. ¿Cómo debe realizarse adecuadamente la primera normalización para evitar las MVD?

**11.10.** Defina la cuarta forma normal. ¿Cuándo se viola? ¿Por qué es útil?

**11.11.** Defina las dependencias de concatenación y la quinta forma normal. ¿Por qué se conoce también a la 5FN como PJNF?

**11.12.** ¿Qué tipo de restricción pretende representar las dependencias de inclusión?

**11.13.** ¿En qué difieren las dependencias de plantilla de los otros tipos de dependencias que hemos estudiado?

**11.14.** ¿Por qué se dice que la DKNF es la última forma normal?

## Ejercicios

**11.15.** Demuestre que los esquemas de relación generados por el Algoritmo 11.2 están en 3FN.

**11.16.** Demuestre que, si la matriz $S$ resultante del Algoritmo 11.1 no tiene una fila en la que todos sus símbolos son $a$, proyectando $S$ en la descomposición y concatenándola de vuelta siempre producirá, al menos, una tupla falsa.

**11.17.** Demuestre que los esquemas de relación generados por el Algoritmo 11.3 están en BCNF.

**11.18.** Demuestre que los esquemas de relación generados por el Algoritmo 11.4 están en BCNF.

**11.19.** Especifique una dependencia de plantilla para las dependencias de concatenación.

**11.20.** Especifique todas las dependencias de inclusión del esquema relacional de la Figura 5.5.

**11.21.** Pruebe que una dependencia funcional satisface la definición formal de una dependencia multivalor.

**11.22.** Considere el ejemplo de normalización de la relación PARCELAS de la Sección 10.4. Determine si su descomposición en {PARCELAS1AX, PARCELAS1AY, PARCELAS1B, PARCELAS2} tiene la propiedad de concatenación sin pérdida, aplicando el Algoritmo 11.1, y también usando la prueba bajo la Propiedad NJB.

**11.23.** Demuestre cómo las MVD NombreE $\twoheadrightarrow$ NombreProyecto y NombreE $\twoheadrightarrow$ NombreSubordinado de la Figura 11.4(a) pueden producirse durante la normalización a 1NF de una relación, donde los atributos NombreProyecto y NombreSubordinado son multivalor.

**11.24.** Aplique el Algoritmo 11.4(a) a la relación del Ejercicio 10.26 para determinar una clave para $R$. Cree un conjunto mínimo de dependencias $G$ que sea equivalente a $F$, y aplique el algoritmo de síntesis (Algoritmo 11.4) para descomponer $R$ en relaciones 3FN.

**11.25.** Repita el Ejercicio 11.24 para las dependencias funcionales del Ejercicio 10.27.

**11.26.** Aplique el algoritmo de descomposición (Algoritmo 11.3) a la relación $R$ y el conjunto de dependencias $F$ del Ejercicio 10.26. Repítalo para las dependencias $G$ del Ejercicio 10.27.

**11.27.** Aplique el Algoritmo 11.4(a) a las relaciones de los Ejercicios 10.29 y 10.30 para determinar una clave de $R$. Aplique el algoritmo de síntesis (Algoritmo 11.4) para descomponer $R$ en relaciones 3FN, y el de descomposición (Algoritmo 11.3) para descomponer $R$ en relaciones BCNF.

**11.28.** Escriba programas que implementen los Algoritmos 11.3 y 11.4.

**11.29.** Considere las siguientes descomposiciones para el esquema de relación $R$ del Ejercicio 10.26. Determine si cada descomposición tiene (1) la propiedad de conservación de dependencia y (2) la propiedad de concatenación sin pérdida, respecto a $F$. Determine también en qué forma normal está cada relación de la descomposición.

**a.** $D_1 = \{R_1, R_2, R_3, R_4, R_5\}$ ; $R_1 = \{A, B, C\}$ , $R_2 = \{A, D, E\}$ , $R_3 = \{B, F\}$ , $R_4 = \{F, G, H\}$ , $R_5 = \{D, I, J\}$

**b.** $D_2 = \{R_1, R_2, R_3\}$ ; $R_1 = \{A, B, C, D, E\}$ , $R_2 = \{B, F, G, H\}$ , $R_3 = \{D, I, J\}$

**c.** $D_3 = \{R_1, R_2, R_3, R_4, R_5\}$ ; $R_1 = \{A, B, C, D\}$ , $R_2 = \{D, E\}$ , $R_3 = \{B, F\}$ , $R_4 = \{F, G, H\}$ , $R_5 = \{D, I, J\}$

**11.30.** Considere la relación FRIGORÍFICO(NúmeroModelo, Año, Precio, PlantaFabricación, Color), que de forma abreviada es FRIGORÍFICO (M, Y, P, MP, C), y el siguiente conjunto $F$ de dependencias funcionales: $F = \{\mathsf{M} \rightarrow \mathsf{MP}, \{\mathsf{M}, \mathsf{Y}\} \rightarrow \mathsf{P}, \mathsf{MP} \rightarrow \mathsf{C}\}$

**a.** Evalúe cada una de las siguientes como una clave candidata de FRIGORÍFICO, dando razones de si puede ser o no una clave: {M} , {M, Y} , {M, C}.

**b.** Basándonos en la determinación de claves anterior, indique si la relación FRIGORÍFICO está en 3FN y en BCNF, dando las razones apropiadas.

**c.** Considere la descomposición de FRIGORÍFICO en $D = \{R_1(\mathsf{M}, \mathsf{Y}, \mathsf{P}), R_2(\mathsf{M}, \mathsf{MP}, \mathsf{C})\}$. ¿Es esta descomposición sin pérdida? Demuéstrelo (puede consultar la comprobación que se encuentra bajo la propiedad LJ1 de la Sección 11.1.4).

**11.31.** Considere la relación LIBRO(TítuloLibro, Autor, Edición, Año) que tiene los siguientes datos:

| TítuloLibro | Autor | Edición | Año |
|---|---|---|---|
| Fundamentos de bases de datos | Navathe | 3 | 2000 |
| Fundamentos de bases de datos | Elmasri | 3 | 2000 |
| Fundamentos de bases de datos | Elmasri | 4 | 2004 |
| Fundamentos de bases de datos | Navathe | 4 | 2004 |

**a.** Basándonos en el sentido común, ¿cuáles son las posibles claves candidatas de esta relación?

**b.** ¿Tienen los datos de la tabla una o más dependencias funcionales? (No liste las DFs aplicando las reglas de derivación.) En caso afirmativo, ¿cuáles son? Especifique cómo eliminaría las dependencias por descomposición.

**c.** ¿Tiene la relación resultante una MVD? En caso afirmativo, ¿qué es?

**d.** ¿Qué aspecto tendrá la descomposición final?

**11.32.** Considere la siguiente relación:

VIAJE(IdViaje, FechaInicio, CiudadesVisitadas, TarjetasUsadas)

Esta relación hace referencia a los viajes de negocio efectuados por los comerciales de una compañía. Suponga que VIAJE tiene una única FechaInicio, pero que implica muchas Ciudades y los comerciales pueden utilizar varias tarjetas de crédito en el viaje. Realice una simulación para la carga de la tabla.

**a.** Comente qué DFs y/o MVDs existen en esta relación.

**b.** Demuestre cómo la normalizaría.

## Ejercicios de práctica

*Nota*. Estos ejercicios utilizan el sistema DBD (Diseñador de base de datos) descrito en el manual de laboratorio. El esquema relacional $R$ y el conjunto de dependencias funcionales $F$ tienen que codificarse como listas. Como ejemplo, $R$ y $F$ están codificados del siguiente modo en el problema 10.26:

$R = [a, b, c, d, e, f, g, h, i, j]$
$F = [[[a, b],[c]],$
$[[a],[d, e]],$
$[[b],[f]],$
$[[f],[g, h]],$
$[[d],[i, j]]]$

Ya que el DBD está implementado en Prolog, el uso de términos en mayúsculas está reservado a las variables del lenguaje y, por consiguiente, se utilizan las constantes en minúsculas para codificar los atributos. Para obtener más información acerca del uso del sistema DBD, por favor consulte el manual del laboratorio.

**11.33.** Usando el sistema DBD, verifique sus respuestas a los siguientes ejercicios:

- **a.** 11.22
- **b.** 11.24
- **c.** 11.25
- **d.** 11.26
- **e.** 11.27
- **f.** 11.29 (a) y (b)
- **g.** 11.30 (a) y (c)

## Bibliografía seleccionada

Los libros de Maier (1983) y de Atzeni y De Antonellis (1992) incluyen un tratamiento global de la teoría de la dependencia relacional. El algoritmo de descomposición (Algoritmo 11.3) se debe a Bernstein (1976). El Algoritmo 11.4 está basado en la algoritmo de normalización presentado en Biskup y otros (1979). Tsou y Fischer (1982) aportan un algoritmo polinómico de tiempo para la descomposición BCNF.

La teoría de la conservación de las dependencias y las concatenaciones sin pérdida aparecieron en Ullman (1988), donde aparecen algunas de las comprobaciones de los algoritmos aquí comentados. La propiedad de concatenación sin pérdida se analiza en Aho y otros (1979). Los algoritmos para determinar las claves de una relación desde las dependencias funcionales aparecen en Osborn (1976); la verificación de la BCNF se comenta en Osborn (1979), mientras que la de la 3FN aparece en Tsou y Fischer (1982). Los algoritmos para el diseño de relaciones BCNF se brindan en Wang (1990) y en Hernández y Chan (1991).

Las dependencias multivalor y la cuarta forma normal están definidas en Zaniolo (1976) y Nicolas (1978). Muchas de las formas normales avanzadas son debidas a Fagin: la cuarta forma normal en Fagin (1977), la PJNF en Fagin (1979) y la DKNF en Fagin (1981). El conjunto completo de reglas para las dependencias funcionales y multivalor aparecen en Beeri y otros (1977). Las dependencias de concatenación se comentan en Rissanen (1977) y en Aho y otros (1979). Las reglas de inferencia para las dependencias de concatenación las facilita Sciore (1982). Las dependencias de inclusión son comentadas por Casanova y otros (1981), y analizadas en Cosmadakis y otros (1990). Su uso para la optimización de esquemas relacionales se trata en Casanova y otros (1989). Las dependencias de plantilla se tratan en Sadri y Ullman (1982). El resto de dependencias se comentan en Nicolas (1978), Furtado (1978) y Mendelzon y Maier (1979). Abiteboul y otros (1995) proporciona un tratamiento teórico de muchas de las ideas presentadas en este capítulo y en el Capítulo 10.

CAPÍTULO 12

# Metodología práctica de diseño de bases de datos y uso de los diagramas UML

En este Capítulo vamos a poner en práctica los conocimientos teóricos adquiridos hasta ahora en el diseño de una base de datos. En varios capítulos anteriores hemos descrito material relevante para el diseño de bases de datos actuales en aplicaciones comerciales reales. Este material incluye los Capítulos 3 y 4, con el modelado conceptual de una base de datos; los Capítulos del 5 al 9, con el modelo relacional, el lenguaje SQL, el álgebra relacional y de cálculo, el mapeo a alto nivel de un esquema conceptual ER o EER a un esquema relacional y la programación en sistemas relaciones (RDBMS); y los Capítulos 10 y 11 con la teoría de la dependencia de datos y los algoritmos de normalización relacional.

La actividad general de diseño de una base de datos suele sufrir un proceso sistemático llamado **metodología de diseño**, tanto si la base de datos objetivo está administrada por un RDBMS, como por un ODBMS (Sistema gestor de bases de datos orientadas a objetos, *Object DataBase Management System*) o un ORDBMS (Sistema de bases de datos relacional con objetos, *Object Relational DataBase Management System*). Varias son las metodologías de diseño implícitas en las herramientas de diseño de bases de datos ofrecidas en la actualidad por los distintos fabricantes. Entre las más populares podemos citar Designer 2000 de Oracle; ERWin, BPWin y AllFusion Component Modeler de Computer Associates; Sybase Enterprise Application Studio; ER Studio de Embarcadero Technologies y Telelogic System Architect de Popkin Software. Nuestro objetivo en este capítulo no es tratar sólo una metodología específica, sino realizar el diseño de una base de datos en un contexto más amplio, tal y como se acomete en organizaciones grandes en las que se da servicio a cientos o miles de usuarios.

En general, el diseño de bases de datos de pequeño tamaño (de hasta unos 20 usuarios) no es demasiado complicado. Pero para entornos de mediano o gran tamaño que sirven varios grupos de aplicaciones, cada uno de ellos con cientos de usuarios, se hace necesario un acercamiento sistemático al diseño global de la base de datos. El tamaño total de una base de datos con información no refleja la complejidad del diseño; es el esquema el que lo hace. Cualquier base de datos con un esquema que incluya más de 30 ó 40 tipos de entidades y un número similar de tipos de relaciones necesita de una metodología de diseño muy cuidadosa.

Usando el término **base de datos grande** para aquellas estructuras con varios cientos de gigabytes de datos y un esquema con más de 30 ó 40 tipos de entidad distintos, podemos abarcar un amplio abanico de bases de datos, incluyendo las gubernamentales, las industriales y las de instituciones financieras y comerciales. Las industrias del sector servicios (por ejemplo, bancos, hoteles, líneas aéreas, compañías aseguradoras y de comunicaciones), utilizan bases de datos para sus operaciones diarias 24 horas al día, 7 días a la semana (lo que se conoce en la industria como operaciones *24 por 7*). Las aplicaciones para estas bases de datos reciben el nombre de *sistemas de procesamiento de transacciones* debido a los grandes volúmenes de transacciones y

evaluaciones que precisan. En este capítulo nos concentraremos en el diseño de una base de datos para empresas de mediano y gran tamaño, donde predomina el procesamiento de transacciones.

Este capítulo tiene una gran variedad de objetivos. La Sección 12.1 se centra en el ciclo vital del sistema de información dentro de las empresas, haciendo especial hincapié en el sistema de bases de datos. La Sección 12.2 destaca las fases de la metodología de diseño de una base de datos en el contexto administrativo. La Sección 12.3 habla de los diagramas UML y ofrece detalles sobre las notaciones de algunos de ellos que son especialmente útiles a la hora de efectuar la toma de requisitos y llevar a cabo el diseño conceptual y lógico de las bases de datos. Se presenta un ejemplo parcial del diseño de una base de datos universitaria. La Sección 12.4 presenta la popular herramienta software de desarrollo Rational Rose, que utiliza diagramas UML como su especificación técnica principal, destacándose sus características específicas para el diseño del esquema y el modelado de la base de datos. La Sección 12.5 hace un breve comentario sobre las herramientas de diseño automatizadas de bases de datos.

# 12.1 El papel de los sistemas de información en las empresas

## 12.1.1 Uso de sistemas de bases de datos en el contexto administrativo

Los sistemas de bases de datos han empezado a formar parte de los sistemas de información de muchas empresas. A principio de los años 60, estos sistemas estaban dominados por los sistemas de ficheros, pero desde comienzos de la década de los 70, las empresas empezaron a migrar a los sistemas de bases de datos. Para acomodar estos sistemas, muchas empresas crearon el puesto de DBA (Administrador de base de datos) o, incluso, departamentos completos de administración para supervisar y controlar las actividades del ciclo vital de la base de datos. De forma análoga, la IT (Tecnologías de la información, *Information Technology*) y la IRM (Administración de recursos de información, *Information Resource Management*) han sido reconocidas por muchas organizaciones como la clave para el éxito de muchos negocios por los siguientes motivos:

- Los datos son considerados como un recurso corporativo, y su administración y control se realiza de una forma centralizada para que la empresa trabaje de forma efectiva.
- Muchas funciones en las empresas están informatizadas, lo que incrementa la necesidad de que grandes volúmenes de datos estén disponibles “al minuto”.
- A medida que crece la complejidad de los datos y las aplicaciones, la complejidad de las relaciones entre los datos debe ser modelada y mantenida.
- En muchas empresas existe una tendencia hacia la consolidación de los recursos de información.
- Muchas organizaciones están reduciendo sus costes de personal al permitir que el usuario final pueda efectuar transacciones comerciales. Esto es algo evidente en agencias de viajes, servicios financieros y en la distribución *online* al por menor de bienes, así como en sistemas de comercio electrónico del tipo cliente-a-empresa, como Amazon.com y EBay. En estas situaciones, debe diseñarse una base de datos accesible y actualizable públicamente y puesta a disposición de las transacciones.

Los sistemas de bases de datos satisfacen con creces los requisitos anteriores. En este entorno también están disponibles otras dos características de los sistemas de bases de datos:

- ***Independencia de los datos*** para proteger las aplicaciones de los cambios en la organización lógica subyacente, en las rutas de acceso físicas y en las estructuras de almacenamiento.
- ***Esquemas externos*** (vistas) para permitir que los mismos datos sean utilizados por varias aplicaciones, teniendo cada una de ellas su propia vista de la información.

Las nuevas capacidades proporcionadas por los sistemas de bases de datos y las siguientes características clave han hecho de ellos componentes integrales de cualquier sistema de información informatizado:

- Integración de los datos de múltiples aplicaciones en una única base de datos.
- Simplicidad a la hora de desarrollar nuevas aplicaciones usando lenguajes de alto nivel como SQL.
- Posibilidad de soporte para los accesos casuales de los administradores para hacer consultas, a la vez que se ofrece soporte para el procesamiento de transacciones a nivel de producción.

Desde comienzos de los 70 hasta mediados de los 80, el movimiento que se produjo fue hacia la creación de grandes almacenes de datos centralizados administrados por un único DBMS central. En los últimos 10 ó 15 años, esta tendencia se ha revertido debido a los siguientes desarrollos:

1. Los computadores personales y productos software parecidos a bases de datos, como Excel, Visual FoxPro y Access (Microsoft), o SQL Anywhere (Sybase), y otros productos de dominio público como MySQL y PostgreSQL, están siendo ampliamente utilizados por los usuarios que anteriormente pertenecían a la categoría de usuarios ocasionales de bases de datos. Muchos administradores, secretarias, ingenieros, científicos y arquitectos pertenecen a esta categoría. Como resultado de ello, la práctica de crear **bases de datos personales** ha ganado en popularidad. Ahora es posible consultar la copia de parte de una gran base desde un *mainframe* o un servidor de bases de datos, trabajar en ella desde un computador personal, y después restaurarla en el *mainframe*. De forma similar, los usuarios pueden diseñar y crear sus propias bases de datos y fundirlas después en otra más grande.
2. La aparición de los DBMS cliente-servidor distribuidos (consulte el Capítulo 25) inauguró la posibilidad de distribuir la base de datos en varios sistemas informáticos para un mejor control local y un procesamiento local más rápido. Al mismo tiempo, los usuarios locales pueden acceder a datos remotos usando las facilidades ofrecidas por el DBMS, o a través de la Web. Se han usado herramientas de desarrollo como PowerBuilder (Sybase) o Developer 2000 (Oracle) con opciones predefinidas para enlazar aplicaciones a varios servidores *back-end* de bases de datos.
3. Muchas empresas utilizan ahora **sistemas de diccionarios de datos** o **almacenes de información**, que son mini-DBMS que gestionan **metadatos**, es decir, datos que describen la estructura de la base de datos, sus restricciones, aplicaciones, autorizaciones, etc. Se utilizan con frecuencia como una herramienta integral para la administración de recursos de información. Un sistema de diccionario de datos útil debe almacenar y administrar los siguientes tipos de información sobre la base de datos:
   a. Descripciones de sus esquemas.
   b. Información detallada de su diseño físico, como estructuras de almacenamiento, rutas de acceso y tamaños de los ficheros y registros.
   c. Descripciones de sus usuarios, sus responsabilidades y permisos de acceso.
   d. Descripciones de alto nivel de sus transacciones y aplicaciones, así como de las relaciones entre los usuarios y dichas transacciones.
   e. La relación entre las transacciones y los datos referenciados por ellas. Esto resulta útil a la hora de determinar qué transacciones se ven afectadas cuando cambian las definiciones de ciertos datos.
   f. Estadísticas de uso, como las frecuencias de las consultas y las transacciones, y el número de accesos a diferentes partes de la base de datos.

Estos metadatos están disponibles para los DBA, los diseñadores y los usuarios autorizados como un sistema de documentación *online*. De este modo se mejora el control que tienen los DBA sobre el sistema de información, así como el entendimiento y uso del sistema por parte de los usuarios. La llegada de la tecnología de almacenamiento de datos ha acentuado la importancia de los metadatos.

Cuando se diseñan **sistemas de procesamiento de transacciones** de alto rendimiento, que precisan de una operativa sin interrupciones durante todo el día, el rendimiento se convierte en un factor crítico. A estas bases

de datos suelen llegar cientos de transacciones por minuto desde terminales remotos y locales. El rendimiento de la transacción, en términos de promedio de transacciones por minuto y de tiempo de respuesta medio y máximo de una de ellas, es crítico. Un cuidado diseño físico de la base de datos, que cumpla las necesidades de procesamiento de transacciones de la empresa, es un requisito imprescindible en estos sistemas.

Algunas empresas han comprometido la administración de sus recursos de información a ciertos DBMS y diccionarios de datos comerciales. Su inversión en el diseño y la implementación de grandes y complejos sistemas complica el cambio a versiones más modernas de esos productos, de modo que esas empresas quedan atrapadas por sus actuales DBMS. En consideración a estas bases de datos grandes y complejas, no vamos a insistir demasiado en la importancia de un diseño cuidado que tenga en cuenta las posibles modificaciones del sistema (la puesta a punto, o refinamiento). Trataremos el refinamiento, junto con la optimización de las consultas, en el Capítulo 16. Si un sistema grande y complejo no puede evolucionar, el coste de cambiar a otro DBMS puede ser bastante elevado.

### 12.1.2 El ciclo vital del sistema de información

En empresas grandes, el sistema de bases de datos suele formar parte del **sistema de información**, el cual incluye todos los recursos implicados en la recopilación, administración, uso y difusión de los recursos de información de la empresa. En un entorno informatizado, estos recursos incluyen los propios datos, el software DBMS, el hardware del sistema y los dispositivos de almacenamiento, el personal que usa y administra los datos (DBA, usuarios finales, etc.), las aplicaciones que acceden y actualizan los datos, y los programadores que desarrollan estas aplicaciones. Un sistema de bases de datos como éste forma parte de un sistema de información centralizado mucho mayor.

En esta sección vamos a examinar el ciclo vital típico de un sistema de información, y cómo se acomoda en él un sistema de base de datos. El ciclo vital de un sistema de información suele recibir el nombre de **ciclo vital principal** ***(macro life cycle***), mientras que el de la base de datos se conoce como **ciclo vital secundario** (***micro life cycle***). La distinción entre ellos es más difusa en aquellos sistemas de información en los que las bases de datos son uno de los componentes integrales principales. El ciclo vital principal típico incluye las siguientes fases:

1. **Análisis de viabilidad.** Esta fase está relacionada con el análisis de las áreas potenciales de la aplicación, la identificación de los costes de obtención y diseminación de información, la realización de los estudios preliminares sobre costes-beneficios, la determinación de la complejidad de los datos y los procesos, y el establecimiento de las prioridades entre las aplicaciones.
2. **Recopilación de requisitos y análisis.** Se recopilan los requisitos de forma detallada interactuando con los usuarios potenciales y los grupos de usuarios, a fin de identificar sus problemas y necesidades particulares. Se identifican también las dependencias entre las aplicaciones, la comunicación y los procedimientos de comunicación.
3. **Diseño.** Esta fase tiene dos aspectos: el diseño del sistema de bases de datos y el diseño de los sistemas de aplicaciones (programas) que usan y procesan la información.
4. **Implementación.** Se implementa el sistema de información, se carga la base de datos, y se implantan y prueban las transacciones.
5. **Validación y pruebas de aceptación.** Se valida el sistema en términos de rendimiento según los requisitos de las reuniones con los usuarios. El sistema se verifica contra criterios de rendimiento y especificaciones de comportamiento.
6. **Implantación, operativa y mantenimiento.** Esta fase incluye la conversión de los usuarios desde los sistemas antiguos y su formación en los nuevos. La fase operativa empieza cuando las funciones de todos los sistemas están operativas y se han validado. A medida que aparecen nuevos requisitos o aplicaciones, todos ellos deben pasar por todas las fases previas hasta que son validados e incorporados al

sistema. La monitorización del rendimiento y el mantenimiento del sistema son actividades importantes durante esta fase.

### 12.1.3 El ciclo vital del sistema de aplicaciones de bases de datos

Las siguientes son las actividades relacionadas con el ciclo vital (secundario) del sistema de aplicaciones de bases de datos:

1. **Definición del sistema**. Se definen el ámbito del sistema de la base de datos, sus usuarios y sus aplicaciones. Es preciso identificar las interfaces de las distintas categorías de usuarios, las restricciones del tiempo de respuesta y las necesidades de procesamiento y almacenamiento.
2. **Diseño de la base de datos.** Al final de esta fase está preparado un completo diseño lógico y físico de la base de datos en el DBMS elegido.
3. **Implementación de la base de datos**. Esta fase incluye la especificación de las definiciones interna, externa y conceptual de la base de datos, la creación de ficheros vacíos y la implementación de las aplicaciones.
4. **Carga o conversión de datos**. La base de datos se rellena, bien mediante una carga directa de datos, bien convirtiendo los ficheros existentes al nuevo formato del sistema.
5. **Conversión de la aplicación**. Cualquier aplicación del sistema anterior se convierte al nuevo sistema.
6. **Verificación y validación**. El nuevo sistema se verifica y valida. En los programas, este proceso puede ser muy complicado, y las técnicas empleadas suelen tratarse en los cursos de ingeniería de software. Existen herramientas automatizadas que ayudan en el proceso, pero su tratamiento está fuera de los objetivos de este libro.
7. **Operativa**. La base de datos y sus aplicaciones se ponen en marcha. Por lo general, el sistema antiguo y el nuevo trabajan en paralelo durante un tiempo.
8. **Monitorización y mantenimiento**. Durante la fase operativa, la monitorización y el mantenimiento son constantes. Tanto en los datos como en las aplicaciones puede haber un crecimiento y una expansión. De vez en cuando pueden ser necesarias modificaciones y reorganizaciones.

Las Actividades 2, 3 y 4 forman parte de las fases de diseño e implementación del ciclo vital del sistema de información más grande. En la Sección 12.2 centramos más nuestra atención en las Actividades 2 y 3, que están dedicadas a las fases de diseño e implementación de la base de datos. La mayoría de las bases de datos de las empresas experimentan todas las actividades de ciclo vital precedentes. Las actividades de conversión (4 y 5) no son aplicables cuando tanto la base de datos como las aplicaciones son nuevas. Cuando una empresa migra de un sistema establecido a otro nuevo, las Actividades 4 y 5 suelen ser las que más tiempo y esfuerzo requieren, por lo que suelen desestimarse. En general, suele existir una retroalimentación entre todos los pasos, ya que es frecuente que en cada nivel aparezcan nuevos requisitos. La Figura 12.1 muestra el bucle de retroalimentación que afecta a las fases de diseño conceptual y lógico como resultado de la implantación y puesta a punto del sistema.

## 12.2 El diseño de la base de datos y el proceso de implementación

Ahora vamos a centrarnos en las Actividades 2 y 3 del ciclo vital del sistema de aplicaciones de bases de datos, que son el diseño y la implementación del misma. El problema de diseñar una base de datos se puede enunciar de este modo:

*Diseñar la estructura lógica y física de una o más bases de datos para acomodar las necesidades de los usuarios de una empresa en cuanto a información para un conjunto concreto de aplicaciones.*

Los objetivos del diseño de una base de datos son varios:

- Satisfacer los requisitos de información de los usuarios y aplicaciones especificados.
- Ofrecer una estructuración de la información natural y fácil de comprender.
- Soportar las necesidades de procesamiento y cualquier objetivo de rendimiento, como el tiempo de respuesta, el tiempo de procesamiento y el espacio de almacenamiento.

Estos objetivos son muy difíciles de lograr y de medir y precisan de contrapartidas: si se intenta alcanzar más *naturalidad* y *comprensibilidad* del modelo, puede resentirse el rendimiento. El problema se agrava porque el proceso de diseño de la base de datos comienza a menudo con unos requisitos informales y pobremente definidos.

Por el contrario, el resultado de la actividad de diseño es un esquema rígidamente definido que no puede modificarse con facilidad una vez que la base de datos está implementada. Podemos identificar seis fases principales en el proceso global de diseño e implementación:

1. Recopilación y análisis de requisitos.
2. Diseño conceptual de la base de datos.
3. Elección de un DBMS.
4. Mapeo del modelo de datos (llamado también diseño lógico de la base de datos).
5. Diseño físico de la base de datos.
6. Implementación y puesta a punto del sistema.

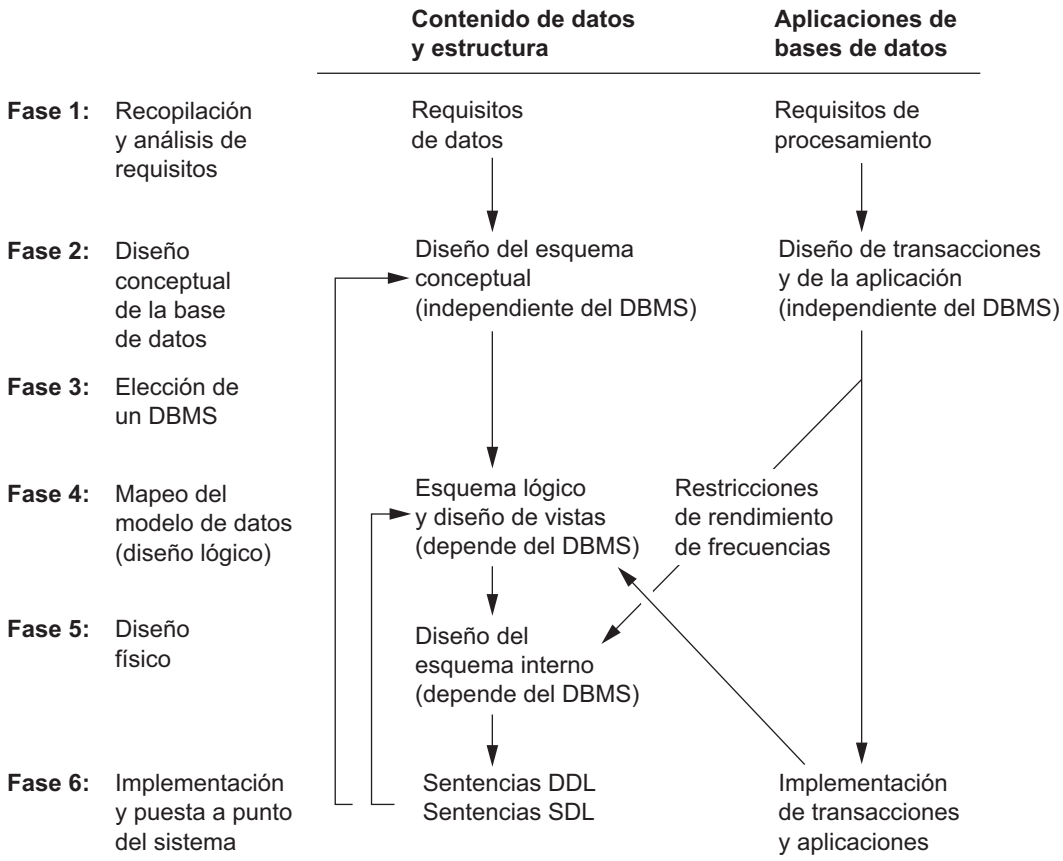
El proceso de diseño consta de dos actividades paralelas (véase la Figura 12.1). La primera supone el diseño de los contenedores de información y de la estructura de la base de datos; la segunda está relacionada con el diseño de las aplicaciones de base de datos. Para que esta figura resulte sencilla, no hemos mostrado la mayoría de las interacciones entre los dos lados, aunque las dos actividades están íntimamente entrelazadas. Por ejemplo, analizando las aplicaciones de bases de datos podemos identificar los elementos que se almacenarán en la base de datos. Además, la fase de diseño físico, durante la cual se eligen las estructuras de almacenamiento y las rutas de acceso a los ficheros, depende de las aplicaciones que usarán esos ficheros. Por otro lado, solemos definir el diseño de las aplicaciones de bases de datos refiriéndonos al esquema de construcción de la misma, que se especifica durante esta primera actividad. Claramente, estas dos actividades influyen la una en la otra. Tradicionalmente, las metodologías de diseño de bases de datos se han centrado inicialmente en la primera de estas actividades, mientras que el diseño del software lo está en la segunda; esto se podría llamar un **diseño orientado-a-datos** frente a un **diseño orientado-a-procesos**. Lo que sí se ha reconocido rápidamente por parte de los diseñadores de bases de datos y los ingenieros de software es que las dos actividades deben ir de la mano, y las herramientas de diseño las combinan cada vez más.

Las seis fases antes mencionadas no tienen que seguirse necesariamente en secuencia. En muchos casos, tendremos que modificar el diseño obtenido en una fase anterior. Estos **bucles de retroalimentación** entre fases (y, con frecuencia, dentro de las fases) son muy comunes.

En la Figura 12.1 sólo se muestran dos de estos bucles, aunque existen muchos más entre cada pareja de fases. Hemos mostrado también alguna interacción entre los datos y los dos lados del proceso de la figura, aunque en la realidad existen muchas más interacciones. La Fase 1 de la Figura 12.1 implica la recopilación de información acerca del uso pretendido que se le va a dar a la base de datos, y la Fase 6 hace referencia a su implementación y rediseño. El corazón del proceso de diseño está en las Fases 2, 4 y 5; resumámoslas:

- **Diseño conceptual de la base de datos (Fase 2)**. El objetivo de esta fase es conseguir un esquema conceptual de la base de datos que sea independiente de un DBMS específico. Con frecuencia, empleamos para ello un modelo de datos de alto nivel, como el ER o el EER (consulte los Capítulos 3 y 4). Por otro lado, especificaremos cuanta información tengamos sobre las aplicaciones o transacciones de la base de datos, usando para ello una notación que sea independiente de cualquier DBMS específico.

**Figura 12.1.** Fases del diseño e implementación de una base de datos grande.




A menudo, la elección del DBMS ya la ha tomado la empresa; a pesar de ello, el intento de un diseño conceptual debe seguir siendo lo más libre posible por consideraciones de implementación.

- **Mapeado del modelo de datos (Fase 4).** Durante esta fase, que también recibe el nombre de diseño lógico de la base de datos, mapeamos (o transformamos) el esquema conceptual procedente del modelo de datos de alto nivel de la Fase 2 al modelo de datos del DBMS elegido. Podemos empezar esta fase después de haber elegido un tipo de DBMS (por ejemplo, si hemos decidido utilizar algún DBMS relacional pero aún no tenemos claro cuál de ellos). Decimos que esto último es un diseño lógico independiente del sistema (pero dependiente del modelo de datos). En palabras de la arquitectura DBMS de tres niveles comentada en el Capítulo 2, el resultado de esta fase es un esquema conceptual en el modelo de datos elegido. Además, el diseño de los esquemas externos (vistas) para las aplicaciones suele realizarse en esta fase.
- **Diseño físico de la base de datos (Fase 5)**. En esta fase preparamos las especificaciones de la base de datos en términos de estructuras físicas de almacenamiento, ubicación del registro e índices. Esto se corresponde con el diseño del *esquema interno* en la terminología de la arquitectura DBMS de tres niveles.
- **Implementación y puesta a punto del sistema de base de datos (Fase 6).** Durante esta fase, se implementan, verifican e implantan la base de datos y los programas que la atacarán. Las diversas transacciones y aplicaciones son testadas individualmente, y después en combinación con otras. Esto suele provocar cambios en el diseño físico, la indexación de los datos, la reorganización y la ubicación de la

información: es una actividad que suele conocerse como **refinamiento de la base de datos**. La puesta a punto es una actividad continua: una parte del mantenimiento del sistema que continúa a lo largo del ciclo vital mientras que la base de datos y las aplicaciones vayan evolucionando como consecuencia del descubrimiento de problemas de rendimiento.

En las subsecciones siguientes trataremos más en profundidad cada una de estas seis fases.

### 12.2.1 Fase 1. Recopilación y análisis de requisitos[1]

Antes de poder diseñar eficazmente la base de datos, debemos conocer y analizar con todo lujo de detalles las expectativas de los usuarios sobre la misma. Este proceso recibe el nombre de **recopilación de requisitos y análisis**. Para especificar los requisitos, primero debemos identificar las otras partes del sistema de información que interactuarán con la base de datos. Esto incluye a los usuarios y las aplicaciones, tanto nuevos como existentes, cuyos requisitos deben recopilarse y analizarse. Habitualmente, las siguientes son algunas de las actividades que forman parte de esta fase:

1. Identificar las áreas de aplicación principales y los grupos de usuarios que usan la base de datos o cuyo trabajo se verá afectado. Dentro de cada grupo, se eligen individuos clave y comités para llevar a cabo la recopilación y especificación de los requisitos.
2. Estudiar y analizar la documentación existente relativa a las aplicaciones. Revisar cualquier otro tipo de documento (manuales de políticas, formularios, informes y gráficos organizativos) para determinar si tiene influencia en la toma de requisitos y el proceso de especificación.
3. Estudiar el entorno operativo actual y el uso que se pretende dar a la información. Esto incluye el análisis de los tipos de transacciones y sus frecuencias, así como el flujo de información dentro del sistema. También hay que estudiar las características geográficas referentes a los usuarios, el origen de las transacciones, el destino de los informes, etc. Especificar la entrada y la salida de datos para las transacciones.
4. Recabar preguntas de los usuarios y grupos potenciales de la base de datos y escribir respuestas para ellas. Estas preguntas conciernen a las prioridades de los usuarios y la importancia que dan a las distintas aplicaciones. Puede entrevistarse a las personas clave para que ayuden a valorar la información y a que establezcan prioridades.

Los análisis de los requisitos para los usuarios finales, o *clientes*, de la base de datos son llevados a cabo por un equipo de analistas o de expertos en requisitos. Las necesidades iniciales serán, muy probablemente, informales, incompletas, incoherentes y parcialmente incorrectas. Por consiguiente, será necesario mucho trabajo para transformar estos datos en una especificación que pueda ser usada por los desarrolladores y probadores como punto de partida para la escritura de la implementación y la verificación. Ya que los requisitos reflejan la posición inicial de un sistema que aún no existe, éstos cambiarán inevitablemente. Por tanto, es importante utilizar técnicas que ayuden a que los clientes converjan rápidamente en los requisitos de implementación.

Existen un gran número de evidencias que demuestran que la participación del cliente en el proceso de desarrollo incrementa la satisfacción de éste con el sistema que se le va a entregar. Por este motivo, muchos especialistas utilizan reuniones y grupos de trabajo que implican a todos los interesados. Una de estas metodologías de refinamiento de los requisitos iniciales del sistema se conoce como JAD (Diseño conjunto de aplicaciones, *Joint Application Design*). Últimamente se han desarrollado otras técnicas, como el Diseño Contextual, que comportan que los diseñadores deban introducirse en el espacio de trabajo en el que se implementará después la aplicación. Para ayudar a que los clientes representativos entiendan más claramente el sistema propuesto, es habitual moverse por el flujo de trabajo o los escenarios de transacción, o crear un prototipo de la aplicación.

[1] Parte de esta sección ha sido aportada por Colin Potts.

Los modos anteriores ayudan a estructurar y refinar los requisitos, pero los mantienen en un estado informal. Para convertirlos a una forma mejor estructurada, se recurre a **técnicas de especificación de requisitos**, que incluyen el OOA (Análisis orientado a objetos, *Object-Oriented Analysis*), los DFD (Diagramas de flujo de datos, *Data Flow Diagrams*) y el refinamiento de los objetivos de la aplicación. Estos métodos utilizan técnicas diagramáticas para organizar y presentar los requisitos de procesamiento de la información. Normalmente, los diagramas van acompañados de documentación en forma de texto, tablas, gráficos y requisitos de decisión. Hay técnicas que producen una especificación formal verificable matemáticamente, de cara a un análisis coherente y del tipo "¿qué pasaría si...?". Estos métodos apenas se utilizan actualmente, pero en el futuro pueden convertirse en un estándar para aquellas partes de los sistemas de información cuya función es crítica y que, por tanto, deben funcionar como se ha planeado. Podemos pensar en los métodos de especificación formal basados en el modelo, de los cuales el más destacado es la metodología y notación Z, como en extensiones del modelo ER y son, por tanto, los más aplicables al diseño de sistemas de información.

Se han propuesto algunas técnicas asistidas por computador, denominadas herramientas CASE de alto nivel (*Upper CASE*), como ayuda para probar la coherencia y la integridad de las especificaciones, que normalmente se almacenan en un solo almacén y se pueden visualizar y actualizar a medida que el diseño progresa. Se utilizan otras herramientas para trazar los enlaces entre los requisitos y otras entidades del diseño, como los módulos de código y los casos de prueba. Este rastreo de las bases de datos es especialmente importante, en combinación con la implementación de unos procedimientos de gestión de cambios en aquellos sistemas donde los requisitos cambian frecuentemente. También se utilizan en proyectos contractuales donde la organización del desarrollo debe proporcionar al cliente una evidencia documental de que se han implementado todos los requisitos.

La fase de recolección y análisis de requisitos puede llevar mucho tiempo, pero es crucial para el éxito del sistema de información. Corregir un error en los requisitos es mucho más costoso que corregir un error durante la implementación, porque los efectos de un error en los requisitos se van arrastrando, y cuanto más avancemos, más trabajo costará su corrección. Si no corregimos el error, significa que el sistema no satisfará al cliente, y puede que éste no llegue a utilizarlo. Hay libros dedicados en exclusiva al tema de la recopilación de los requisitos y su análisis.

### 12.2.2 Fase 2. Diseño conceptual de la base de datos

La segunda fase del diseño de una base de datos comporta dos actividades paralelas.[2] La primera, el **diseño del esquema conceptual**, examina los requisitos de datos resultantes de la Fase 1 y produce un esquema conceptual de la base de datos. La segunda actividad, el **diseño de transacciones y aplicaciones**, examina las aplicaciones de bases de datos analizadas en la Fase 1 y produce especificaciones de alto nivel para esas aplicaciones.

**Fase 2a: Diseño del esquema conceptual.** El esquema conceptual producido en esta fase normalmente se incluye en un modelo de datos independiente del DBMS y de alto nivel, por las siguientes razones:

1. El objetivo del diseño del esquema conceptual es llegar a un conocimiento completo de la estructura de la base de datos, su significado (semántica), sus interrelaciones y sus restricciones. Es recomendable hacerlo con independencia de un DBMS en particular, porque cada DBMS normalmente tiene rarezas y restricciones que no deberían influir en el diseño del esquema conceptual.
2. El esquema conceptual es inestimable como *descripción estable* del contenido de la base de datos. La elección del DBMS y las posteriores decisiones sobre el diseño pueden cambiar sin que haya que modificar el "esquema conceptual independiente del DBMS".

[2] Esta fase del diseño se explica más en profundidad en los primeros siete capítulos de Batini y otros (1992); aquí sólo ofrecemos un resumen.

3. Una buena comprensión del esquema conceptual es crucial para los usuarios de la base de datos y para los diseñadores de las aplicaciones. Es muy importante utilizar un modelo de datos de alto nivel, ya que es más expresivo y general que los modelos de datos de los DBMSs.
4. La descripción diagramática del esquema conceptual puede servir como un vehículo de comunicación excelente entre los usuarios, los diseñadores y los analistas de la base de datos. Como los modelos de datos normalmente dependen de conceptos que son más fáciles de entender que los modelos de datos de bajo nivel propios de los DBMS o que las definiciones sintácticas de datos, cualquier comunicación concerniente al diseño del esquema es más exacta y directa.

En esta fase del diseño de la base de datos, es importante utilizar un modelo de datos conceptual de alto nivel, que tenga las siguientes características:

1. **Expresividad**. El modelo de datos debe ser suficientemente expresivo para distinguir los distintos tipos de datos, relaciones y restricciones.
2. **Simplicidad y comprensibilidad**. El modelo debe ser bastante simple para que los usuarios no experimentados lo entiendan y utilicen sus conceptos.
3. **Minimalista**. El modelo debe tener una pequeña cantidad de conceptos básicos que sean distintos y cuyo significado no se solape.
4. **Representación diagramática**. El modelo debe tener una notación diagramática fácil de entender y que sirva para visualizar el esquema conceptual.
5. **Formalidad**. Un esquema conceptual expresado en el modelo de datos debe representar una especificación formal e inequívoca de los datos. Por tanto, los conceptos del modelo deben definirse con precisión y sin ambigüedad.

Muchos de estos requisitos (el primero de ellos en particular), a menudo entran en conflicto con otros. Para el diseño de bases de datos se han propuesto muchos modelos conceptuales de alto nivel (consulte la bibliografía seleccionada del Capítulo 4). En la siguiente exposición, utilizaremos la terminología del modelo Entidad-Relación mejorado (EER) que presentamos en el Capítulo 4 y asumiremos que está utilizándose en esta fase. El diseño del esquema conceptual, incluyendo el modelado de los datos, se está convirtiendo en parte integral de las metodologías de diseño y análisis orientadas a objetos. El UML tiene diagramas de clase basados principalmente en las extensiones del modelo EER.

Metodologías de diseño del esquema conceptual. Para el diseño del esquema conceptual, debemos identificar los componentes básicos del esquema: los tipos de entidades, los tipos de relaciones y los atributos. También debemos especificar los atributos clave, la cardinalidad y las restricciones de participación en las relaciones, los tipos de entidades débiles, y la especialización/generalización de jerarquías/entramados. Hay dos metodologías para diseñar el esquema conceptual, derivadas de los requisitos recopilados durante la Fase 1.

La primera metodología es la **metodología centralizada** (o ***one shot***) **de diseño del esquema**, en la que se fusionan en un solo conjunto los requisitos de las diferentes aplicaciones y grupos de usuarios recopilados durante la Fase 1, antes de empezar con el diseño del esquema. Después se diseña un solo esquema, correspondiente al conjunto de requisitos fusionados. Cuando existen muchos usuarios y aplicaciones, la mezcla de todos los requisitos puede ser una tarea ardua y larga. Hemos de suponer que hay una autoridad centralizada, el DBA, responsable de decidir cómo se fusionan los requisitos y cómo se diseña el esquema conceptual de toda la base de datos. Una vez diseñado y finalizado el esquema conceptual, el DBA puede especificar los esquemas externos para los distintos grupos de usuarios y aplicaciones.

La segunda metodología es la **metodología de integración de vistas**, en la que los requisitos no se fusionan. Por contra, se diseña un esquema (o vista) por cada grupo de usuarios o aplicaciones en base únicamente a sus propios requisitos. De este modo, desarrollamos un esquema (vista) de alto nivel por cada grupo de usuarios o aplicaciones. Durante una subsiguiente fase de **integración de vistas**, estos esquemas se fusionan o

integran en un **esquema conceptual global** para toda la base de datos. Después de la integración de las vistas, es posible reconstruir las vistas individuales como esquemas externos.

La principal diferencia entre las dos metodologías descansa en la manera o etapa en la que se reconcilian y fusionan varias vistas o requisitos de los muchos usuarios y aplicaciones. En la metodología centralizada, la reconciliación la realiza manualmente el personal del DBA antes de diseñar cualquier esquema y se aplica directamente a los requisitos recopilados en la Fase 1. De este modo, el peso de reconciliar las diferencias y los conflictos entre los grupos de usuarios recae en el personal del DBA. Normalmente, este problema se ha encargado a consultores/expertos en diseño externos que han implantado sus propias formas de resolver estos conflictos. Debido a las dificultades de dirigir esta tarea, la metodología de la integración de vistas tiene cada vez más aceptación.

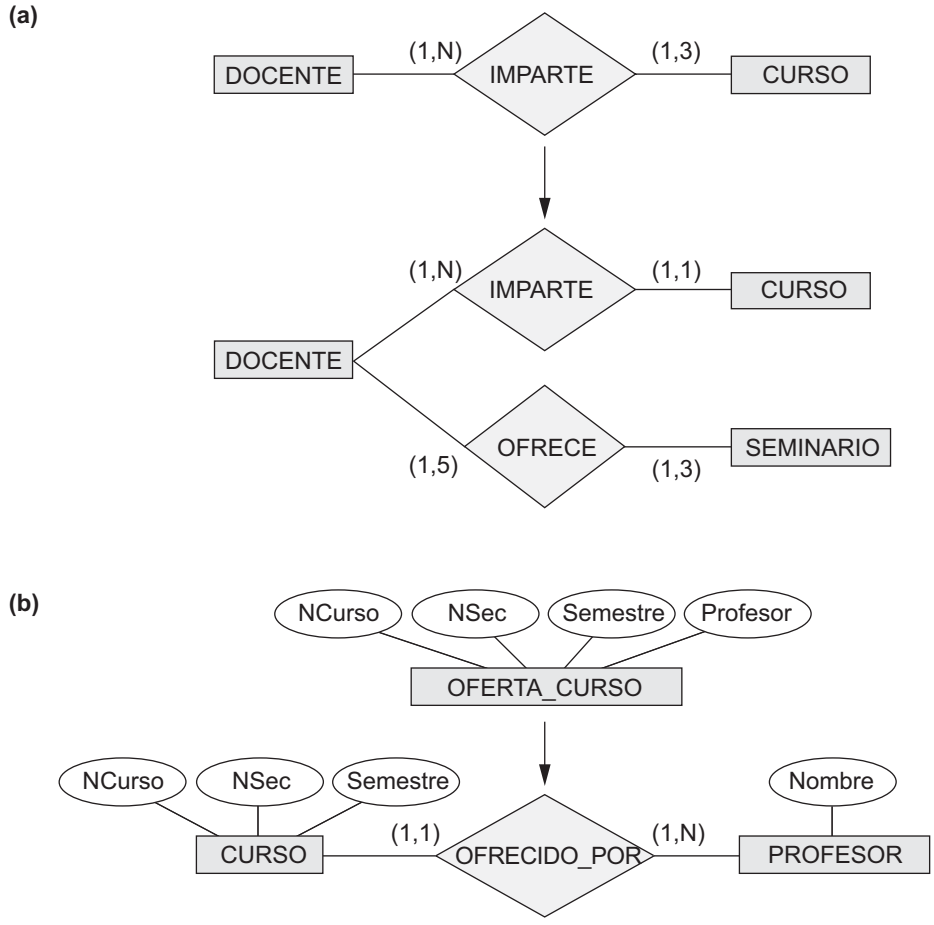
En esta última metodología, cada grupo de usuarios o aplicaciones diseña su propio esquema (EER) conceptual a partir de sus requisitos. Después, el DBA aplica un proceso de integración a esos esquemas (vistas) para formar el esquema integrado global. Aunque la integración de vistas se puede realizar manualmente, su aplicación a una base de datos grande que implica decenas de grupos de usuarios requiere una metodología y el uso de herramientas automatizadas que ayuden a llevar a cabo la integración. Las correspondencias entre los atributos, los tipos de entidades y los tipos de relaciones en varias vistas deben especificarse antes de que se pueda aplicar la integración. Además, hay que enfrentarse a problemas como integrar las vistas en conflicto y verificar la coherencia de las correspondencias entre esquemas.

Estrategias para diseñar un esquema. Dado un conjunto de requisitos, para un solo usuario o para una comunidad de usuarios más grande, debemos crear un esquema conceptual que satisfaga dichos requisitos, que podemos diseñar siguiendo varias estrategias. La mayoría de ellas sigue un método incremental (es decir, empiezan con algunas estructuras de esquema derivadas de los requisitos y, después, las modifican, refinan o construyen incrementalmente). Vamos a explicar algunas de estas estrategias:

1. **Estrategia descendente**. Empezamos con un esquema que contiene abstracciones de alto nivel y después aplicamos refinamientos sucesivos de arriba abajo. Por ejemplo, podemos especificar sólo unos cuantos tipos de entidades de nivel alto y, a continuación, al especificar sus atributos, dividirlas en tipos de entidades de bajo nivel y relaciones. El proceso de especialización para refinar un tipo de entidad en subclases que ilustramos en las Secciones 4.2 y 4.3 (véanse las Figuras 4.1, 4.4 y 4.5) es otro ejemplo de estrategia descendente.
2. **Estrategia ascendente.** Empezamos con un esquema que contiene las abstracciones básicas, y después combinamos o añadimos a esas abstracciones. Por ejemplo, podemos empezar con los atributos y agruparlos en tipos de entidades y relaciones. Podemos añadir relaciones nuevas entre los tipos de entidades a medida que progresa el diseño. El proceso de generalizar los tipos de entidades en superclases generalizadas de alto nivel (consulte las Secciones 4.2 y 4.3, y la Figura 4.3) es otro ejemplo de estrategia de diseño ascendente.
3. **Estrategia de dentro a fuera.** Se trata de un caso especial de estrategia ascendente, en la que la atención se centra en un conjunto central de conceptos que son muy evidentes. El modelado se extiende después hacia fuera considerando los conceptos nuevos que se encuentran en la vecindad de los ya existentes. Podríamos especificar unos pocos tipos de entidades claramente evidentes en el esquema y continuar añadiendo otros tipos de entidades y relaciones relacionados con ellos.
4. **Estrategia mixta.** En lugar de seguir una estrategia en particular durante el diseño, los requisitos se dividen según una estrategia descendente, y se diseña parte del esquema para cada una de esas divisiones siguiendo una estrategia ascendente. Después se combinan las distintas partes del esquema.

Las Figuras 12.2 y 12.3 ilustran los refinamientos descendente y ascendente, respectivamente. Un ejemplo primitivo de refinamiento descendente es la descomposición de un tipo de entidad en varios tipos de entidades. La Figura 12.2(a) muestra un CURSO refinado en CURSO y SEMINARIO, y la relación IMPARTE dividida en IMPARTE y OFRECE. La Figura 12.2(b) muestra un tipo de entidad OFERTA_CURSO refinado en

**Figura 12.2.** Ejemplos de refinamiento descendente. (a) Generación de un nuevo tipo de entidad. (b) Descomposición de un tipo de entidad en dos tipos de entidades y un tipo de relación.

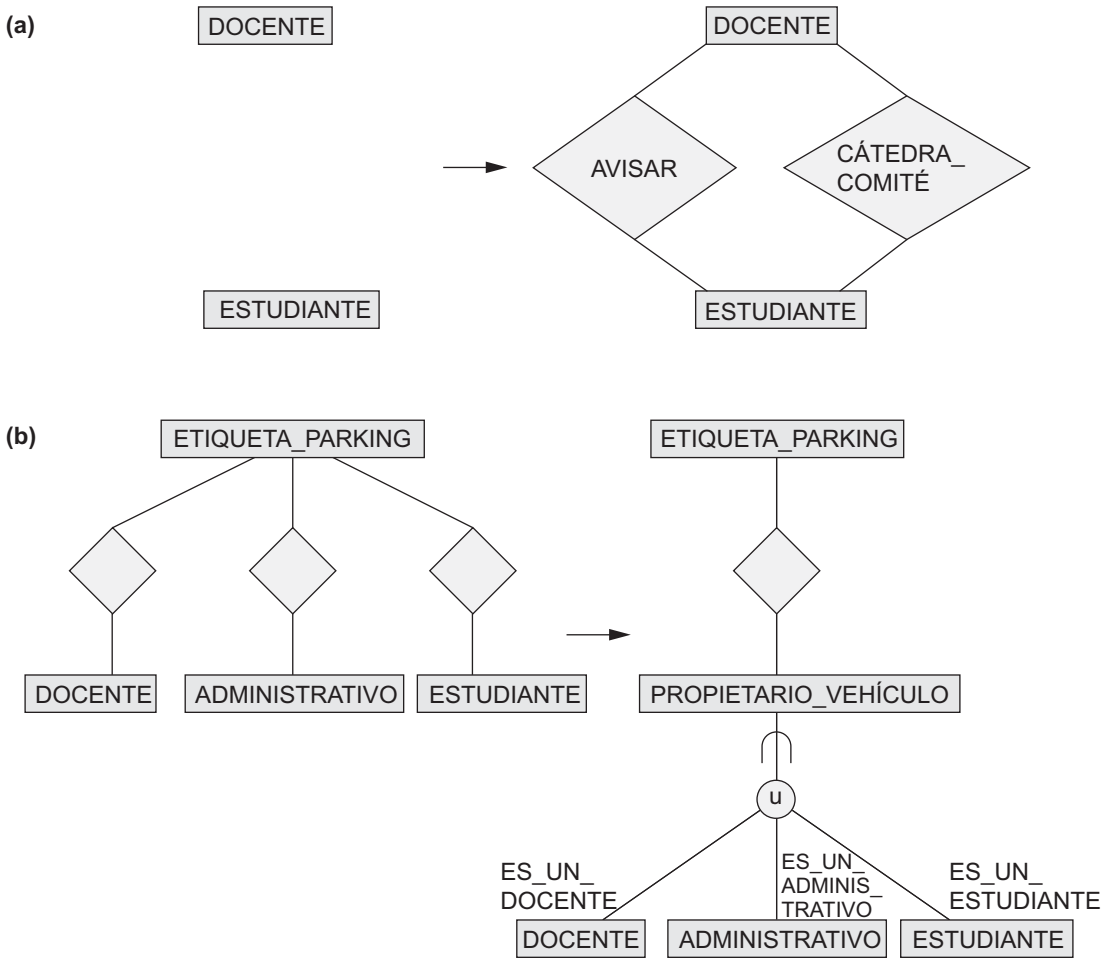



dos tipos de entidades (CURSO y PROFESOR) y una relación entre ellas. El refinamiento normalmente obliga al diseñador a hacer más preguntas y extraer más restricciones y detalles: por ejemplo, las razones de cardinalidad (mín, máx) entre CURSO y PROFESOR se obtienen durante el refinamiento. La Figura 12.3(a) muestra el primitivo refinamiento ascendente consistente en generar relaciones nuevas entre los tipos de entidades. El refinamiento ascendente utilizando la categorización (tipo unión) se ilustra en la Figura 12.3(b), donde se descubre el concepto nuevo de PROPIETARIO_VEHÍCULO a partir de los tipos de entidades DOCENTE, ADMINISTRATIVO y ESTUDIANTE existentes; este proceso de creación de una categoría y la notación diagramática relacionada sigue lo que explicamos en la Sección 4.4.

Integración de esquema (vista). En las bases de datos grandes con posibilidad de muchos usuarios y aplicaciones, es posible utilizar la metodología de la integración de vistas para diseñar esquemas individuales y después mezclarlos. Como las vistas individuales se pueden mantener con un tamaño relativamente pequeño, se simplifica el diseño de los esquemas. No obstante, se necesita una metodología para integrar las vistas en un esquema de base de datos global. La integración del esquema se puede dividir en las siguientes subtareas:

1. **Identificar correspondencias y conflictos entre los esquemas**. Como los esquemas se diseñan individualmente, debemos especificar estructuras en los esquemas que representen el mismo concepto que

**Figura 12.3.** Ejemplos de refinamiento ascendente. (a) Descubrimiento y adición de relaciones nuevas. (b) Descubrimiento y relación de una nueva categoría (tipo unión).




en el mundo real. Estas correspondencias deben identificarse antes de efectuar la integración. Durante este proceso, podemos descubrir varios tipos de conflictos entre los esquemas:

a. **Conflictos con los nombres.** Hay dos tipos: sinónimos y homónimos. Un **sinónimo** se produce cuando dos esquemas utilizan nombres diferentes para describir el mismo concepto; por ejemplo, un tipo de entidad CLIENTE en un esquema puede describir el mismo concepto que un tipo de entidad CLIENTE en otro esquema. Un **homónimo** se produce cuando dos esquemas utilizan el mismo nombre para describir conceptos diferentes; por ejemplo, un tipo de entidad PIEZA puede representar los componentes de un computador en un esquema y elementos de mobiliario en otro.

b. **Conflictos de tipo.** El mismo concepto puede representarse en dos esquemas mediante estructuras de modelado diferentes. Por ejemplo, el concepto de un DEPARTAMENTO puede ser un tipo de entidad en un esquema y un atributo en otro.

c. **Conflictos de dominio (conjunto de valores).** Un atributo puede tener dominios diferentes en dos esquemas. Por ejemplo, Dni puede declararse como un entero en un esquema y como cadena de caracteres en otro. Por ejemplo, se puede producir un conflicto de unidad de medida si un esquema representa el Peso en libras y otro en kilogramos.

   d. **Conflictos entre restricciones.** Dos esquemas pueden imponer restricciones diferentes; por ejemplo, la clave de un tipo de entidad puede ser diferente en cada esquema. Otro ejemplo puede ser el que implique a diferentes restricciones estructurales en una relación como IMPARTE; un esquema puede representarla como 1:N (un curso tiene un profesor), mientras que otro esquema puede representarla como M:N (un curso puede tener más de un profesor).

2. **Modificación de vistas para adaptarse entre sí.** Algunos esquemas se modifican para que se adapten mejor a otros esquemas. Algunos de los conflictos identificados en la primera subtarea se resuelven durante este paso.

3. **Combinación de vistas.** El esquema global se crea mezclando los esquemas individuales. En el esquema global sólo se representan una vez los conceptos correspondientes, y se especifican los mapeos entre las vistas y el esquema global. Éste es el paso más difícil para lograr implicar en las bases de datos reales cientos de entidades y relaciones. Implica una considerable cantidad de intervención y negociación humanas para resolver los conflictos y llegar a las soluciones más razonables y aceptables en lo que al esquema global se refiere.

4. **Reestructuración.** Como paso final (y opcional), debemos analizar y reestructurar el esquema global para eliminar las redundancias o las complejidades innecesarias.

Algunas de estas ideas se ilustran con ejemplos sencillos en las Figuras 12.4 y 12.5. En la Figura 12.4 se han mezclado dos vistas para crear una base de datos bibliográfica. Durante la identificación de las correspondencias entre las dos vistas, descubrimos que INVESTIGADOR Y AUTOR son sinónimos (hasta donde concierne a esta base de datos), al igual que REALIZADO_POR y ESCRITO_POR. Además, hemos decidido modificar la vista 1 para incluir un TEMA para el ARTÍCULO, como se muestra en la Figura 12.4, *para configurar* la vista 2. La Figura 12.5 muestra el resultado de combinar la vista 1 modificada con la vista 2. Generalizamos los tipos de entidad ARTÍCULO y LIBRO en el tipo de entidad PUBLICACIÓN, con su atributo Título común. Las relaciones REALIZADO_POR y ESCRITO_POR se mezclan, así como INVESTIGADOR y AUTOR. El atributo Editor sólo se aplica a la entidad LIBRO, mientras que el atributo Tamaño y el tipo de entidad PUBLICADO_EN sólo se aplica a ARTÍCULO.

El ejemplo anterior ilustra la complejidad del proceso de combinación y de cómo hay que considerar el significado de varios conceptos para simplificar el diseño del esquema resultante. En los diseños reales, el proceso de integración del esquema requiere una metodología más disciplinada y sistemática. Se han propuesto varias estrategias para el proceso de integración de vistas (véase la Figura 12.6):

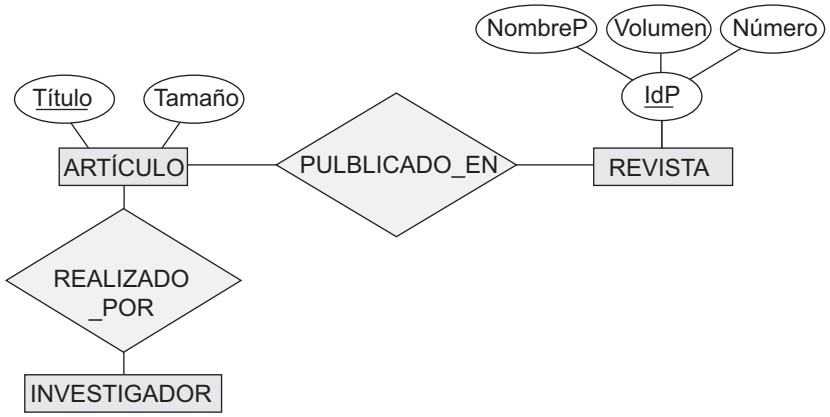
1. **Integración de escala binaria**. Primero se integran dos esquemas que son muy parecidos. El esquema resultante se integra después con otro esquema, y el proceso se repite hasta que se hayan integrado todos los esquemas. El orden de los esquemas en la integración puede estar basado en alguna medida de similitud de los esquemas. Esta estrategia es adecuada para una integración manual, debido a su metodología paso a paso.

2. **Integración de n vistas**. Todas las vistas se integran en un procedimiento después de un análisis y de la especificación de sus correspondencias. Esta estrategia requiere herramientas informáticas para los problemas de diseño más difíciles. Se han desarrollado herramientas de este tipo como prototipos de investigación, pero todavía no están disponibles comercialmente.

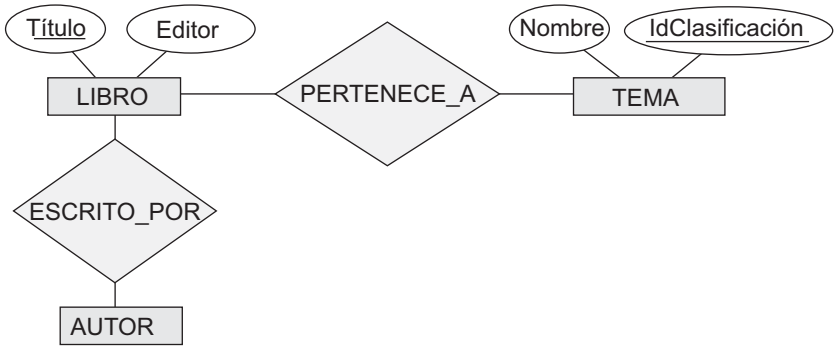
3. **Estrategia equilibrada binaria**. Primero se integran pares de esquemas; después, se emparejan los esquemas resultantes para un integración posterior; este procedimiento se repite hasta llegar a un esquema global final.

4. **Estrategia mixta**. Inicialmente, los esquemas se dividen en grupos basándose en su similitud, y cada grupo se integra por separado. Los esquemas intermedios se agrupan e integran de nuevo, y así sucesivamente.

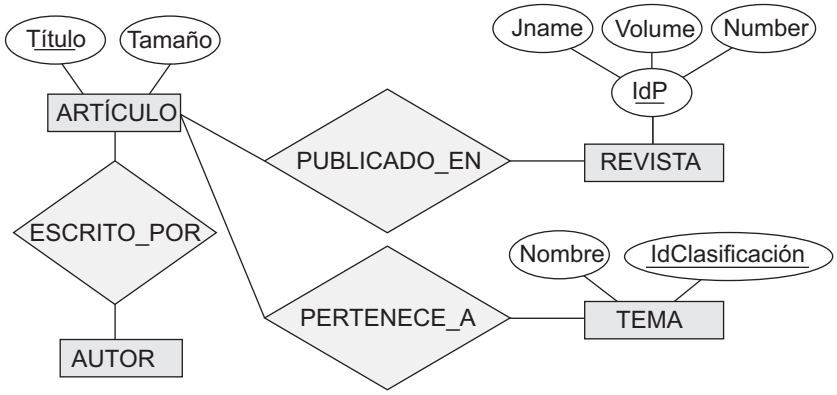
**Figura 12.4.** Modificación de vistas antes de la integración.




**Vista 1**



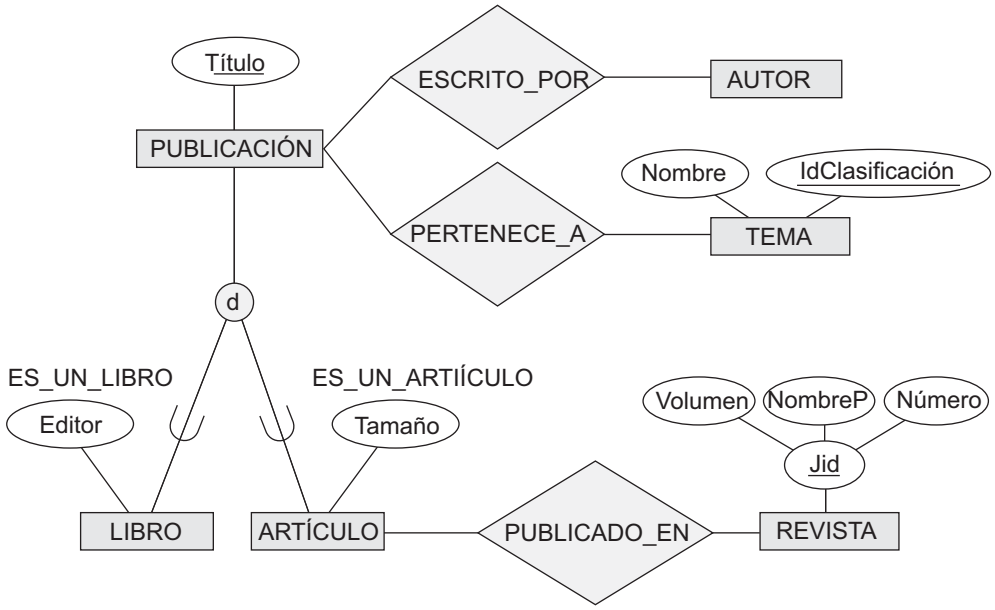

**Vista 2**




**Vista 1 modificada**

**Fase 2b: Diseño de transacciones.** El propósito de esta fase, que prosigue en paralelo con la Fase 2a, es diseñar las características de las transacciones (aplicaciones) conocidas de la base de datos, independiente-

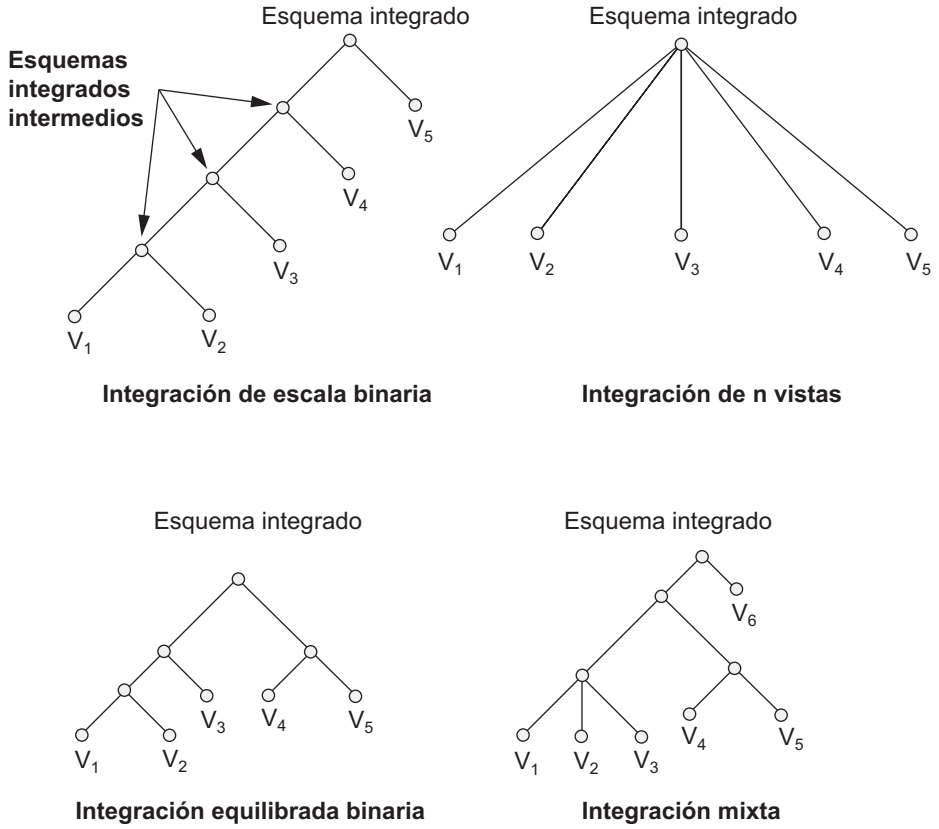
**Figura 12.5.** Esquema integrado después de combinar las vistas 1 y 2.




mente del DBMS. Cuando se está diseñando un sistema de bases de datos, los diseñadores son conscientes de las muchas aplicaciones (o **transacciones**) conocidas que se ejecutarán en la base de datos una vez implementada. Una parte importante del diseño de la base de datos es especificar lo antes posible en el proceso de diseño las características funcionales de esas transacciones. Esto garantiza que el esquema de la base de datos incluirá toda la información requerida por esas transacciones. Además, el conocimiento de la relativa importancia de las distintas transacciones y de las tasas esperadas de su invocación es una parte crucial del diseño físico de la base de datos (Fase 5). Normalmente, sólo se conocen algunas de las transacciones en la fase de diseño; una vez implementado el sistema de bases de datos, se identifican e implementan continuamente nuevas transacciones. No obstante, las transacciones más importantes a menudo se conocen antes de implementar el sistema, y deben especificarse en una etapa más temprana. En este contexto es típico aplicar la *regla 80–20*: el 80 por ciento de la carga de trabajo es representada por el 20 por ciento de las transacciones más frecuentes, que gobiernan el diseño. En las aplicaciones en las que hay variedad de consultas y procesamientos por lotes, deben identificarse las consultas y aplicaciones que procesan una cantidad sustancial de datos.

Una técnica común para especificar las transacciones a nivel conceptual es identificar su **comportamiento de entrada/salida** y **funcional**. Al especificar los parámetros (argumentos) de entrada y salida, y el flujo de control interno funcional, los diseñadores pueden especificar una transacción de una forma conceptual e independiente del sistema. Las transacciones normalmente se pueden agrupar en tres categorías: (1) **transacciones de recuperación**, que se utilizan para recuperar datos para luego visualizarlos en pantalla o para generar un informe; (2) **transacciones de actualización**, que se utilizan para introducir datos nuevos o modificar los existentes en una base de datos; (3) **transacciones mixtas**, que se utilizan para aplicaciones más complejas que suponen alguna recuperación y alguna actualización. Por ejemplo, considere la base de datos de reservas en una aerolínea. Una transacción de recuperación podría listar todos los vuelos matinales entre dos ciudades en una fecha determinada. Una transacción de actualización podría hacer la reserva de plaza en un vuelo determinado. Una transacción mixta podría visualizar primero algunos datos, como la reserva de un cliente en algún vuelo, y después actualizar la base de datos (por ejemplo, cancelando la reserva y borrándola, o añadiendo un segmento de vuelo a una reserva existente). Las transacciones (aplicaciones) se pueden originar en

**Figura 12.6.** Diferentes estrategias para el proceso de integración de vistas.




una herramienta frontal como PowerBuilder 10.0 (Sybase) o Developer 2000 (Oracle), que recopila *online* los parámetros y después envía una transacción al DBMS que actúa como *back-end*.[3]

Varias de las técnicas de especificación de requisitos incluyen la notación para especificar **procesos**, que en este contexto son operaciones más complejas consistentes en varias transacciones. Las herramientas de modelado de procesos como BPWin, así como las de modelado del flujo de trabajo, son cada vez más populares en las empresas, que las utilizan para identificar los flujos de información. El lenguaje UML, que proporciona el modelado de datos a través de diagramas de objetos y clases, cuenta con distintos diagramas de modelado de procesos, incluyendo diagramas de transacción del estado, diagramas de actividad, diagramas de secuencia y diagramas de colaboración. Todos ellos se refieren a actividades, eventos y operaciones dentro del sistema de información, las entradas y las salidas de los procesos y los requisitos de secuenciación y sincronización, entre otras condiciones. Es posible refinar estas especificaciones y extraer transacciones individuales de ellas. Otras propuestas para la especificación de transacciones son TAXIS, GALILEO y GORDAS (consulte la bibliografía seleccionada). Algunas de ellas se han implementado en prototipos de sistemas y herramientas. El modelado de procesos sigue siendo un área activa de la investigación.

El diseño de transacciones es tan importante como el diseño del esquema, pero a menudo se considera que es parte de la ingeniería de software, más que del diseño de bases de datos. Muchas metodologías de diseño

[3] Esta filosofía se ha seguido durante más de 20 años en productos tan conocidos como CICS, que sirve como herramienta para generar transacciones para los DBMSs heredados como IMS.

actuales dan más importancia a uno que a otro. Hay que pasar por las Fases 2a y 2b en paralelo, utilizando el refinamiento, hasta conseguir un diseño estable del esquema y las transacciones.[4]

### 12.2.3 Fase 3. Elección de un DBMS

La elección de un DBMS está gobernada por varios factores (algunos técnicos, otros económicos y, a pesar de todo, otros relativos a las políticas de la empresa). Los factores técnicos tienen que ver con la idoneidad del DBMS para la tarea entre manos. Los problemas que deben considerarse son el tipo de DBMS (relacional, objetos relacionales, objetos, otros), las estructuras de almacenamiento y las rutas de acceso soportadas por el DBMS, las interfaces de usuario y programación disponibles, los tipos de lenguajes de consulta de alto nivel, la disponibilidad de herramientas de desarrollo, la posibilidad de interactuar con otros DBMSs a través de interfaces estándar, las opciones de arquitectura relacionadas con la operativa cliente-servidor, etcétera. Los factores no técnicos incluyen el estado financiero y el soporte por parte del distribuidor o fabricante. En esta sección nos centraremos en explicar los factores económicos y organizativos que afectan a la elección del DBMS. Debemos tener en cuenta los siguientes costes:

1. **Coste de adquisición del software**. Es el coste *directo* por comprar el software, incluyendo las opciones del lenguaje, las diferentes opciones de interfaz (formularios, menús y herramientas GUI [interfaz gráfica de usuario] basadas en la Web), las opciones de recuperación y copia de seguridad, los métodos especiales de acceso, y la documentación. Hay que seleccionar la versión correcta del DBMS para un sistema operativo concreto. En el precio básico normalmente no se incluyen las herramientas de desarrollo, las herramientas de diseño y el soporte adicional del idioma.
2. **Coste de mantenimiento**. Es el coste recurrente por recibir un servicio de mantenimiento estándar ofrecido por el fabricante y por mantener actualizada la versión del DBMS.
3. **Coste de adquisición del hardware**. Es posible que se necesite hardware nuevo, como memoria adicional, terminales, unidades de disco y controladores, o almacenamiento especializado para el DBMS.
4. **Coste de creación y conversión de la base de datos**. Es el coste de crear el sistema de bases de datos desde el principio o de convertir un sistema existente al nuevo software DBMS. En el último caso es habitual hacer funcionar el sistema existente en paralelo con el sistema nuevo hasta haber implementado y probado completamente todas las aplicaciones nuevas. Este coste es muy difícil de proyectar y a menudo se subestima.
5. **Coste de personal**. La adquisición de un software DBMS por primera vez suele suponer para la empresa una reorganización del departamento de procesamiento de datos. En la mayoría de las empresas que han adoptado un DBMS normalmente cuenta con DBA y su personal.
6. **Coste de formación**. Como los DBMSs suelen ser sistemas complejos, a menudo es preciso formar al personal en su uso y programación. La formación es necesaria a todos los niveles, incluyendo la programación, el desarrollo de aplicaciones y la administración de la base de datos.
7. **Coste de funcionamiento**. El coste de un funcionamiento continuado del sistema de bases de datos normalmente no se tiene en cuenta en la evaluación de alternativas, porque se incurre en él independientemente del DBMS seleccionado.

Los beneficios de adquirir un DBMS no son fácilmente cuantificables. Un DBMS ofrece varias ventajas intangibles frente a los sistemas de ficheros tradicionales, como la facilidad de uso, la consolidación de la información de la empresa, la amplia disponibilidad de datos, y un acceso más rápido a la información. Con el acceso basado en la Web, es posible que ciertas partes de los datos sean globalmente accesibles por parte de empleados y usuarios externos. Como beneficios más tangibles podemos citar la reducción del coste por desarrollo de la aplicación, la reducción de la redundancia de datos, y la mejora del control y la seguridad.

[4] El modelado de transacciones de alto nivel se explica en Batini y otros (1992, Capítulos 8, 9 y 11). A lo largo de este libro se defiende la filosofía de la conjunción funcional y el análisis de datos.

Aunque las bases de datos se han "atrincherado" fuertemente en la mayoría de las empresas, surge con frecuencia la decisión de si moverse de un sistema basado en ficheros a un sistema centrado en bases de datos. Este movimiento viene determinado por los siguientes factores:

1. **Complejidad de los datos**. Al hacerse más complejas las relaciones entre los datos, crece la necesidad de un DBMS.
2. **Posibilidad de compartir entre aplicaciones**. Cuanto más se comparte entre aplicaciones, más redundancia hay entre los ficheros y, por tanto, mayor necesidad de un DBMS.
3. **Evolución o crecimiento dinámico de los datos**. Si los datos cambian constantemente, es más fácil hacer frente a estos cambios con un DBMS que utilizando un sistema de ficheros.
4. **Consultas de datos frecuentes**. No todos los sistemas de ficheros son adecuados para la recuperación adecuada de datos.
5. **Volumen de datos y necesidad de control**. El volumen de datos y la necesidad de control requieren a veces un DBMS.

Es difícil desarrollar un conjunto genérico de directrices para adoptar una sola metodología de administración de datos dentro de una empresa (relacional, orientada a objetos o de objetos relacionales). Si los datos que se van a almacenar en la base de datos tienen un nivel alto de complejidad y muchos tipos de datos, lo normal es recurrir a un DBMS de objetos o de objetos relacionales.[5] Además, los beneficios de la herencia entre clases y la correspondiente ventaja que supone la reutilización favorecen estas metodologías. Por último, la elección de un DBMS u otro se ve afectada por diversos factores económicos y organizativos:

1. **Adopción en toda la empresa de una determinada filosofía**. A menudo, éste es un factor dominante que afecta a la aceptabilidad de un cierto modelo de datos (por ejemplo, relacional frente a objetos), un determinado fabricante, o una determinada metodología de desarrollo y herramientas (por ejemplo, puede que se requiera de todas las aplicaciones nuevas el uso de una metodología y una herramienta de análisis y diseño orientadas a objetos).
2. **Familiaridad del personal con el sistema**. Si el personal de programación de la empresa está familiarizado con un DBMS concreto, se puede reducir el coste de formación y el tiempo de aprendizaje.
3. **Disponibilidad de servicios por parte del proveedor**. Es muy importante la disponibilidad de asistencia por parte del proveedor para resolver los problemas del sistema; hay que tener en cuenta que pasar de un entorno sin DBMS a uno DBMS es una gran tarea y requiere al principio mucha asistencia por parte del proveedor.

Otro factor que debe tenerse en cuenta es la portabilidad del DBMS entre los diferentes tipos de hardware. Muchos DBMSs comerciales tienen ahora versiones que se pueden ejecutar en muchas configuraciones hardware/software (o **plataformas**). También hay que pensar en la necesidad de aplicaciones para copias de seguridad, recuperación, rendimiento, integridad y seguridad. Actualmente se están diseñando muchos DBMSs como *soluciones totales* para satisfacer las necesidades de las empresas en cuanto a procesamiento de la información y administración de los recursos de información. La mayoría de los proveedores de DBMSs están combinando sus productos con las siguientes opciones o características integradas:

- Editores de texto y exploradores.
- Generadores de informes y utilidades de listado.
- Software de comunicación (denominado a veces monitor de teleprocesamiento).
- Características de entrada y visualización de datos, como formularios, pantallas y menús con características de edición automáticas.

[5] Consulte la explicación del Capítulo 22 relativa a este tema.

- Herramientas de búsqueda y acceso que se pueden utilizar en la World Wide Web (herramientas compatibles con la Web).
- Herramientas gráficas de diseño de bases de datos.

En el mercado abundan *aplicaciones de terceros* que proporcionan funcionalidad añadida a un DBMS en cada una de las áreas anteriormente citadas. En casos raros es preferible desarrollar software dentro de la empresa en lugar de utilizar un DBMS; por ejemplo, si las aplicaciones están muy bien definidas y se conoce todo de antemano. Bajo estas circunstancias, un sistema diseñado a medida y desarrollado en la empresa puede ser apropiado para implementar las aplicaciones conocidas de una forma más eficaz. No obstante, en la mayoría de los casos, tras la implementación del sistema surgen nuevas aplicaciones que no se previeron durante el diseño. Es precisamente por esto por lo que los DBMSs son tan populares: facilitan la incorporación de aplicaciones nuevas únicamente con modificaciones incrementales del diseño existente de la base de datos. Esta evolución del diseño (o **evolución del esquema**) es una característica presente en distintos grados en los DBMSs comerciales.

### 12.2.4 Fase 4. Mapeo del modelo de datos (diseño lógico de la base de datos)

La siguiente fase del diseño de una base de datos es crear un esquema conceptual y esquemas externos en el modelo de datos del DBMS elegido mapeando los esquemas producidos en la Fase 2a. El mapeo se puede llevar a cabo en dos etapas:

1. **Mapear independientemente del sistema**. En esta etapa, el mapeo no considera ninguna característica específica o caso especial que se aplique a la implementación DBMS del modelo de datos. En la Sección 7.1 explicamos el mapeo de un esquema ER a un esquema relacional independientemente del DBMS, y en la Sección 7.2 el mapeo de los esquemas EER a esquemas relacionales.
2. **Ajustar los esquemas a un DBMS específico.** Los diferentes DBMSs implementan un modelo de datos utilizando características de modelado y restricciones específicas. Puede que haya que ajustar los esquemas obtenidos en el paso 1 para adecuarlos a las características de implementación específicas del modelo de datos que se utiliza en el DBMS elegido.

El resultado de esta fase deben ser sentencias DDL en el lenguaje del DBMS elegido que especifican los esquemas a nivel conceptual y externo del sistema de bases de datos. Pero si las sentencias DDL incluyen algunos parámetros de diseño físico, la especificación DDL completa debe esperar hasta haberse completado la fase de diseño físico de la base de datos. Muchas herramientas de diseño CASE (ingeniería de software asistida por computador) automatizadas (consulte la Sección 12.5) pueden generar DDL para los sistemas comerciales a partir de un diseño de esquema conceptual.

### 12.2.5 Fase 5. Diseño físico de la base de datos

El diseño físico de la base de datos es el proceso de elegir estructuras de almacenamiento específicas y rutas de acceso para los ficheros de la base, a fin de lograr un buen rendimiento de las distintas aplicaciones de la base de datos. Cada DBMS ofrece distintas opciones para la organización de los ficheros y las rutas de acceso, entre las que destacan varios tipos de indexación, agrupamiento de registros relacionados en bloques de disco, enlace de registros relacionados mediante punteros, y varios tipos de dispersión (*hashing*). Una vez elegido un DBMS, el proceso de diseño físico de la base de datos queda restringido a elegir las estructuras más apropiadas para lo ficheros de la base de datos entre las opciones ofrecidas por dicho DBMS. En esta sección ofrecemos unas directrices genéricas relativas a las decisiones de diseño físico, que sirven para cualquier tipo de DBMS. A menudo se utilizan los siguientes criterios como guía para elegir las opciones de diseño físico de la base de datos:

1. **Tiempo de respuesta**. Es el tiempo transcurrido entre el envío de una transacción de base de datos para su ejecución y la recepción de una respuesta. Una influencia importante en el tiempo de respuesta que está bajo el control del DBMS es el tiempo de acceso de la base de datos a los datos a los que hace referencia la transacción. El tiempo de respuesta también se ve influido por factores que no están bajo el control del DBMS, como la carga del sistema, la planificación del sistema operativo, o los retardos en la comunicación.
2. **Utilización del espacio**. Es la cantidad de espacio de almacenamiento utilizada por los ficheros de la base de datos y sus estructuras de ruta de acceso al disco, incluyendo los índices y otras rutas de acceso.
3. **Rendimiento o flujo de transacciones**. Es la cantidad media de transacciones que se pueden procesar por minuto; es un parámetro crítico para los sistemas de transacciones, como los que se utilizan para las reservas en aerolíneas o en la banca. Este dato debe medirse cuando el sistema se encuentre bajo condiciones de máximos (picos).

Normalmente, se especifican los valores medio y mínimo de los parámetros anteriores como parte de los requisitos de rendimiento del sistema. Se utilizan técnicas analíticas o experimentales, que pueden incluir prototipos y simulaciones, para estimar dichos valores bajo diferentes decisiones de diseño físico, a fin de determinar si satisfacen los requisitos de rendimiento especificados.

El rendimiento depende del tamaño del registro y del número de registros del fichero. Por tanto, debemos calcular estos parámetros por cada fichero. Además, debemos estimar acumulativamente los patrones de actualización y recuperación de todas las transacciones. Los atributos utilizados para seleccionar registros deben tener rutas de acceso principales e índices secundarios construidos para ellos. También debemos tener en cuenta durante el diseño físico de la base de datos el crecimiento de los ficheros, como consecuencia del aumento del tamaño del registro debido a la adición de nuevos atributos o porque aumente el número de ficheros.

El resultado de la fase de diseño físico de la base de datos es una determinación *inicial* de las estructuras de almacenamiento y de las rutas de acceso para los ficheros de la base de datos. Casi siempre es necesario modificar el diseño en base al rendimiento observado después de haber implementado el sistema de bases de datos. Esta actividad de **refinamiento de la base de datos** la incluimos en la siguiente fase y la explicamos en el Capítulo 16 en el contexto de la optimización de consultas.

### 12.2.6 Fase 6. Implementación y puesta a punto del sistema

Una vez completados los diseños lógico y físico, podemos implementar el sistema de bases de datos, una tarea que normalmente es responsabilidad del DBA y que lleva a cabo junto con los diseñadores de la base de datos. Para crear los esquemas de la base de datos y los ficheros (vacíos) de la misma, se utilizan las sentencias del lenguaje DDL (lenguaje de definición de datos), incluso del SDL (lenguaje de definición del almacenamiento), del DBMS seleccionado. Después de esto, ya podemos **cargarla** (rellenarla) con los datos. Si es necesario convertir éstos porque proceden de un sistema computerizado anterior, es posible que necesitemos **rutinas de conversión** para reformatear los datos y poder cargarlos en la base de datos nueva.

Las transacciones de bases de datos las deben implementar los programadores de la aplicación, refiriéndose a las especificaciones conceptuales de las transacciones, y escribiendo y probando después el código con los comandos DML incrustados. Una vez listas las transacciones y cargados los datos en la base de datos, la fase de diseño e implementación se da por terminada y empieza la fase operativa del sistema de bases de datos.

Casi todos los sistemas incluyen una utilidad de monitorización para recopilar estadísticas de rendimiento, que se almacenan en el catálogo del sistema o en el diccionario de datos para análisis posteriores. Estas estadísticas incluyen el número de invocaciones de las transacciones o consultas predefinidas, la actividad de entrada/salida contra los ficheros, el recuento de páginas de fichero o registros de índice, y la frecuencia de uso de los índices. A medida que cambian los requisitos del sistema de bases de datos, a menudo se hace nece-

sario añadir o eliminar tablas existentes, así como reorganizar algunos ficheros cambiando los métodos de acceso principal o eliminando los índices antiguos y construyendo otros nuevos. También se puede dar el caso de tener que reescribir algunas consultas o transacciones para mejorar el rendimiento. El refinamiento de la base de datos continúa mientras siga existiendo la base de datos, mientras se descubran problemas de rendimiento y mientras los requisitos sigan cambiando.

# 12.3 Uso de diagramas UML como ayuda a la especificación del diseño de la base de datos[6]

## 12.3.1 UML como un estándar para la especificación del diseño

En la primera sección de este capítulo explicamos cómo trabajan las empresas con los sistemas de información y elaboramos distintas actividades para el ciclo vital de los mismos. En la mayoría de las empresas, las bases de datos son parte integral de los sistemas de información. Las fases del diseño de la base, empezando con el análisis de los requisitos hasta la implementación y el refinamiento del sistema, se introdujeron al final de la Sección 12.1 y se explicaron en la Sección 12.2. La industria siempre necesita algunas metodologías estándar para cubrir todo este espectro de análisis de los requisitos, modelado, diseño, implementación e implantación. El método que está recibiendo más atención y aceptación y que también ha sido propuesto como estándar por el OMG (Grupo gestor de objetos, *Object Management Group*) es el método **UML** (**Lenguaje unificado de modelado**, *Unified Modeling Language*), que proporciona un mecanismo en forma de notación diagramática y sintaxis de lenguaje asociado para abarcar todo el ciclo vital. Actualmente, UML lo utilizan los desarrolladores de aplicaciones, los modeladores de datos, los diseñadores de datos, los arquitectos de bases de datos, etcétera, para definir la especificación detallada de una aplicación. Estos profesionales también lo utilizan para especificar el entorno consistente en software, comunicaciones y hardware para implementar e implantar la aplicación.

UML combina los conceptos comúnmente aceptados de muchas metodologías y métodos orientados a objetos (O-O) (consulte la bibliografía seleccionada si desea conocer las metodologías que han contribuido a llegar a UML). Es aplicable a cualquier dominio y es independiente del lenguaje y de la plataforma; de este modo y con UML, los arquitectos pueden modelar cualquier tipo de aplicación, en cualquier sistema operativo, lenguaje de programación o red. Esto ha logrado que la metodología sea ampliamente aplicable. Actualmente son muy populares herramientas como Rational Rose para dibujar los diagramas UML, pues permiten a los desarrolladores de aplicaciones desarrollar modelos limpios y fáciles de entender para especificar, visualizar, construir y documentar los componentes de los sistemas software. Como el ámbito de UML se ha extendido generalmente al desarrollo de software y aplicaciones, no veremos aquí todos sus aspectos. Nuestro objetivo es mostrar algunas notaciones UML relevantes que se utilizan con frecuencia en la recopilación y análisis de los requisitos, así como en las fases de diseño del esquema conceptual (consulte las Fases 1 y 2 de la Figura 12.1). Una explicación detallada de la metodología de desarrollo con UML queda fuera de los objetivos de este libro, pero puede encontrarla en diferentes libros dedicados al diseño orientado a objetos, la ingeniería de software y UML (consulte la bibliografía seleccionada).

Los diagramas de clases, que son el resultado final del diseño conceptual de la base de datos, se explicaron en las Secciones 3.8 y 4.6. Para llegar a dichos diagramas, la información puede recopilarse y especificarse utilizando diagramas de caso de uso, diagramas de *secuencia* y diagramas de *gráficas de estado* (o simplemente diagramas de estado). En el resto de esta sección haremos una introducción breve a los diferentes tipos de diagramas UML para que tenga una idea del ámbito de UML. Después, presentaremos una pequeña

[6] Apreciamos mucho la contribución de Abrar Ul-Haque a las secciones de UML y Rational Rose.

aplicación de ejemplo para ilustrar el uso de los diagramas de caso de uso, secuencia y estado, y mostrar cómo han conducido al diagrama de clases eventual como diseño conceptual final. Los diagramas presentados en esta sección pertenecen a la notación UML estándar y se han dibujado con Rational Rose. La Sección 12.4 estará dedicada a una explicación general del uso de Rational Rose en el diseño de una aplicación de base de datos.

### 12.3.2 UML para el diseño de una aplicación de base de datos

UML se desarrolló como una metodología de ingeniería de software. Como dijimos anteriormente en la Sección 3.8, la mayoría de los sistemas software tienen componentes de bases de datos grandes. La comunidad de bases de datos ha empezado a adoptar UML, y ahora algunos diseñadores y desarrolladores de bases de datos están utilizándolo para modelar los datos y para las subsiguientes fases del diseño de la base de datos. La ventaja de UML es que aunque sus conceptos están basados en técnicas de orientación a objetos, los modelos resultantes de estructura y comportamiento pueden utilizarse para diseñar bases de datos relacionales, orientadas a objetos y de objetos relacionales (en los Capítulos 20 a 22 encontrará las definiciones de todos estos términos). En las Secciones 3.8 y 4.6 ofrecimos una introducción a los **diagramas de clases** UML, que son parecidos a los diagramas ER y EER. Ofrecen una especificación estructural de los esquemas de bases de datos con un significado orientado a objetos, mostrando el nombre, los atributos y las operaciones de cada clase. Su uso normal es describir las colecciones de objetos de datos y sus interrelaciones, que son coherentes con el objetivo del diseño conceptual de la base de datos.

Una de las mayores contribuciones de la metodología UML ha sido juntar a los modeladores, analistas y diseñadores tradicionales de bases de datos con los desarrolladores de software. En la Figura 12.1 mostrábamos las fases del diseño y la implementación de una base de datos y cómo pueden aplicarse a estos dos grupos. UML ha podido proponer una notación común que ambas comunidades pueden adoptar y ajustar a sus necesidades. Aunque en los Capítulos 3 y 4 solamente hablamos del aspecto estructural del modelado, UML también nos permite realizar un modelado del comportamiento y/o dinámico mediante la introducción de varios tipos de diagramas. El resultado es una especificación/descripción más completa de la aplicación de base de datos en su conjunto. En las siguientes secciones resumiremos los diferentes diagramas UML y ofreceremos un ejemplo de cada uno en el entorno de una aplicación de muestra.

### 12.3.3 Diferentes diagramas en UML

UML define nueve tipos de diagramas divididos en dos categorías:

- **Diagramas estructurales.** Describen las relaciones estructurales o estáticas entre los componentes. Esta categoría está formada por el diagrama de clases, el diagrama de objetos, el diagrama de componentes y el diagrama de implantación.
- **Diagramas de comportamiento.** Su propósito es describir las relaciones de comportamiento o dinámicas entre los componentes. Esta categoría está compuesta por el diagrama de caso de uso, el diagrama de secuencia, el diagrama de colaboración, el diagrama de estado y el diagrama de actividad.

A continuación ofrecemos una introducción breve de los nueve tipos. Los diagramas estructurales son los siguientes:

**A. Diagramas de clases.** Los diagramas de clases capturan la estructura estática del sistema y actúan como base de otros modelos. Muestran las clases, las interfaces, las colaboraciones, las dependencias, las generalizaciones, las asociaciones y otras relaciones. Estos diagramas son muy útiles para modelar el esquema conceptual de la base de datos. En su momento mostramos ejemplos de los diagramas de clases para el esquema de la base de datos de la Figura 3.16 y para una jerarquía de generalización en la Figura 4.10.

Diagramas de paquetes. Estos diagramas son un subconjunto de los diagramas de clases. Organizan los elementos del sistema en grupos relacionados denominados paquetes. Un paquete puede ser una colección de

clases relacionadas y de las relaciones entre ellas. Los diagramas de paquetes ayudan a minimizar las dependencias en un sistema.

**B. Diagramas de objetos.** Los diagramas de objetos muestran un conjunto de objetos y sus relaciones. Corresponden con lo que en los Capítulos 3 y 4 habíamos denominado diagramas de instancia. Ofrecen una vista estática de un sistema en un momento dado y normalmente se utilizan para probar la exactitud de los diagramas de clases.

**C. Diagramas de componentes.** Los diagramas de componentes ilustran las organizaciones y las dependencias entre los componentes software. Un diagrama de componentes normalmente consta de componentes, interfaces y relaciones de dependencia. Un componente puede ser un componente de código fuente, un componente *runtime* o un componente ejecutable. Es un bloque constructivo físico del sistema y se representa como un rectángulo con dos pequeños rectángulos o etiquetas solapadas en su lado izquierdo. Una interfaz es un grupo de operaciones que un componente utiliza o crea, y normalmente se representa con un círculo pequeño. La relación de dependencia se utiliza para modelar la relación entre dos componentes y se representa con una flecha discontinua que apunta desde un componente al componente del que depende. En el caso de las bases de datos, los diagramas de componentes simbolizan los datos almacenados, como los espacios de tabla (*tablespaces*) o particiones. Las interfaces se refieren a las aplicaciones que utilizan los datos almacenados.

**D. Diagramas de implantación.** Los diagramas de implantación representan la distribución de componentes (ejecutables, librerías, tablas, ficheros) por la topología hardware. Representan los recursos físicos de un sistema, incluyendo nodos, componentes y conexiones, y se utilizan básicamente para mostrar la configuración de los elementos de procesamiento *runtime* (los nodos) y de los procesos software que residen en ellos (los hilos).

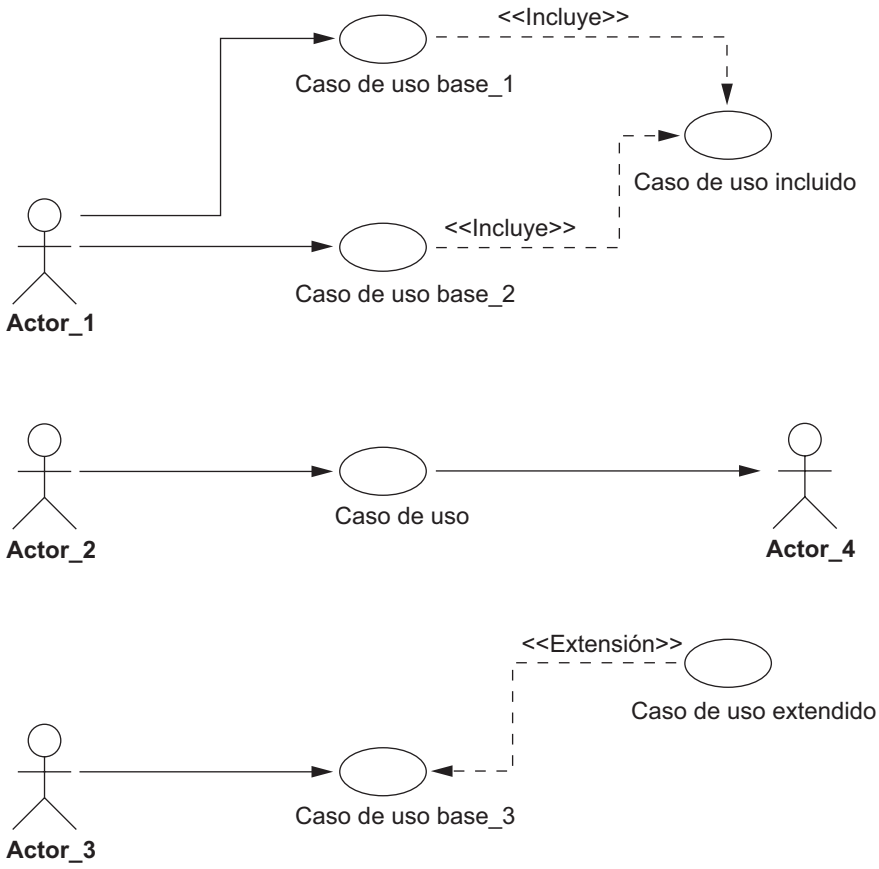
Ahora describiremos los diagramas conductistas y nos extenderemos un poco más en los más interesantes.

**E. Diagramas de casos de uso.** Estos diagramas se utilizan para modelar las interacciones funcionales entre los usuarios y el sistema. Un **escenario** es una secuencia de pasos que describen una interacción entre un usuario y un sistema. Un **caso de uso** es un conjunto de escenarios que tienen un objetivo común. El diagrama de caso de uso fue introducido por Jacobson[7] para visualizar los casos de uso. El **diagrama de caso de uso** muestra a los actores interactuando con casos de uso y pueden entenderse fácilmente sin conocer notación alguna. Un caso de uso individual se representa con un óvalo y simboliza una tarea específica llevada a cabo por el sistema. Un **actor**, mostrado con el símbolo de una persona, representa un usuario externo, que puede ser un ser humano, un grupo representativo de usuarios, un cierto rol de una persona de una empresa, o cualquier cosa externa al sistema (véase la Figura 12.7). El diagrama de caso de uso muestra las posibles interacciones del sistema (en nuestro caso, un sistema de bases de datos) y describe como casos de uso las tareas específicas que el sistema realiza. Como no incluyen ningún detalle de la implementación y son muy fáciles de entender, son un buen vehículo de comunicación entre los usuarios finales y los desarrolladores, a la vez que facilitan la validación de usuario en una etapa más temprana. Con los diagramas de casos de uso también se pueden generar fácilmente planes de prueba. La Figura 12.7 muestra la notación de este tipo de diagramas. La relación **incluye** se utiliza para factorizar algún comportamiento común a partir de dos o más de los casos de uso originales; es una forma de reutilización. Por ejemplo, en un entorno universitario como el de la Figura 12.8, los casos de uso *registro para cursos* e *introducir notas* en los que los actores estudiante y profesor están implicados, incluyen un caso de uso común denominado *validar usuario*. Si un caso de uso incorpora dos o más escenarios significativamente diferentes, basados en circunstancias o condiciones variables, se utiliza la relación de **extensión** para mostrar los subcasos adjuntos al caso de uso.

---

[7] Consulte Jacobsen y otros (1992).

**Figura 12.7.** Notación de un diagrama de caso de uso.




Diagramas de interacción. Estos diagramas se utilizan para modelar los aspectos dinámicos de un sistema. Consisten en un conjunto de mensajes intercambiados entre un conjunto de objetos. Hay dos tipos de diagramas de interacción: de secuencia y de colaboración.

**F. Diagramas de secuencia.** Los diagramas de secuencia describen las interacciones entre varios objetos en el transcurso del tiempo. Básicamente, ofrecen una vista dinámica del sistema mostrando el flujo de mensajes entre los objetos. Dentro del diagrama de secuencia, un objeto o un actor se muestra como un recuadro en la parte superior de una línea vertical discontinua, que se denomina línea de vida (*lifeline*) del objeto. En una base de datos, este objeto es normalmente algo físico (como un libro en una biblioteca) que se podría almacenar en la base de datos, un documento o formulario externo (por ejemplo, un formulario de pedido), o una pantalla visual externa, que puede formar parte de una interfaz de usuario. La línea de vida representa la existencia de un objeto en el tiempo. La activación, que indica cuándo un objeto está realizando una acción, se representa con un rectángulo en una línea de vida. Los mensajes se representan como una flecha entre las líneas de vida de dos objetos. Un mensaje lleva un nombre y puede tener argumentos e información de control para explicar la naturaleza de la interacción. El orden de los mensajes es de arriba abajo. Un diagrama de secuencia también ofrece la opción de la autollamada, que es básicamente un mensaje de un objeto a sí mismo. En los diagramas de secuencia también pueden mostrarse **marcadores de condición** e **interacción** para especificar cuándo ha de enviarse el mensaje, así como la condición para enviar varios marcadores. Una línea de retorno discontinua muestra un retorno desde el mensaje y es opcional a menos que transporte un significado

**Figura 12.8.** Ejemplo de un diagrama de caso de uso para la base de datos UNIVERSIDAD.

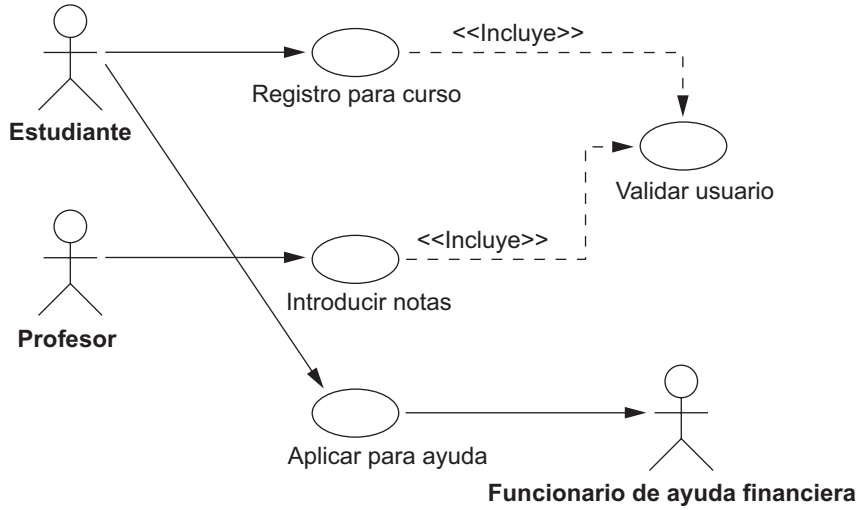
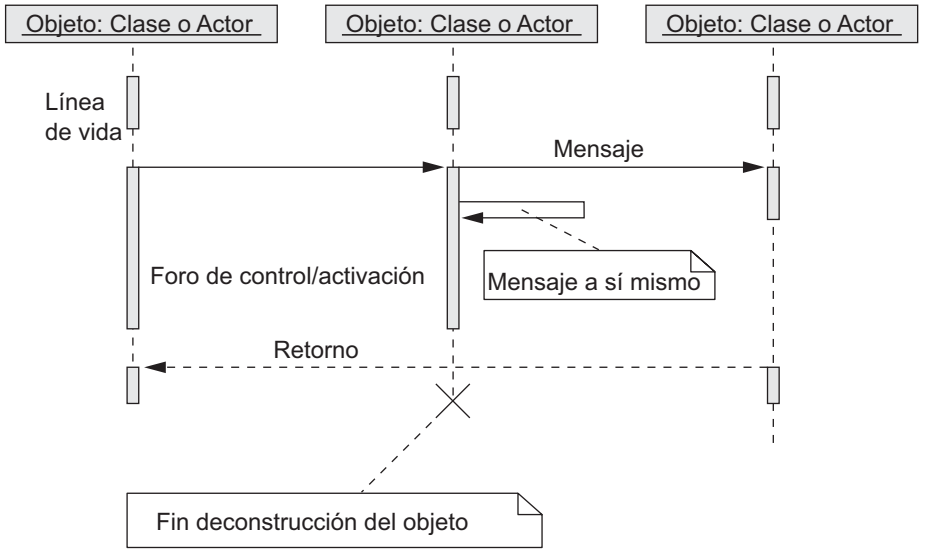


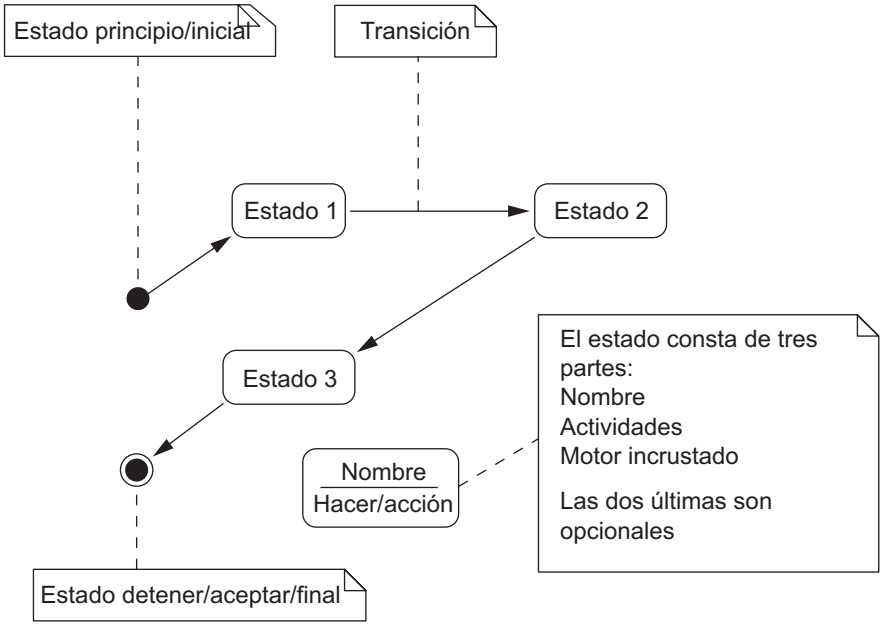

**Figura 12.9.** Notación del diagrama de secuencia.




especial. La eliminación de un objeto se muestra con una X grande. La Figura 12.9 explica parte de la notación de un diagrama de secuencia.

**G. Diagramas de colaboración.** Los diagramas de colaboración representan las interacciones entre objetos como una serie de mensajes en secuencia. En estos diagramas se centra la atención en la organización estructural de los objetos que envían y reciben mensajes, mientras que en los diagramas de secuencia la atención se centra en la ordenación de los mensajes en el tiempo. Los diagramas de colaboración muestran los objetos como iconos y numeran los mensajes; los mensajes numerados representan un orden. La disposición espacial de los diagramas de colaboración permiten enlaces entre los objetos, que muestran sus relaciones estructurales. El uso de diagramas de colaboración y secuencia para representar las interacciones es una cuestión de preferencias; a partir de ahora sólo utilizaremos los diagramas de secuencia.

**Figura 12.10.** Notación de un diagrama de estado.




**H. Diagramas de estado.** Estos diagramas describen cómo cambia el estado de un objeto en respuesta a eventos externos.

Para describir el comportamiento de un objeto, es habitual en la mayoría de las técnicas orientadas a objetos dibujar un diagrama de estado para mostrar todos los estados posibles que un objeto puede tener en su vida. Las gráficas de estado UML están basadas en las gráficas de estado de David Harel[8]. Muestran un mecanismo de estado consistente en estados, transiciones, eventos y acciones, y resulta muy útil en el diseño conceptual de la aplicación que trabaja contra la base de datos de objetos almacenados.

Los elementos importantes de un diagrama de estado que se muestran en la Figura 12.10 son los siguientes:

- **Estados**. Se muestran como cajas de bordes redondeados, y representan situaciones de la vida de un objeto.
- **Transiciones**. Se muestran como flechas sólidas entre los estados. Representan las rutas entre los diferentes estados de un objeto. Están etiquetadas con el nombre del evento[defensa]/acción; el evento lanza la transición y la acción es el resultado de ella. La "defensa" es una condición adicional y opcional que especifica una condición bajo la que no puede darse el cambio de estado.
- **Estado principio/inicial**. Se muestra como un círculo sólido con una flecha de salida hacia un estado.
- **Estado detener/final**. Se muestra como un círculo relleno de borde doble con una flecha apuntando a él desde un estado.

Los diagramas de estado son útiles para especificar cómo la reacción de un objeto a un mensaje depende de su estado. Un evento es algo que se le hace a un objeto, como enviarle un mensaje; una acción es algo que un objeto hace, como enviar un mensaje.

**I. Diagramas de actividad.** Los diagramas de actividad presentan una vista dinámica del sistema modelando el flujo de control de actividad a actividad. Los podemos considerar como diagramas de flujo con los

[8] Consulte Harel (1987).

estados. Una actividad es un estado de hacer algo que podría ser un proceso real o una operación en alguna clase de la base de datos. Normalmente, los diagramas de actividad se utilizan para modelar el flujo de trabajo y las operaciones comerciales internas para una aplicación.

### 12.3.4 Ejemplo de modelado y diseño: base de datos UNIVERSIDAD

En esta sección ilustraremos brevemente el uso de los diagramas UML que presentamos anteriormente, al objeto de diseñar un ejemplo de base de datos relacional para un entorno universitario. Hemos omitido muchos detalles para poder ahorrar espacio; únicamente ilustramos el uso por etapas de estos diagramas que conducen a un diseño conceptual y al diseño de los componentes del programa. Como indicamos anteriormente, el DBMS eventual sobre el que puede implementarse esta base de datos puede ser relacional, de objetos relacionales, u orientado a objetos. Esto no cambiará el análisis por etapas y el modelado de la aplicación al utilizar los diagramas UML.

Imagine un escenario con estudiantes matriculados en cursos, que son impartidos por profesores. La oficina de admisión está al cargo de mantener una planificación de los cursos en un catálogo; tiene autoridad para añadir y borrar cursos, y para introducir cambios. También establece los límites de matriculación en los cursos. La oficina de ayuda financiera procesa las aplicaciones de ayuda al estudiante. Asumimos que tenemos que diseñar una base de datos que nos sirva para mantener los datos de los estudiantes, los profesores, los cursos, las ayudas, etcétera. También queremos diseñar las aplicaciones que nos permitan realizar la matriculación en un curso, procesar la aplicación de ayuda financiera y mantener un catálogo de cursos de toda la universidad por parte de la oficina de admisión. Los requisitos anteriores pueden plasmarse en una serie de diagramas UML que mostramos más adelante.

Como mencionamos anteriormente, uno de los primeros pasos del diseño de una base de datos es la recopilación de los requisitos del cliente, y la mejor forma de hacerlo es con los diagramas de caso de uso. Supongamos que uno de los requisitos de la base de datos de la universidad es permitir que los profesores introduzcan las notas de los cursos que están impartiendo, y que los estudiantes puedan matricularse en cursos y solicitar una ayuda financiera. El diagrama de caso de uso correspondiente a estos casos de uso se puede dibujar como muestra la Figura 12.8.

**Figura 12.11.** Ejemplo de diagrama de estado para la base de datos UNIVERSIDAD.

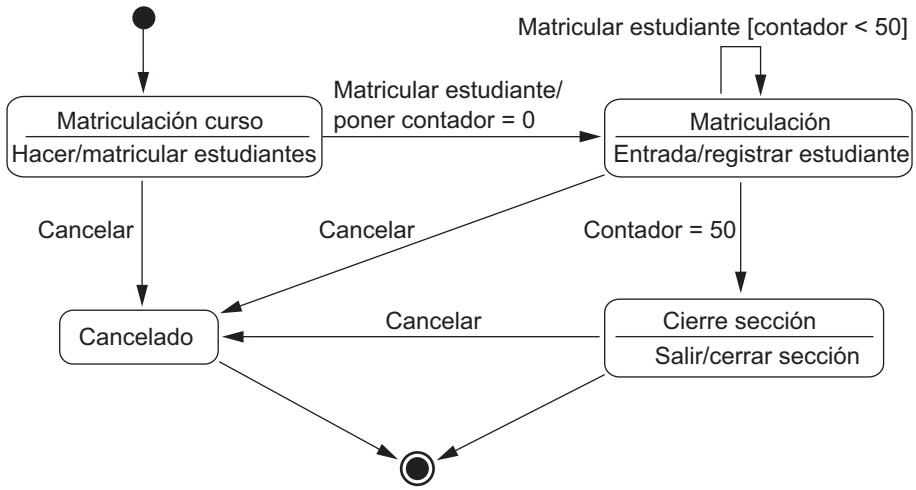
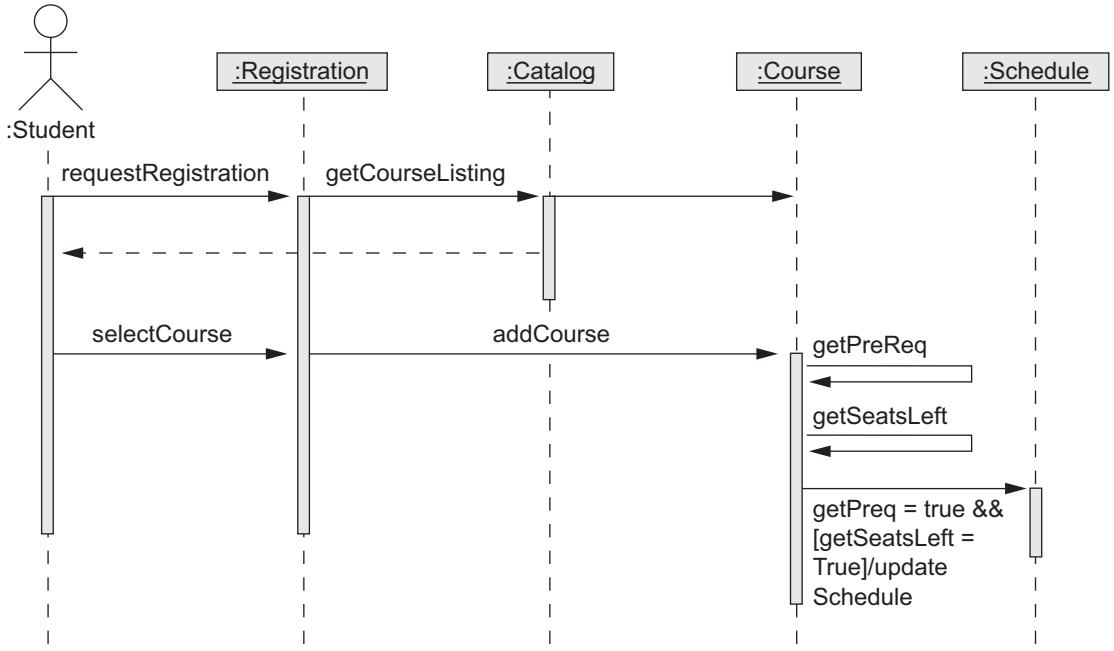


**Figura 12.12.** Diagrama de secuencia para la base de datos UNIVERSIDAD.

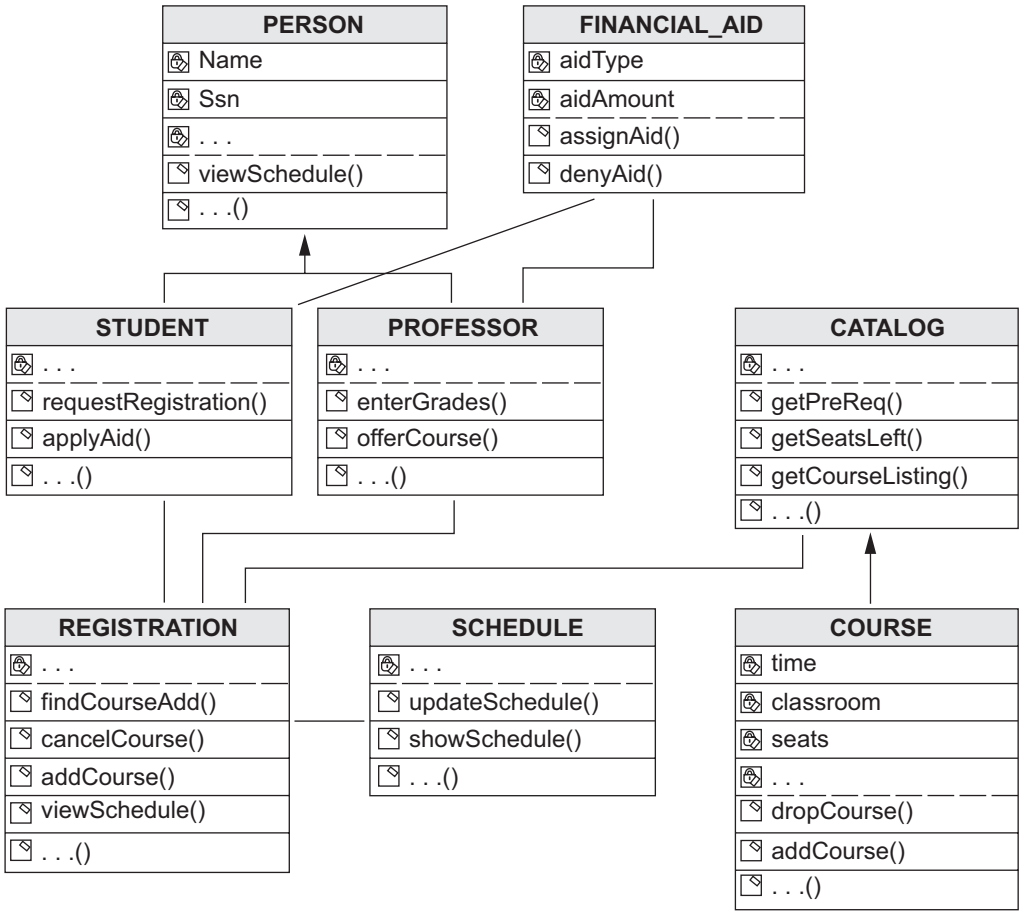



Otro elemento que resulta útil al diseñar un sistema es la representación gráfica de algunos de los estados que ese sistema puede tener durante el curso de la aplicación. Por ejemplo, en nuestra base de datos, los distintosestados por los que el sistema puede pasar durante la matriculación en un curso que se abre con 50 plazas se pueden representar con el diagrama de estado de la Figura 12.11; muestra los estados de un curso cuando la matriculación está en curso. Durante el estado de matriculación, la transición *Matricular estudiante* continúa mientras el contador de estudiantes matriculados sea inferior a 50.

Habiendo realizado el caso de uso y la gráfica de estado, podemos confeccionar el diagrama de secuencia para visualizar la ejecución de los casos de uso. En la base de datos que nos ocupa, este diagrama corresponde al caso de uso "*el estudiante solicita matricularse y selecciona un curso en particular*" (véase la Figura 12.12). Se comprueban entonces los prerrequisitos y la capacidad del curso, y después se añade éste al programa de estudios del estudiante, siempre y cuando se satisfagan los prerrequisitos y haya plaza en el curso.

Los diagramas UML anteriores no son la especificación completa de la base de datos Universidad. Habrá otros casos de uso con el registrador como actor, o con el estudiante haciendo una prueba para un curso y recibiendo una calificación para el mismo, etcétera. No es objetivo de este libro una metodología completa de cómo llegar a los diagramas de clases a partir de los diversos diagramas anteriormente ilustrados. Las metodologías de diseño siguen siendo una cuestión de sentido común, preferencias personales, etcétera. No obstante, podemos asegurarnos de que el diagrama de clases responderá a todas las especificaciones reseñadas en forma de casos de uso, gráficas de estado y diagramas de secuencia. El diagrama de clases de la Figura 12.13 muestra las clases con las relaciones estructurales y las operaciones dentro de las clases que se derivan de esos diagramas. Estas clases habrá que implementarlas para desarrollar la base de datos Universidad y, junto con las operaciones, implementarán la aplicación "planificación de clases/matriculación/ayuda" completa. Para no complicar la explicación, sólo mostramos algunos de los atributos más importantes de las clases, con determinados métodos que se originan a partir de los diagramas mostrados. Es concebible que estos diagramas de clases se actualicen constantemente, a medida que se especifiquen más detalles y surjan más funciones propias de la evolución de la aplicación.

**Figura 12.13.** Diseño de la base de datos UNIVERSIDAD como diagrama de clases.




# 12.4 Rational Rose, una herramienta de diseño basada en UML

## 12.4.1 Rational Rose para el diseño de bases de datos

Rational Rose es una de las herramientas de modelado más importantes del mercado para el desarrollo de sistemas de información. Como apuntamos en las dos primeras secciones de este capítulo, la base de datos es un componente central de la mayoría de los sistemas de información y, por tanto, Rational Rose proporciona la especificación inicial en UML que, con el tiempo, lleva al desarrollo de la base de datos. En las últimas versiones de Rose se han incorporado muchas extensiones para modelar datos, y ahora proporciona soporte para el modelado y el diseño conceptual, lógico y físico de la base de datos.

## 12.4.2 Rational Rose Data Modeler

Rational Rose Data Modeler es una herramienta de modelado visual para diseñar bases de datos. Una de las razones de su popularidad es que, a diferencia de otras herramientas de modelado de datos, está basada en UML; ofrece una herramienta común y un lenguaje a modo de puente de comunicación entre los diseñadores

de la base de datos y los desarrolladores de la aplicación. Hace posible que los diseñadores, desarrolladores y analistas de bases de datos trabajen juntos, recopilen y compartan los requisitos comerciales, y hagan un seguimiento de los mismos durante todo el proceso. Además, al permitir que los diseñadores modelen y diseñen todas las especificaciones en la misma plataforma utilizando la misma notación, mejora el proceso de diseño y reduce el riego de tener errores.

Otra ventaja importante de Rose son sus capacidades de modelado de procesos que permiten el modelado del comportamiento de las aplicaciones de bases de datos como vimos anteriormente con el pequeño ejemplo de los distintos tipos de diagramas. Hay más diagramas de colaboración que muestran las interacciones entre los objetos y diagramas de actividad para modelar el flujo de control que nosotros no elaboramos. El objetivo es generar la especificación de la base de datos y el código de la aplicación en la medida de lo posible. Con Rose Data Modeler podemos capturar *triggers*, procedimientos almacenados, etcétera (consulte el Capítulo 24, ya que las bases de datos activas tienen estas características) explícitamente en el diagrama, en lugar de representarlos con valores etiquetados ocultos. Data Modeler también proporciona la capacidad de enviar al ingeniero una base de datos en términos de requisitos constantemente cambiantes y devolverle una base de datos implementada existente en su diseño conceptual.

### 12.4.3 Modelado de datos con Rational Rose Data Modeler

Rose Data Modeler tiene muchas herramientas y opciones para modelar datos. Permite crear un modelo de datos basándose en la estructura de la base de datos o crear una base de datos basándose en el modelo de datos.

**Ingeniería inversa.** La ingeniería inversa de la base de datos permite crear un modelo de datos basándose en la estructura de una base de datos. Si ya tenemos una base de datos en un DBMS o un fichero DDL, podemos recurrir al asistente de ingeniería inversa de Rational Rose Data Modeler para generar un modelo de datos conceptual. El asistente de ingeniería inversa básicamente lee el esquema de la base de datos o el fichero DDL y lo recrea en un modelo de datos. Al hacerlo, también incluye los nombres de todas las entidades.

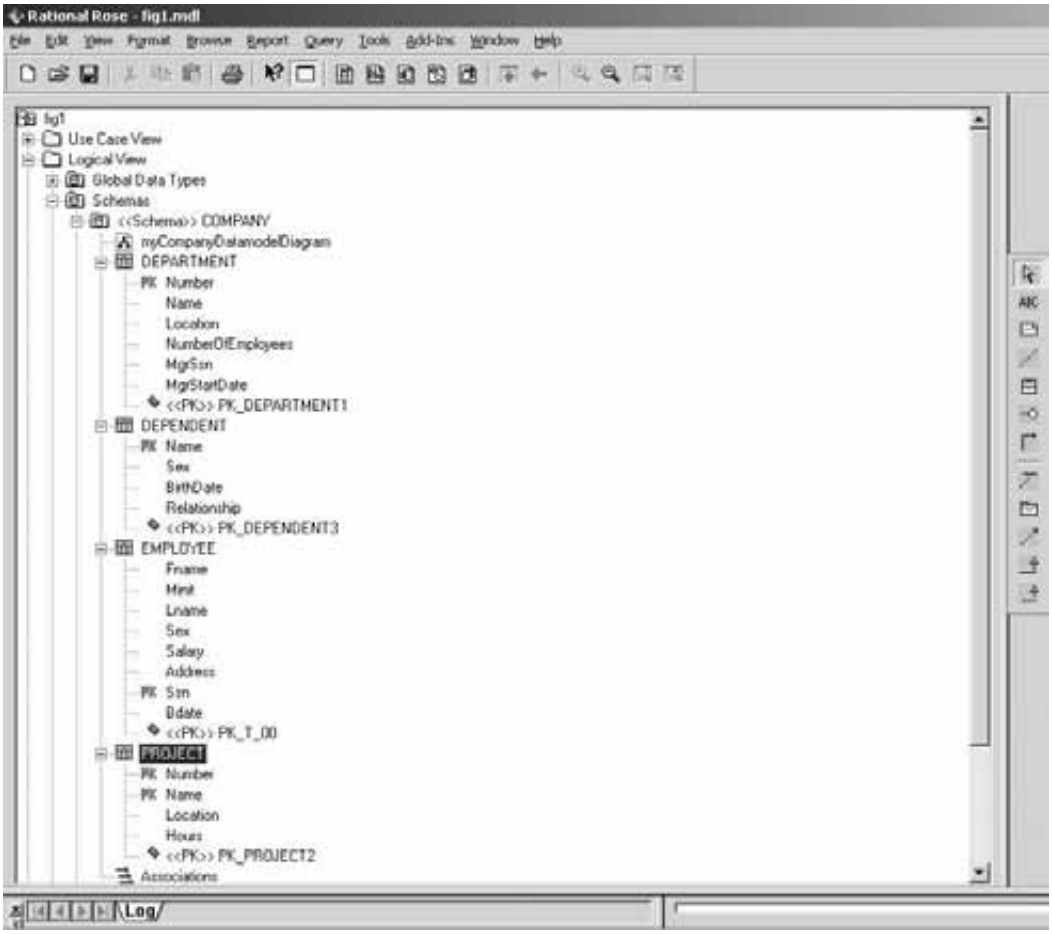
**Ingeniería directa y generación DDL.** En Rose también podemos crear desde el principio un modelo de datos. Habiendo creado el modelo de datos[9] también podemos utilizarlo para generar el DDL en un DBMS específico a partir de dicho modelo. En Modeler hay un asistente de ingeniería directa, que lee el esquema del modelo de datos, o lee tanto el esquema del modelo de datos como los espacios de tabla del modelo de almacenamiento de datos, y genera el código DDL apropiado en un fichero DDL. El asistente también ofrece la opción de generar una base de datos ejecutando el fichero DDL generado.

**Diseño conceptual en notación UML.** Una de las principales ventajas de Rose es el modelado de bases de datos con notación UML. En Rational Rose, los diagramas ER que con más frecuencia se utilizan en el diseño conceptual de bases de datos se pueden construir fácilmente en notación UML como diagramas de clases; por ejemplo, el esquema ER de nuestra empresa de ejemplo del Capítulo 3 se puede trazar en Rose con notación UML como muestra la Figura 12.14. La especificación textual de la Figura 12.14 puede convertirse en la representación gráfica de la Figura 12.15 utilizando la opción de diagrama de modelo de datos de Rose.

Los diagramas anteriores corresponden en parte al esquema relacional (lógico), aunque están a un nivel conceptual. Muestran las relaciones entre las tablas mediante relaciones del tipo clave principal (PK)–*foreign key* (FK). Las **relaciones identificadas** especifican que una tabla hija no puede existir sin la tabla padre (tablas dependientes), mientras que las **no identificadas** especifican una asociación regular entre dos tablas independientes. Es posible actualizar directamente los esquemas en su forma textual o gráfica. Por ejemplo, es posible borrar la relación entre EMPLEADO y PROYECTO denominada TRABAJA_EN y Rose se preocupará automáticamente de todas las *foreign keys*, etcétera, de la tabla.

[9] El término modelo de datos utilizado por Rational Rose Modeler corresponde a la noción que tenemos de un modelo de aplicación o esquema conceptual.

**Figura 12.14.** Definición de un diagrama de un modelo de datos lógico en Rational Rose.



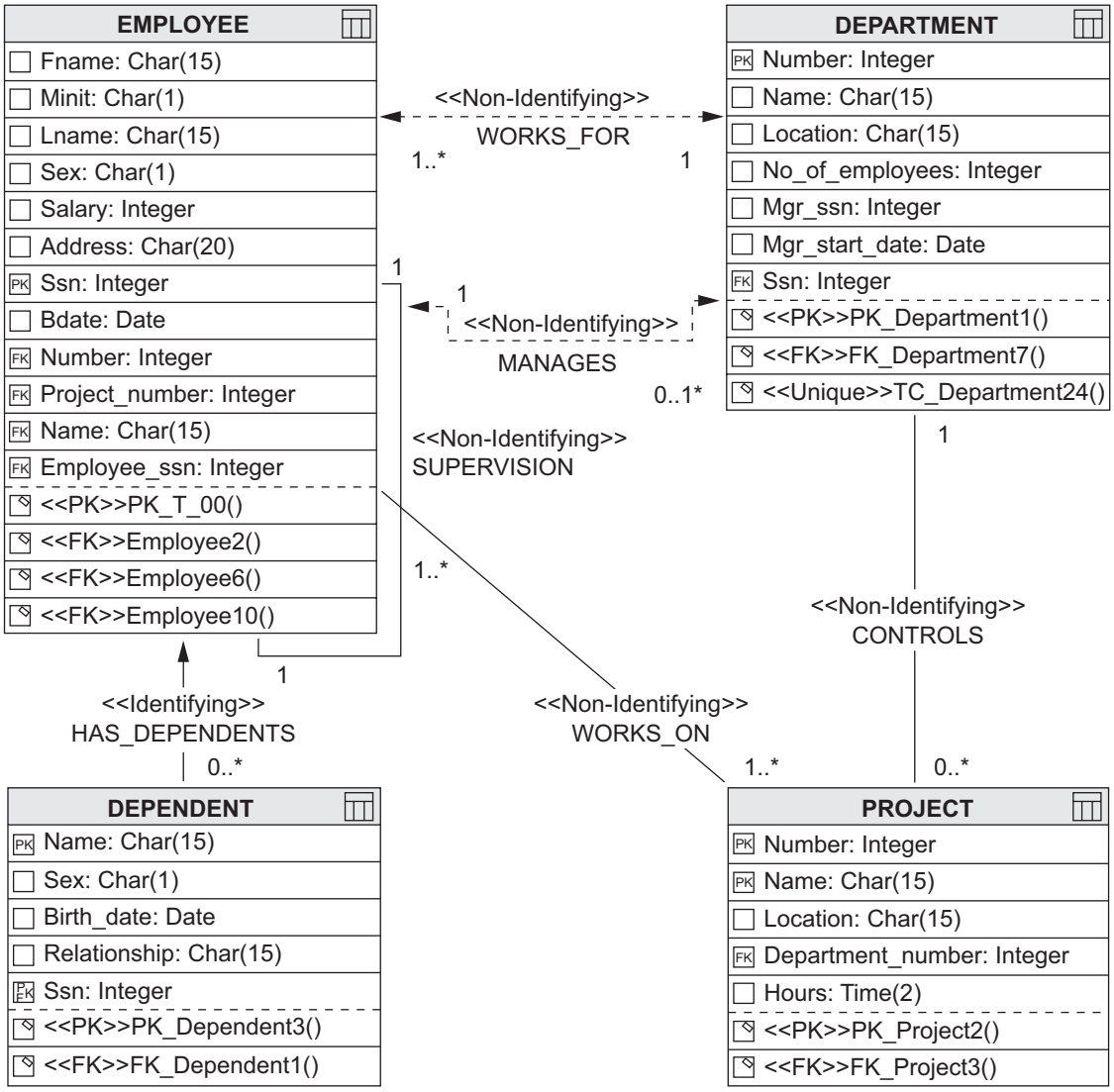

Una diferencia importante entre la Figura 12.15 y la notación que ofrecimos en los Capítulos 3 y 4 es que los atributos de *foreign key* realmente aparecen en los diagramas de clases. Esto es común en varias notaciones diagramáticas para conseguir que el diseño conceptual se aproxime más a cómo se materializa en la implementación del modelo relacional. En los Capítulos 3 y 4, los diagramas ER y EER conceptuales y los diagramas de clases UML no incluían los atributos de *foreign key*, que se añadían al esquema relacional durante el proceso de mapeo (consulte el Capítulo 7).

**Bases de datos soportadas.** Algunos de los DBMSs que actualmente son soportados por Rational Rose son los siguientes:

- IBM DB2 versiones MVS y UDB 5.x, 6.x, and 7.0.
- Oracle DBMS versiones 8.x, 9.x, and 10.x.
- SQL Server DBMS versiones 6.5, 7.0 y 2000.
- Sybase Adaptive Server versión 12.x.

SQL 92 Data Modeler no soporta la ingeniería inversa de los DDLs ANSI SQL 92; no obstante, puede aplicar la ingeniería directa de los modelos de datos SQL 92 a los DDLs.

**Figura 12.15.** Diagrama gráfico de un modelo de datos en Rational Rose para la base de datos EMPRESA.




**Conversión del modelo de datos lógico al modelo de objetos, y viceversa.** Rational Rose Data Modeler también ofrece la opción de convertir un diseño lógico de base de datos en un diseño de modelo de objetos, y viceversa. Por ejemplo, el modelo de datos lógico de la Figura 12.14 puede convertirse en un modelo de objetos. Este tipo de mapeo permite un entendimiento más profundo de las relaciones entre el modelo lógico y la base de datos, y ayuda a mantener actualizados los dos con los cambios introducidos durante el proceso de desarrollo. La Figura 12.16 muestra la tabla EMPLEADO después de convertirla en una clase del modelo de objetos. Las distintas fichas de la ventana permiten introducir/visualizar distintos tipos de información: operaciones, atributos y relaciones para la clase en cuestión.

**Sincronización entre el diseño conceptual y la base de datos real.** Rose Data Modeler permite mantener sincronizados el modelo de datos y la base de datos. También permite visualizar los dos y después, basándose en las diferencias, ofrece la opción de actualizar el modelo o cambiar la base de datos.

**Figura 12.16.** La clase OM_EMPLOYEE correspondiente a la tabla EMPLEADO de la Figura 12.14.

**Soporte de dominio amplio.** Con Data Modeler, los diseñadores de bases de datos pueden crear un conjunto estándar de tipos de datos definidos por el usuario y asignarlos a cualquier columna del modelo de datos. Las propiedades del dominio se aplican en cascada a las columnas asignadas. Estos dominios pueden mantenerse después con un grupo de estándares e implantarse para todos los modeladores que empiezan a crear modelos nuevos utilizando la estructura de Rational Rose.

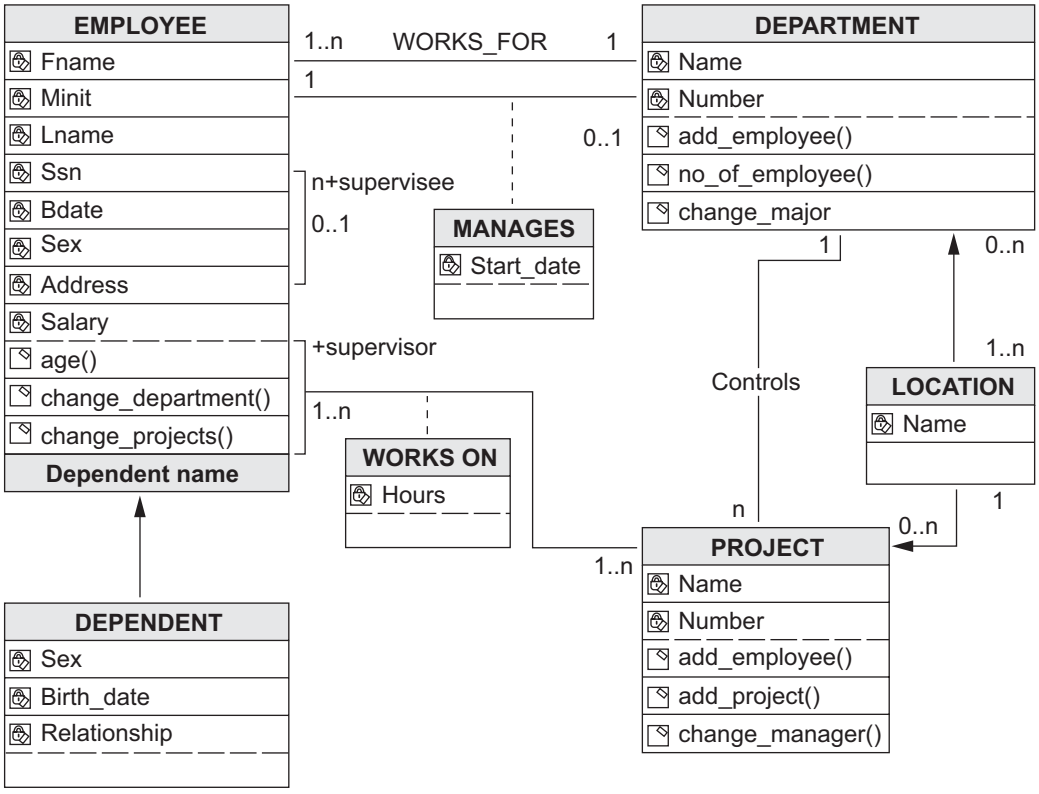
**Comunicación sencilla entre los equipos de diseño.** Como mencionamos anteriormente, el uso de una herramienta común facilita la comunicación entre los equipos. En Data Modeler, el desarrollador de una aplicación puede acceder a los modelos de objetos y de datos y ver cómo se relacionan, lo que le permitirá estar mejor informado y elegir, por tanto, las mejores opciones sobre cómo construir los métodos de acceso a los datos. También existe la opción de utilizar *Rational Rose Web Publisher* para que los modelos y los metadatos de estos modelos estén disponibles para todos los integrantes del equipo.

Lo que acabamos de ofrecer es una descripción parcial de las capacidades de la herramienta en relación con las fases de diseño conceptual y lógico de la Figura 12.1. En Rose se pueden desarrollar y mantener todos los diagramas UML que se describieron en la Sección 12.3. Si desea información más detallada sobre un producto, puede consultar su documentación. La Figura 12.17 ofrece otra versión del diagrama de clases de la Figura 3.16 utilizando Rational Rose. La Figura 12.17 difiere de la Figura 12.15 en que los atributos de la *foreign key* no se muestran explícitamente. Por tanto, la Figura 12.17 está utilizando las notaciones presentadas en los Capítulos 3 y 4.

## 12.5 Herramientas automáticas de diseño de bases de datos

La actividad de diseño de la base de datos se extiende principalmente por la Fase 2 (diseño conceptual), la Fase 4 (mapeo del modelo de datos, o diseño lógico) y la Fase 5 (diseño físico de la base de datos) en el proceso de diseño que explicamos en la Sección 12.2. La explicación de la Fase 5 la aplazamos hasta el Capítulo 16, cuando la veamos en el contexto de la optimización de consultas. En la Sección 12.3 explicamos en profundidad las Fases 2 a 4 haciendo uso de la notación UML, y en la Sección 12.4 mencionamos algunas de las

**Figura 12.17.** Diagrama de clases de la base de datos EMPRESA (Figura 3.16) trazado en Rational Rose.




características de la herramienta Rational Rose, que soporta estas fases. Como mencionamos, Rational Rose es más que una simple herramienta de diseño de bases de datos. Es una herramienta de desarrollo de software y convierte el modelado de bases de datos y el diseño de esquemas en forma de diagramas de clases en parte de su metodología de desarrollo de aplicaciones orientado a objetos. En esta sección resumimos las características y los defectos del conjunto de herramientas comerciales destinadas a automatizar el proceso del diseño conceptual, lógico y físico de las bases de datos.

Cuando se introdujo por primera vez la metodología de bases de datos, la mayor parte del diseño la realizaban a mano expertos diseñadores con amplia experiencia y conocimiento del proceso de diseño. No obstante, hay dos factores que indican que debieron utilizar en lo posible alguna forma de automatización:

1. A medida que la aplicación implica más y más complejidad de los datos en términos de relaciones y restricciones, aumenta rápidamente el número de opciones o de diferentes diseños para modelar la misma información. Se hace muy difícil tratar manualmente esta complejidad y las correspondientes alternativas de diseño.
2. El tamaño total de algunas bases de datos topa con cientos de tipos de entidades y tipos de relaciones, haciendo casi imposible la tarea de realizar manualmente estos diseños. La metainformación relacionada con el proceso de diseño que describimos en la Sección 12.2 produce otra base de datos que debe crearse, mantenerse y consultarse como una base de datos.

Los factores anteriores han dado lugar a muchas herramientas que se agrupan bajo la categoría general de herramientas CASE (Ingeniería de software asistida por computador, *Computer-Aided Software Engineering*) para el diseño de bases de datos. Rational Rose es un buen ejemplo de herramienta CASE moderna. Normalmente, estas herramientas consisten en una combinación de los siguientes servicios:

1. **Diagramación**. Permite al diseñador trazar un diagrama del esquema conceptual, en alguna notación específica de la herramienta. La mayoría de las notaciones incluyen tipos de entidades, tipos de relaciones que se muestran como recuadros separados o simplemente como líneas con o sin dirección, restricciones de la cardinalidad mostradas al lado de las líneas o en términos de los diferentes tipos de puntas de flecha o restricciones mín/máx, atributos, claves, etcétera.[10] Algunas herramientas muestran las jerarquías de herencia y utilizan una notación adicional para mostrar la naturaleza parcial frente a total y disjunta frente a solapada de las generalizaciones. Los diagramas se almacenan internamente como diseños conceptuales y están disponibles para la modificación, así como para la generación de informes, los listados de referencia cruzada, y otros usos.
2. **Mapeo de modelo**. Implementa unos algoritmos de mapeo parecidos a los presentados en las Secciones 7.1 y 7.2. El mapeo es específico del sistema: la mayoría de las herramientas generan esquemas en SQL DDL para Oracle, DB2, Informix, Sybase y otros RDBMSs. Esta parte de la herramienta es más sensible a la automatización. Si es necesario, el diseñador puede editar los ficheros DDL producidos.
3. **Normalización del diseño**. Esto utiliza un conjunto de dependencias funcionales que se proporcionan en el diseño conceptual o después de haber producido los esquemas relacionales durante el diseño lógico. Los algoritmos de descomposición del diseño del Capítulo 11 se aplican para descomponer las relaciones existentes normales superiores. Normalmente, las herramientas carecen de la metodología para generar diseños 3NF o BCNF alternativos y permitir al diseñador seleccionar entre ellos basándose en algunos criterios como el número mínimo de relaciones o la menor cantidad de almacenamiento.

La mayoría de las herramientas incorporan alguna forma de diseño físico, incluyendo la opción de los índices. Existe un completo conjunto de herramientas para monitorizar y medir el rendimiento. El problema de refinar un diseño o la implementación de la base de datos sigue siendo todavía una actividad humana de toma de decisiones. Fuera de las fases de diseño descritas en este capítulo, un área donde apenas hay herramientas comerciales de apoyo es la integración de vistas (consulte la Sección 12.2.2).

No vamos a estudiar aquí las herramientas de diseño de bases de datos, sino que únicamente mencionaremos las características que debe poseer una buena herramienta de diseño:

1. **Una interfaz fácil de usar**. Esta característica es crítica porque permite que los diseñadores se centren en la tarea que tienen entre manos, y no en conocer la herramienta. Las interfaces que más se utilizan son las gráficas y del tipo "apuntar y clic". Unas cuantas herramientas, como SECSI, utilizan como entrada el lenguaje natural. Es posible personalizar diferentes interfaces para principiantes o para expertos.
2. **Componentes analíticos**. Las herramientas deben proporcionar componentes analíticos para las tareas que son difíciles de realizar a mano, como evaluar las alternativas de diseño físico o detectar las restricciones en conflicto entre las vistas. Esta área es más bien débil en la mayoría de las herramientas actuales.
3. **Componentes heurísticos**. Los aspectos del diseño que no se pueden cuantificar con exactitud pueden automatizarse introduciendo reglas heurísticas en la herramienta de diseño para evaluar los diseños alternativos.
4. **Análisis de contrapartidas**. Una herramienta debe presentar al diseñador el análisis comparativo adecuado si presenta varias alternativas entre las que elegir. Es recomendable que las herramientas incorporen un análisis de la modificación de un diseño a nivel del diseño conceptual hasta el nivel físico. Debido a las muchas alternativas posibles de diseño físico en un sistema dado, es difícil incorporar una analítica semejante y la mayoría de las herramientas actuales carecen de ella.

[10] Las notaciones de los diagramas de clases ER, EER y UML las vimos en los Capítulos 3 y 4. Consulte el Apéndice A para hacerse una idea de los diferentes tipos de notaciones diagramáticas utilizados.

5. **Visualización del resultado del diseño**. El resultado de un diseño, como ocurre con los esquemas, a menudo se muestra en forma de diagrama. No es fácil generar automáticamente diagramas estéticamente agradables y bien compuestos. Las composiciones de diseño multipágina que sean fáciles de leer son otro reto. Otros tipos de resultados del diseño se pueden mostrar como tablas, listas o informes que se puedan interpretar fácilmente.
6. **Verificación del diseño**. Es una característica altamente recomendable. Su propósito es comprobar que el diseño resultante satisface los requisitos iniciales. A menos que los requisitos se capturen y representen internamente de alguna forma analizable, la verificación no podrá intentarse.

Actualmente, hay un conocimiento creciente del valor de las herramientas de diseño, que se están haciendo necesarias para tratar con los problemas de diseño de las bases de datos más grandes. También hay una percepción creciente de que el diseño del esquema y de la aplicación deben ir de la mano, y la tendencia actual entre las herramientas CASE es abarcar las dos áreas. La popularidad de Rational Rose se debe al hecho de que acerca los dos brazos del proceso de diseño mostrados en la Figura 12.1 al mismo tiempo, abordando el diseño de la base de datos y el diseño de la aplicación como una actividad unificada. Después de que IBM adquiriera Rational en 2003, la suite de herramientas Rational se ha mejorado como herramientas XDE (entorno de desarrollo extendido). Algunos fabricantes, como Platinum (Computer Associates), proporcionan una herramienta para el modelado de datos y el diseño de esquemas (ERWin) y otra para el modelado de procesos y el diseño funcional (BPWin). Otras herramientas (por ejemplo, SECSI) utilizan un sistema experto para guiarnos por el proceso de diseño mediante reglas. La tecnología de sistemas expertos también resulta útil en

**Tabla 12.1.** Algunas de las herramientas automáticas de diseño de bases de datos actualmente disponibles.

| Empresa | Herramienta | Funcionalidad |
|---|---|---|
| Embarcadero Technologies | ER Studio<br>DB Artisan | Modelado de la base de datps en ER e IDEF1x.<br>Administración de bases de datos, y administración del espacio y la seguridad. |
| Oracle | Developer 2000 y Designer 2000 | Modelado de bases de datos, desarrollo de aplicaciones. |
| Popkin Software | Telelogic System Architect | Modelado de datos, modelado de objetos, modelado de procesos, diseño/análisis estructurado. |
| Platinum Technology (Computer Associates) | Platinum Modelmart, ERwin, BPWin, AllFusion Component Modeler | Modelado de datos, procesos y componentes comerciales. |
| Persistence Inc. | Powertier | Mapeo de modelo O-O a relacional. |
| Rational (IBM) | Rational Rose XDE Developer Plus | Modelado en UML y generación de aplicaciones en C++ y Java. |
| Resolution Ltd. | XCase | Modelado conceptual hasta el mantenimiento del código. |
| Sybase | Enterprise Application Suite | Modelado de datos, modelado lógico comercial. |
| Visio | Visio Enterprise | Modelado de datos, diseño y reingeniería en Visual Basic y Visual C++. |

la fase de recopilación y análisis de requisitos, que normalmente es un proceso laborioso y frustrante. La tendencia es utilizar almacenes de metadatos y herramientas de diseño para lograr los mejores diseños de las bases de datos complejas. Sin pretender ser exhaustivos, la Tabla 12.1 muestra algunas de las herramientas de diseño de bases de datos y modelado de aplicaciones más populares. Las empresas de la tabla aparecen en orden alfabético.

# 12.6 Resumen

Hemos empezado el capítulo explicando el papel que juegan los sistemas de información en las empresas; los sistemas de bases de datos pueden verse como una parte de los sistemas de información en las aplicaciones a escala grande. Hemos explicado cómo encajan las bases de datos dentro de un sistema de información para la administración de los recursos de información en una empresa, así como el ciclo vital que deben pasar. Después vimos las seis fases del proceso de diseño. Las tres fases que normalmente se incluyen como una parte del diseño de bases de datos son el diseño conceptual, el diseño lógico (mapeo del modelo de datos) y el diseño físico. También explicamos la fase inicial, la recopilación y análisis de requisitos, que a menudo se considera como una *fase de prediseño*. Además, en algún momento del diseño, es preciso elegir un paquete DBMS específico. Explicamos algunos de los criterios organizativos que entran en juego a la hora de elegir un DBMS. En cuanto se detectan problemas de rendimiento o se añaden aplicaciones nuevas, hay que modificar los diseños. Hemos resaltado la importancia de diseñar tanto el esquema como las aplicaciones (o transacciones). Explicamos las diferentes metodologías para diseñar el esquema conceptual y la diferencia entre un diseño centralizado del esquema y la metodología de la integración de vistas.

También hemos realizado una introducción a los diagramas UML como ayuda para especificar los modelos y los diseños de las bases de datos. Después de introducir los diagramas estructurales y de comportamiento, describimos los detalles de notación para los siguientes tipos de diagramas: caso de uso, secuencia y estado. Los diagramas de clases ya los explicamos en las Secciones 3.8 y 4.6, respectivamente. Vimos cómo se especifican los requisitos de la base de datos Universidad mediante los diagramas, y cómo se pueden utilizar esos requisitos para desarrollar el diseño conceptual de la base de datos. Sólo hemos ofrecido unos detalles ilustrativos, y no una especificación completa. Después vimos la herramienta de desarrollo de software más popular actualmente, Rational Rose y Rose Data Modeler, que proporciona soporte para las fases de diseño conceptual y diseño lógico del diseño de las bases de datos. Rose es una herramienta mucho más amplia para el diseño de sistemas de información. Por último, explicamos brevemente la funcionalidad y las características deseables de las herramientas automáticas de diseño de bases de datos comerciales que se centran más en el diseño de la base de datos, en oposición a Rose.

## Preguntas de repaso

**12.1.** ¿Cuáles son las seis fases del diseño de una base de datos? Explíquelas.

**12.2.** ¿Cuáles de las seis fases están consideradas como las principales actividades del proceso de diseño de bases de datos? ¿Por qué?

**12.3.** ¿Por qué es importante diseñar en paralelo los esquemas y las aplicaciones?

**12.4.** ¿Por qué es importante utilizar un modelo de datos independiente de la implementación durante el diseño del esquema conceptual? ¿Qué modelos se utilizan en las herramientas de diseño actuales? ¿Por qué?

**12.5.** Explique la importancia de la recopilación y el análisis de los requisitos.

**12.6.** Considere una aplicación real de un sistema de bases de datos de su interés. Defina los requisitos de los diferentes niveles de usuarios en términos de los datos necesarios, tipos de consultas y transacciones a procesar.

**12.7.** Explique las características que un modelo de datos debe poseer para el diseño del esquema conceptual.

**12.8.** Compare y contraste las dos metodologías principales para diseñar el esquema conceptual.

**12.9.** Explique las estrategias para diseñar un esquema conceptual a partir de sus requisitos.

**12.10.** ¿Cuáles son los pasos de la integración de vistas para el diseño del esquema conceptual? ¿Cuáles son las dificultades durante cada paso?

**12.11.** ¿Cómo debería trabajar una herramienta de integración de vistas? Diseñe una arquitectura modular de ejemplo para esa herramienta.

**12.12.** ¿Cuáles son las diferentes estrategias para la integración de las vistas?

**12.13.** Explique los factores que influyen a la hora de elegir un paquete DBMS para el sistema de información de una empresa.

**12.14.** ¿Qué es el mapeo de un modelo de datos independiente del sistema? ¿En qué se diferencia del mapeo de un modelo de datos independiente del sistema?

**12.15.** ¿Cuáles son los factores importantes que influyen en el diseño físico de una base de datos?

**12.16.** Explique las decisiones que se toman durante el diseño físico de una base de datos.

**12.17.** Explique los ciclos vitales principal y secundario de un sistema de información.

**12.18.** Explique las directrices del diseño físico de una base de datos en los RDBMSs.

**12.19.** Explique los tipos de modificaciones que pueden aplicarse al diseño lógico de bases de datos de una base de datos relacional.

**12.20.** ¿Cuáles son las funciones típicas que proporcionan las herramientas de diseño de bases de datos?

**12.21.** ¿Qué tipo de funcionalidad sería deseable en las herramientas automáticas a fin de dar soporte a un diseño óptimo de bases de datos grandes?

**12.22.** ¿Cuáles son los DBMSs relacionales que dominan el mercado actualmente? Elija uno con el que esté familiarizado y preséntelo en base a los criterios detallados en la Sección 12.2.3.

**12.23.** A continuación tiene un posible DDL correspondiente a la Figura 5.1:

```
CREATE TABLE ESTUDIANTE (
    Nombre      VARCHAR(30)       NOT NULL,
    Dni         CHAR(9)           PRIMARY KEY,
    TlfCasa     VARCHAR(14),
    Direcc      VARCHAR(40),
    TlfOficina  VARCHAR(14),
    Edad  INT,
    Gpa         DECIMAL(4,3)
);
```

**a.** Explique las siguientes decisiones de diseño:

**b.** La opción de que Nombre sea NON NULL.

**c.** Selección de Dni como PRIMARY KEY.

**d.** Elección de los tamaños de campo y su precisión.

**e.** Cualquier modificación de los campos definidos en esta base de datos.

**f.** Cualquier restricción en los campos individuales.

**12.24.** ¿Qué convenciones de denominación puede desarrollar para ayudar a identificar las *foreign keys* con más eficacia?

**12.25.** ¿Qué funciones proporcionan las herramientas de diseño de bases de datos típicas?

## Bibliografía seleccionada

Hay muchísimos libros dedicados al diseño de bases de datos. Vamos a ver algunos de ellos. Batini y otros (1992) es un tratamiento integral del diseño conceptual y lógico de bases de datos. Wiederhold (1986) abarca todas las fases del diseño de bases de datos, haciendo hincapié en el diseño físico. O'Neil (1994) incluye una detallada explicación del diseño físico y de las transacciones en referencia a los RDBMSs comerciales. En los ochenta se desarrolló un gran trabajo en modelado conceptual y diseño. Brodie y otros (1984) ofrece una colección de capítulos sobre modelado conceptual, especificación y análisis de restricciones, y diseño de transacciones. Yao (1985) es una colección de trabajos que van desde las técnicas de especificación de requisitos hasta la reestructuración de esquemas. Teorey (1998) se centra en el modelado EER y explica varios aspectos del diseño conceptual y lógico de bases de datos. McFadden and Hoffer (1997) es una buena introducción a los problemas de las aplicaciones comerciales en cuanto a la administración de bases de datos.

Navathe y Kerschberg (1986) explica todas las fases del diseño de bases de datos y apunta al papel de los diccionarios de datos. Goldfine y Konig (1988) y ANSI (1989) explican el papel que desempeñan los diccionarios de datos en el diseño de bases de datos. Rozen y Shasha (1991) y Carlis y March (1984) presentan diferentes modelos para el problema del diseño físico de bases de datos. El diseño de bases de datos orientado a objetos se explica en Schlaer y Mellor (1988), Rumbaugh y otros (1991), Martin y Odell (1991), y Jacobson (1992). Libros más recientes, como Blaha and Premerlani (1998) y Rumbaugh y otros (1999), consolidan las técnicas existentes de diseño orientado a objetos. Fowler y Scott (1997) es una introducción rápida a UML. Si desea un tratamiento más global de UML y su uso en el proceso de desarrollo de software, consulte Jacobsen y otros (1999) y Rumbaugh y otros (1999).

La recopilación y análisis de requisitos es un tema ampliamente estudiado. Chatzoglu y otros (1997) y Lubars y otros (1993) presentan estudios de las prácticas de captura, modelado y análisis de requisitos actuales. Carroll (1995) ofrece un conjunto de lecturas basadas en escenarios de recopilación de requisitos en las etapas más tempranas del desarrollo de un sistema.

Wood y Silver (1989) ofrece una buena panorámica del proceso *Joint Application Design* (JAD, Diseño conjunto de aplicaciones) oficial. Potter y otros (1991) describe la notación Z y la metodología de especificación formal del software. Zave (1997) tiene clasificados los logros alcanzados en la investigación de la ingeniería de requisitos.

También se ha trabajado mucho en los problemas de la integración del esquema y la vista, algo que ahora es muy relevante debido a la necesidad de integrar distintas bases de datos existentes. Navathe y Gadgil (1982) define las metodologías de integración de vistas. Estas metodologías se comparan en Batini y otros (1986). En Navathe y otros (1986), Elmasri y otros (1986), y Larson y otros (1989) puede encontrar un trabajo detallado de la integración de *n* vistas. En Sheth y otros (1988) se describe una herramienta de integración basada en Elmasri y otros (1986). Otro sistema de integración de vistas se explica en Hayne y Ram (1990). Casanova y otros (1991) describe una herramienta para el diseño modular de bases de datos. Motro (1987) explica la integración respecto a las bases de datos existentes. La estrategia de equilibrado binario para la integración de vistas se explica en Teorey and Fry (1982). En Casanova y Vidal (1982) se ofrece una aproximación formal a la integración de vistas que utiliza las dependencias de inclusión. Ramesh y Ram (1997) describe una metodología de integración de relaciones en los esquemas, utilizando el conocimiento de las restricciones de integridad; es una ampliación del trabajo anterior realizado en Navathe y otros (1984a). Sheth y otros (1993) describe los problemas de construir esquemas globales, razonando sobre las relaciones de los atributos y las equivalencias de las entidades. Navathe y Savasere (1996) describe un método práctico de construcción de esquemas globales basándose en los operadores aplicados a los componentes de esquema. Santucci (1998) proporciona un tratamiento detallado del refinamiento de los esquemas EER de cara a la integración. Castano y otros (1999) presenta un estudio global sobre las técnicas de análisis conceptual de esquemas.

El diseño de transacciones es un tema que se ha investigado algo menos. Mylopoulos y otros (1980) propuso el lenguaje TAXIS, y Albano y otros (1987) desarrolló el sistema GALILEO: los dos son lenguajes globales enfocados a la especificación de transacciones. El lenguaje GORDAS para el modelo ECR (Elmasri y otros 1985) tiene la capacidad de especificar transacciones. Navathe y Balaraman (1991) y Ngu (1991) explican el modelado de transacciones en general para la semántica de los modelos de datos. Elmagarmid (1992) explica los modelos de transacción para aplicaciones más avanzadas. Batini y otros (1992, Capítulos 8, 9 y 11) explica el diseño de transacciones de alto nivel y el análisis conjunto de datos y funciones. Shasha (1992) es una excelente fuente para el refinamiento de bases de datos.

En los sitios web de los desarrolladores encontrará información sobre las herramientas de diseño comerciales de bases de datos (en la Tabla 12.1 tiene el nombre de estas compañías). Los principios que hay detrás de las herramientas de diseño se explican en Batini y otros (1992, Capítulo 15). La herramienta SECSI se describe en Metais y otros (1998). DKE (1997) es una publicación especial sobre los problemas del lenguaje natural en las bases de datos.

# PARTE 4

# Almacenamiento de datos, indexación, procesamiento de consultas y diseño físico

CAPÍTULO 13

# Almacenamiento en disco, estructuras básicas de ficheros y dispersión

Las bases de datos se almacenan físicamente como ficheros de registros, normalmente en discos magnéticos. Este capítulo y el siguiente hablan de la organización de las bases de datos en el almacenamiento y de las técnicas para acceder a ellas de forma eficaz mediante diversos algoritmos, algunos de los cuales requieren estructuras auxiliares de datos denominadas índices. Empezamos en la Sección 13.1 haciendo una introducción de los conceptos de las jerarquías de almacenamiento y de cómo se utilizan en los sistemas de bases de datos. La Sección 13.2 está dedicada a una descripción de los dispositivos de almacenamiento en disco magnético y sus características, y también describe brevemente los dispositivos de almacenamiento de cinta magnética. Después de explicar las diferentes tecnologías de almacenamiento, centramos nuestra atención en los métodos de organización de los datos en el disco. La Sección 13.3 explica la técnica del doble búfer, que se utiliza para acelerar la recuperación de varios bloques de disco. En la Sección 13.4 hablamos de las distintas formas de formatear y almacenar registros de fichero en disco. En la Sección 13.5 explicamos los distintos tipos de operaciones que se aplican normalmente a los registros de los ficheros. Presentamos los tres métodos principales para organizar los registros de un fichero en un disco: registros desordenados en la Sección 13.6; registros ordenados en la Sección 13.7; y registros dispersos en la Sección 13.8.

La Sección 13.9 explica brevemente los ficheros de registros mixtos y otros métodos de organización de registros, como los árboles B. Son particularmente interesantes para almacenar las bases de datos orientadas a objetos, que explicamos posteriormente en los Capítulos 20 y 21. La Sección 13.10 describe RAID (Matriz redundante de discos baratos [o independientes], *Redundant Array of Inexpensive [o Independent] Disks*), una arquitectura de sistema de almacenamiento de datos que se utiliza normalmente en empresas grandes para mejorar la fiabilidad y el rendimiento. Por último, en la Sección 13.11 describimos dos desarrollos recientes que han visto la luz en el área de los sistemas de almacenamiento: las redes de área de almacenamiento (SAN, *storage area networks*) y el almacenamiento conectado a la red (NAS, *network attached storage*). En el Capítulo 14 explicamos técnicas para crear estructuras auxiliares de datos, denominadas índices, que aceleran la búsqueda y la recuperación de registros. Estas técnicas implican el almacenamiento de datos auxiliares, denominados ficheros de índice, además de los propios registros del fichero.

El lector puede ojear los Capítulos 13 y 14, o incluso omitirlos si ya ha estudiado las organizaciones de ficheros. El material aquí explicado, en particular las Secciones 13.1 a 13.8, es necesario para comprender los Capítulos 15 y 16, que están dedicados al procesamiento y la optimización de las consultas.

# 13.1 Introducción

La colección de datos que constituye una base de datos computerizada debe almacenarse físicamente en un **medio de almacenamiento** de un computador. El software DBMS puede recuperar, actualizar y procesar después estos datos cuando lo necesite. Los medios de almacenamiento forman una *jerarquía de almacenamiento* que incluye dos categorías principales:

- **Almacenamiento principal o primario.** Esta categoría incluye los medios de almacenamiento en los que la CPU (unidad central de procesamiento) puede operar, como la memoria principal del computador y las memorias caché, más pequeñas pero más rápidas. El almacenamiento principal normalmente proporciona un acceso rápido a los datos, pero tiene una capacidad de almacenamiento limitada.
- **Almacenamientos secundario y terciario.** Esta categoría incluye los discos magnéticos, los discos ópticos y las cintas. Las unidades de disco duro se clasifican como almacenamiento secundario, mientras que los medios removibles o extraíbles están considerados como almacenamiento terciario. Estos dispositivos normalmente tienen gran capacidad, cuestan poco y proporcionan un acceso más lento a los datos que los dispositivos de almacenamiento principales. La CPU no puede procesar directamente los datos almacenados en un almacenamiento secundario o terciario; primero deben copiarse en el almacenamiento principal.

En la Sección 13.1.1 ofrecemos una visión general de los distintos dispositivos de almacenamiento que se utilizan como almacenamiento principal y secundario, y en la Sección 13.1.2 veremos cómo se manipulan las bases de datos en la jerarquía de almacenamiento.

## 13.1.1 Jerarquías de memoria y dispositivos de almacenamiento

En un computador moderno, los datos residen y se transportan por una jerarquía de medios de almacenamiento. La memoria de alta velocidad es la más cara y, por tanto, está disponible con una menor capacidad. La memoria de velocidad más baja es el almacenamiento en cinta *offline*, que está esencialmente disponible con capacidades de almacenamiento indefinido.

En el extremo más caro del *nivel de almacenamiento principal* tenemos la **memoria caché**, que es una RAM (memoria de acceso aleatorio) estática. La memoria caché la utiliza normalmente la CPU para acelerar la ejecución de los programas. El siguiente nivel del almacenamiento principal es la DRAM (RAM dinámica), que proporciona el área de trabajo principal para que la CPU almacene programas y datos. Popularmente se denomina **memoria principal**. La ventaja de la DRAM es su bajo coste, que sigue bajando; el inconveniente es su volatilidad[1] y baja velocidad en comparación con la RAM estática. En el *nivel de almacenamiento secundario y terciario*, la jerarquía incluye los discos magnéticos, así como el **almacenamiento en masa** en forma de dispositivos CD-ROM (Disco compacto–memoria de sólo lectura) y DVD y, finalmente, las cintas en el extremo más caro de la jerarquía. La **capacidad de almacenamiento** se mide en kilobytes (Kbyte o 1.000 bytes), megabytes (MB o 1 millón de bytes), gigabytes (GB o mil millones de bytes), e incluso terabytes (1.000 GB).

Los programas residen y se ejecutan en la DRAM. Por lo general, las bases de datos permanentes más grandes residen en el almacenamiento secundario, y las porciones de la base de datos se leen y se escriben en los búferes de la memoria principal según se necesitan. Ahora que los computadores personales y las estaciones de trabajo tienen cientos de megabytes de datos en la DRAM, es posible cargar gran parte de la base de datos en la memoria principal. Es común que un servidor tenga de 8 a 16 GB de memoria RAM. En algunos casos, la memoria principal puede albergar la base de datos entera (con una copia de seguridad en un disco magnético), lo que lleva a las **bases de datos en memoria principal**; son particularmente útiles en las aplicaciones

[1] La memoria volátil normalmente pierde su contenido en caso de una interrupción del suministro eléctrico, algo que no ocurre con la memoria no volátil.

en tiempo real que requieren unos tiempos de respuesta extremadamente rápidos. Un ejemplo son las aplicaciones de conmutación telefónica, que almacenan bases de datos que contienen información de enrutamiento y de línea en la memoria principal.

Entre la DRAM y el almacenamiento en disco magnético encontramos otra forma de memoria, la **memoria flash**, muy popular actualmente debido principalmente a que no es volátil. Las memorias flash son de alta densidad y alto rendimiento que utilizan la tecnología EEPROM (Memoria de sólo lectura programable y borrable eléctricamente, *Electrically Erasable Programmable Read-Only Memory*). La ventaja de la memoria flash es su alta velocidad de acceso; el inconveniente es que debe borrarse y escribirse un bloque entero simultáneamente.[2] Las tarjetas de memoria flash se presentan como medio de almacenamiento de datos en dispositivos cuya capacidad va desde unos pocos megabytes hasta unos cuantos gigabytes. Dichas tarjetas se utilizan en cámaras, reproductores de MP3, accesorios de almacenamiento USB, etcétera.

Los discos CD-ROM almacenan los datos ópticamente y los leen mediante un láser. Los CD-ROMs contienen datos pregrabados que no se pueden sobrescribir. Los discos WORM (Escribir una vez y leer muchas veces, *Write-Once-Read-Many*) son una forma de almacenamiento óptico que se utiliza para archivar datos; permiten que los datos se escriban una vez y se lean un número indefinido de veces sin posibilidad de borrarlos. Permiten almacenar aproximadamente medio giga de datos por disco.[3] Las **memorias jukebox ópticas** utilizan un array de bandejas CD-ROM, que se van cargando en las unidades bajo demanda. Aunque los jukeboxes ópticos tienen capacidades que rondan cientos de gigabytes, sus tiempos de recuperación se mueven en cientos de milisegundos, un poco más lentos que los discos magnéticos. Este tipo de almacenamiento está en declive debido a la rápida bajada de precios y el incremento de la capacidad de los discos magnéticos. El DVD (disco de vídeo digital) es un estándar reciente en discos ópticos, que permite una capacidad de almacenamiento de 4,5 a 15 GB por disco. La mayoría de las unidades de disco de los computadores personales leen discos CD-ROM y DVD.

Por último, las **cintas magnéticas** se utilizan para archivar y para el almacenamiento de las copias de seguridad de los datos. Los **jukeboxes de cinta** (que contienen un banco de cintas que se catalogan y pueden cargarse automáticamente en las unidades de cinta) se están empezando a popularizar como **almacenamiento terciario** para almacenar terabytes de datos. Por ejemplo, el sistema EOS (Satélite de observación de la Tierra, *Earth Observation Satellite*) de la NASA almacena de este modo las bases de datos archivadas.

Ya es normal que las bases de datos de muchas empresas grandes ronden el terabyte de tamaño. El término **base de datos muy grande** ya no puede definirse con precisión, porque las capacidades de almacenamiento de los discos siguen creciendo y los costes reduciéndose. Muy pronto, este término puede quedar reservado para las bases de datos que contienen decenas de terabytes.

## 13.1.2 Almacenamiento de bases de datos

Las bases de datos normalmente almacenan grandes cantidades de datos que pueden persistir durante largos periodos de tiempo. Durante ese periodo, se acceden y procesan los datos repetidamente. Esto contrasta con la noción de estructuras de datos *transitorias* que persisten únicamente durante el tiempo limitado que corresponde a la ejecución del programa. La mayoría de las bases de datos se almacenan de forma permanente (o *persistentemente*) en almacenamiento secundario de disco magnético, por las siguientes razones:

- Generalmente, las bases de datos son demasiado grandes para entrar completamente en la memoria principal.

---

[2] Por ejemplo, la INTEL DD28F032SA es una memoria flash con capacidad para 32 megabits con una velocidad de acceso de 70 nanosegundos, y una tasa de transferencia de escritura de 430 KB/segundo.

[3] Sus velocidades rotacionales son más bajas (aproximadamente 400 rpm), lo que arroja unos retardos por latencia más altos y unas velocidades de transferencia bajas (aproximadamente 100 a 200 KB/segundo).

- Las circunstancias que provocan la pérdida permanente de datos almacenados son menos frecuentes con el almacenamiento secundario que con el principal. Por tanto, nos referimos al disco (y a otros dispositivos de almacenamiento secundario) como **almacenamiento no volátil**, mientras que la memoria principal se denomina, a menudo, **almacenamiento volátil**.
- El coste de almacenamiento por unidad de datos es un orden de magnitud inferior para el almacenamiento secundario en disco que para el almacenamiento principal.

Es probable que algunas de las tecnologías más nuevas (como los discos ópticos, los DVDs y los jukeboxes de cinta) ofrezcan alternativas viables al uso de discos magnéticos. En el futuro, las bases de datos pueden residir, por tanto, en diferentes niveles de la jerarquía de memoria de los descritos en la Sección 13.1.1. No obstante, se prevé que los discos magnéticos sigan siendo el medio de almacenamiento principal para las bases de datos grandes en los próximos años. Por consiguiente, es importante estudiar y entender las propiedades y características de los discos magnéticos y la forma en que los ficheros de datos pueden organizarse en el disco, a fin de diseñar bases de datos eficaces con un rendimiento aceptable.

Las cintas magnéticas se utilizan con frecuencia como medio de almacenamiento para hacer copias de seguridad de las bases de datos, porque el almacenamiento en cinta es incluso menos costoso que el almacenamiento en disco. No obstante, el acceso a los datos almacenados en una cinta es bastante lento. Los datos almacenados en las cintas están **offline**; es decir, para que los datos estén disponibles, antes se necesita alguna intervención por parte de un operador (o un dispositivo de carga automática) para cargar una cinta. Por el contrario, los discos son dispositivos **online** a los que se puede acceder directamente en cualquier momento.

Las técnicas utilizadas para almacenar grandes cantidades de datos estructurados en disco son importantes para los diseñadores de bases de datos, el DBA y los encargados de implantar un DBMS. Los diseñadores de bases de datos y el DBA deben conocer las ventajas y los inconvenientes de cada técnica de almacenamiento cuando se dispongan a diseñar, implementar y operar una base de datos en un DBMS concreto. Normalmente, el DBMS tiene varias opciones para organizar los datos. El proceso de **diseño físico de la base de datos** implica elegir las técnicas de organización de datos particulares que mejor se adapten a los requisitos de la aplicación. Los encargados de implementar el sistema DBMS deben estudiar las técnicas de organización de datos para poder implementarlas eficazmente y, de este modo, ofrecer al DBA y los usuarios del DBMS las opciones suficientes.

Las aplicaciones típicas de bases de datos sólo necesitan procesar una pequeña porción de la base de datos en cada momento. Siempre que se necesita una determinada porción de datos, debe buscarse en el disco, copiarse en la memoria principal para su procesamiento y, después, reescribirse en el disco si los datos han cambiado. Los datos almacenados en el disco se organizan como **ficheros** de **registros**. Un registro es una colección de valores de datos que pueden interpretarse como hechos relacionados con las entidades, sus atributos y sus relaciones. Los registros deben almacenarse en disco de un modo que haga posible su localización de la forma más eficaz cuando sean necesarios.

Hay varias **organizaciones principales de ficheros**, que determinan la *colocación física* de los registros del fichero en el disco y, por tanto, *cómo se puede acceder a ellos*. Un *fichero heap* (o fichero desordenado) coloca los registros en el disco sin un orden particular, añadiendo los registros nuevos al final del fichero, mientras que un *fichero ordenado* (o *fichero secuencial*) mantiene ordenados los registros por el valor de un campo particular (denominado clave de ordenación). Un *fichero disperso* utiliza una función de dispersión (*hash*) aplicada a un campo concreto (denominado clave de dispersión) para determinar la ubicación de un registro en el disco. Otras organizaciones de ficheros, como los árboles B, utilizan estructuras en árbol. Explicaremos las principales organizaciones de ficheros en las Secciones 13.6 a 13.9. Una **organización secundaria** o **estructura de acceso auxiliar** permite un acceso eficaz a los registros de un fichero basándose en campos alternos a los utilizados para la organización principal del fichero. La mayoría de ellos son índices y los explicaremos en el Capítulo 14.

# 13.2 Dispositivos de almacenamiento secundario

En esta sección describimos algunas de las características de los dispositivos de almacenamiento de disco magnético y de cinta magnética. Los lectores que ya hayan estudiado estos dispositivos, pueden echar un vistazo simplemente a la sección.

## 13.2.1 Descripción del hardware de los dispositivos de disco

Los discos magnéticos se utilizan para almacenar grandes cantidades de datos. La unidad de datos más básica es el **bit** de información. Al magnetizar un área del disco de determinadas formas, podemos hacer que represente un valor de bit de 0 (cero) o 1 (uno). Para codificar la información, los bits se agrupan en **bytes** (o **caracteres**). El tamaño de los bytes es normalmente de 4 a 8 bits, en función del computador y del dispositivo. Asumimos que un carácter se almacena en un byte, y utilizamos indistintamente los términos *byte* y *carácter*. La **capacidad** de un disco es el número de bytes que puede almacenar, que normalmente es muy grande. Los disquetes que se utilizaban con los microcomputadores almacenaban de 400 KB a 1,5 MB; los discos duros de los microcomputadores almacenan desde varios cientos de MB hasta decenas de GB; y los grandes paquetes de discos que se utilizan en los servidores y los *mainframes* tienen capacidades de cientos de GB. La capacidad de los discos sigue creciendo a medida que mejora la tecnología.

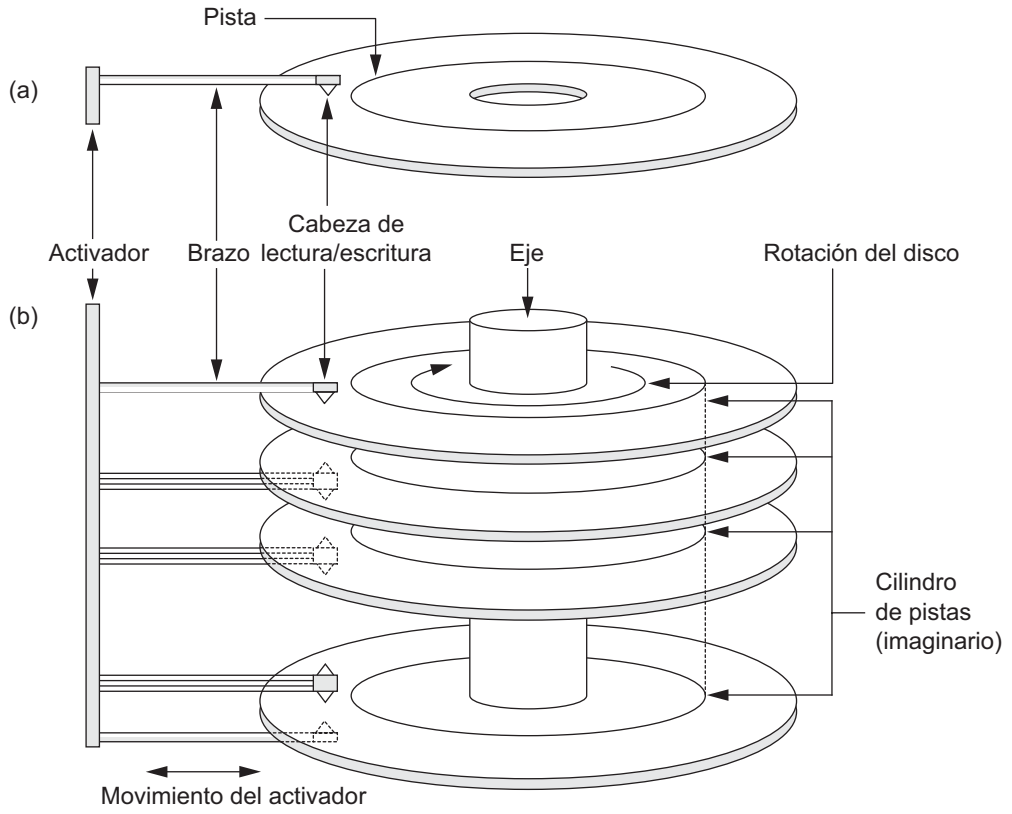
Cualquiera que sea su capacidad, todos los discos están fabricados con un material magnético al que se le da forma de disco circular, como muestra la Figura 13.1(a), que va protegido por una carcasa plástica o acrílica. Un disco es de **una cara** si sólo almacena información en una de sus superficies, y de **doble cara** si utiliza las dos superficies. Para aumentar la capacidad de almacenamiento, los discos se ensamblan como **paquetes de discos** (véase la Figura 13.1[b]), que pueden incluir muchos discos y, por tanto, muchas superficies. La información se almacena en la superficie del disco en círculos concéntricos de *poca anchura,*[4] y cada uno de un diámetro distinto. Cada uno de esos círculos es una **pista**. En los paquetes de discos, las pistas de las distintas superficies que tienen el mismo diámetro reciben el nombre de **cilindro**, debido a la forma que tendrían si se conectaran en el espacio. El concepto de cilindro es importante porque los datos almacenados en un mismo cilindro se pueden recuperar mucho más rápidamente que si estuvieran distribuidos por diferentes cilindros.

El número de pistas de un disco varía de unos cuantos centenares a unos cuantos miles, y la capacidad de cada pista normalmente varía de unas decenas de Kbytes a 150 Kbytes. Como una pista normalmente contiene una gran cantidad de información, está dividida en bloques más pequeños o **sectores**. La división de una pista en sectores está codificada en la superficie del disco y no se puede cambiar. Un tipo de organización de sectores (véase la Figura 13.2[a]), toma una porción de una pista que subtiende un ángulo fijo al centro de un sector. Son posibles otras organizaciones de los sectores, una de las cuales tiene los sectores subtendidos al centro en ángulos más pequeños a medida que nos alejamos, lo que mantiene una densidad uniforme de grabación (véase la Figura 13.2[b]). Una técnica denominada ZBR (*Zone Bit Recording*) permite que un rango de cilindros tenga la misma cantidad de sectores por arco. Por ejemplo, los cilindros 0–99 pueden tener un sector por pista, los cilindros 100–199 pueden tener dos por pista, etcétera. No todos los discos tienen sus pistas divididas en sectores.

La división de una pista en **bloques de disco** (o **páginas**) es establecida por el sistema operativo durante el **formateo** (o **inicialización**) del disco. El tamaño del bloque se fija durante la inicialización y no puede cambiarse dinámicamente. El tamaño típico de un bloque de disco oscila entre 512 y 8.192 bytes. A menudo, los discos que tienen los sectores codificados subdividen los sectores en bloques durante la inicialización. Los bloques están separados por **huecos entre bloques** de un tamaño fijo, que incluyen información de control codificada de forma especial y que se escribe durante la inicialización del disco. Esta información se utiliza

[4] En algunos discos, los círculos están ahora conectados como en una especie de espiral continua.

**Figura 13.1.** (a) Un disco de una cara con hardware de lectura/escritura. (b) Un paquete de discos con hardware de lectura/escritura.




para determinar el bloque de la pista que va a continuación de cada hueco. La Tabla 13.1 representa las especificaciones de un disco típico.

Hay una mejora continua en la capacidad de almacenamiento y las velocidades de transferencia asociadas con los discos; además, su precio baja progresivamente (actualmente, un megabyte de almacenamiento de disco cuesta una fracción de dólar).

Un disco es un dispositivo direccionable de *acceso aleatorio*. La transferencia de datos entre la memoria principal y el disco tiene lugar en unidades de bloques de disco. La **dirección de hardware** de un bloque (una combinación de número de cilindro, número de pista [número de superficie dentro del cilindro en el que se encuentra la pista] y número de bloque [dentro de la pista]) se suministra al hardware de E/S del disco. En muchas unidades de disco modernas, el controlador de la unidad de disco mapea automáticamente un número denominado LBA (Dirección de bloque lógica, *Logical Block Address*), que es un número entre 0 y $n$ (asumiendo que la capacidad total del disco es $n + 1$ bloques), al bloque correcto. También se proporciona la dirección de un **búfer** (un área reservada contigua en el almacenamiento principal que almacena un bloque). Para un comando de **lectura**, el bloque del disco se copia en el búfer, mientras que para un comando de **escritura**, el contenido del búfer se copia en el bloque de disco. En ocasiones, varios bloques contiguos, denominados **clúster**, se pueden transferir como una unidad. En este caso, el tamaño del búfer se ajusta para que coincida con el número de bytes del clúster.

El mecanismo hardware actual que lee o escribe un bloque es la **cabeza de lectura/escritura** del disco, que es parte de un sistema denominado **unidad de disco**. En la unidad de disco se monta un disco o un paquete de discos, que giran gracias a un motor incluido en la unidad. Una cabeza de lectura/escritura incluye un

**Figura 13.2.** Diferentes organizaciones de sectores en un disco. (a) Sectores subtendidos en un ángulo fijo. (b) Sectores que mantienen una densidad de grabación uniforme.

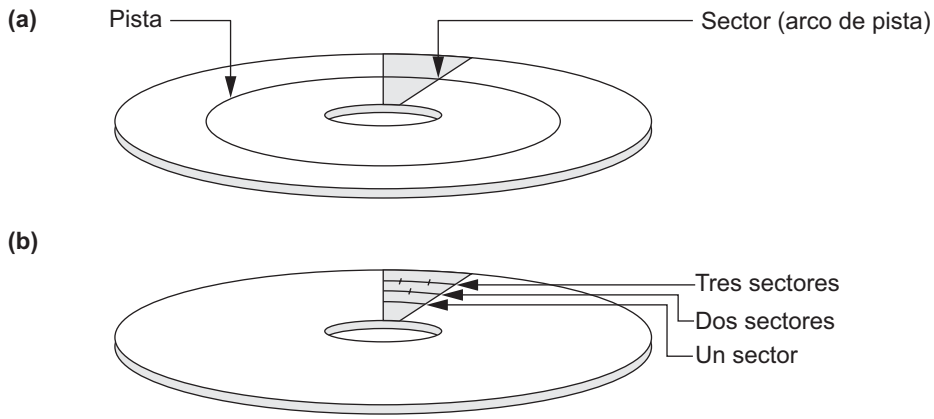



**Tabla 13.1.** Especificaciones de los discos Cheetah de alta gama de Seagate.

| **Descripción** | **Cheetah 10K.6** | **Cheetah 10K.7** |
|---|---|---|
| Número de modelo | ST3146807LC | ST3300007LW |
| Factor de forma (anchura) | 3,5 pulgadas | 3,5 pulgadas |
| Factor de forma (altura) | 1 pulgada | 1 pulgada |
| Altura | 25,4 mm. | 25,4 mm. |
| Anchura | 101,6 mm. | 101,6 mm. |
| Longitud | 146,05 mm. | 146,05 mm. |
| Peso | 0,73 Kg. | 0,726 kg. |
| | | |
| **Capacidad/Interfaz** | | |
| Capacidad formateado | 146,8 Gbytes | 300 Gbytes |
| Tipo de interfaz | 80-pin | 68-pin |
| | | |
| **Configuración** | | |
| Número de discos (físicos) | 4 | 4 |
| Número de cabezas (físicas) | 8 | 8 |
| Número de cilindros | 49.854 | 90.774 |
| Total de pistas | | 726.192 |
| Bytes por sector | 512 | 512 |
| Densidad superficial | 36.000 Mb/pulg.cuadrada | |
| Densidad de pista | 64.000 pistas/pulgada | 105.000 pistas/pulgada |
| Densidad de grabación | 570.000 bits/pulgada | 658.000 bits/pulgada |
| Bytes/pista (promedio) | | 556 |
| | | |
| **Rendimiento** | | |
| Velocidades de transferencia | | |
| Velocidad de transferencia interna (mín.) | 475 Mb/seg. | 472 Mb/seg. |
| Velocidad de transferencia interna (máx.) | 840 Mb/seg. | 944 Mb/seg. |
| Velocidad de transferencia int. formateado (mín.) | 43 MB/seg. | 59 MB/seg. |
| Velocidad de transferencia int. formateado (máx.) | 78 MB/seg. | 118 MB/seg. |
| Velocidad de transferencia de E/S externa (máx.) | 320 MB/seg. | 320 MB/seg. |
| Promedio de velocidad de transferencia formateado | 59,9 MB/seg. | 59,5 MB/seg. |

**Tabla 13.1.** Especificaciones de los discos Cheetah de alta gama de Seagate. (*Continuación*)

| **Tiempos de búsqueda** | **Cheetah 10K.6** | **Cheetah 10K.7** |
|---|---|---|
| Tiempo medio de búsqueda (lectura) | 4,7 ms. (típica) | 4,7 ms. (típica) |
| Tiempo medio de búsqueda (escritura) | 5,2 ms. (típica) | 5,3 ms. (típica) |
| Pista a pista, búsqueda, lectura | 0,3 ms. (típica) | 0,2 ms. (típica) |
| Pista a pista, búsqueda, escritura | 0,5 ms. (típica) | 0,5 ms. (típica) |
| Buscar en el disco entero, lectura | | 9,5 ms. (típica) |
| Buscar en el disco entero, escritura | | 10,3 ms. (típica) |
| Latencia media | 2,99 ms. | 3 mseg. |
| **Otros** | | |
| Tamaño predeterminado del búfer (caché) | 8.000 KB | 8.192 KB |
| Velocidad de giro | 10.000 RPM | 10.000 RPM |
| Tiempo desde el encendido hasta estar listo | | 25 segundos |
| **Requisitos eléctricos** | | |
| **Corriente** | | |
| Corriente normal (12VDC +/- 5%) | 0,95 amperios | 1,09 amperios |
| Corriente típica (5VDC +/- 5%) | 0,9 amperios | 0,68 amperios |
| Potencia en modo de bajo consumo (typ) | 10,6 vatios | 10,14 vatios |
| **Fiabilidad** | | |
| Promedio de tiempo entre fallos (MTBF) | 1.200.000 horas | 1.400.000 horas |
| Errores de lectura recuperables | 10 por 1012 bits | 10 por 1012 bits en lectura |
| Errores de lectura no recuperables | 10 por 1015 bits | 10 por 1015 bits en lectura |
| Errores de búsqueda | 10 por 108 bits | |
| Vida de servicio | 5 año(s) | 5 año(s) |
| Periodo de garantía limitado | 5 año(s) | 5 año(s) |

(Cortesía de Seagate Technology)

componente electrónico conectado a un **brazo mecánico**. Los paquetes de discos con varias superficies están controlados por varias cabezas de lectura/escritura, una por cada superficie (véase la Figura 13.1[b]). Todos los brazos están conectados a un **activador**, conectado a otro motor eléctrico, que mueve las cabezas de lectura/escritura al unísono y las coloca con precisión sobre el cilindro o las pistas especificadas en una dirección de bloque.

Las unidades de disco de los discos duros hacen girar el paquete de discos continuamente, a una velocidad constante (normalmente, entre 5400 y 15.000 rpm). En el caso de un disquete, la unidad de disco empieza a girar el disco siempre que se inicia una solicitud de lectura o escritura, y la rotación cesa tan pronto como se completa la transferencia de datos. Una vez que la cabeza de lectura/escritura está posicionada en la pista correcta y el bloque especificado en la dirección del bloque ubicado bajo dicha cabeza, el componente electrónico de la cabeza se activa para transferir los datos. Algunas unidades de disco tienen cabezas de lectura/escritura fijas, tantas como pistas. Estas unidades se conocen como **discos de cabeza fija**, mientras que las unidades de disco con un activador se denominan **discos de cabeza móvil**. En las primeras, una pista o cilindro se selecciona electrónicamente cambiando a la cabeza de lectura/escritura adecuada, en lugar de recurrir a un movimiento mecánico real; en consecuencia, son mucho más rápidas. No obstante, el coste de las cabezas de lectura/escritura adicionales es muy alto, por lo que estos discos no se utilizan comúnmente.

El **controlador de disco**, normalmente incrustado en la unidad de disco, controla ésta y su interacción con el sistema. Una de las interfaces estándar que actualmente se utilizan para las unidades de disco de los PCs y las estaciones de trabajo es SCSI (Interfaz de almacenamiento para equipos pequeños, *Small Computer Storage Interface*). El controlador acepta comandos de E/S de alto nivel y toma la acción apropiada para posicionar el

brazo y provoca la acción de lectura/escritura. Para transferir un bloque de disco, dada su dirección, el controlador de disco primero debe colocar mecánicamente la cabeza de lectura/escritura en la pista correcta. El tiempo requerido para ello es el **tiempo de búsqueda**. Los tiempos de búsqueda típicos son de 7 a 10 msegs en los sobremesa y de 3 a 8 msegs en los servidores. Siguiendo a esto, hay otro retardo, denominado **retardo rotacional** o **latencia**, que se produce mientras el principio del bloque deseado gira hasta su posición bajo la cabeza de lectura/escritura. Este retardo depende de las rpm del disco. Por ejemplo, a 15.000 rpm, el tiempo por rotación es de 4 mseg y el retardo rotacional medio es el tiempo por media revolución, o 2 mseg. Por último, aún se necesita algo más de tiempo para transferir los datos; es lo que se conoce como **tiempo de transferencia del bloque**. Por tanto, el tiempo total necesario para localizar y transferir un bloque arbitrario, dada su dirección, es la suma del tiempo de búsqueda, el retardo rotacional y el tiempo de transferencia del bloque. El tiempo de búsqueda y el retardo rotacional son normalmente mucho más grandes que el tiempo de transferencia del bloque. Para que la transferencia de varios bloques sea más eficaz, es común transferir varios bloques consecutivos de la misma pista o cilindro. Esto elimina el tiempo de búsqueda y el retardo rotacional de todos los bloques menos del primero, lo que puede ahorrar un tiempo sustancial al transferir numerosos bloques contiguos. Normalmente, el fabricante del disco proporciona una **velocidad de transferencia en masa** para calcular el tiempo requerido para transferir bloques consecutivos. El Apéndice B contiene una explicación de éstos y de otros parámetros.

El tiempo necesario para localizar y transferir un bloque de disco está en el orden de los milisegundos, normalmente de 9 a 60 msegs. En el caso de bloques contiguos, la localización del primer bloque lleva aproximadamente de 9 a 60 msegs, pero la transferencia de los bloques subsiguientes puede tomar sólo de 0,4 a 2 msegs cada uno. Muchas técnicas de búsqueda se benefician de la recuperación consecutiva de bloques cuando se buscan datos en el disco. En cualquier caso, un tiempo de transferencia en el orden de milisegundos se considera bastante alto en comparación con el tiempo requerido para procesar los datos en la memoria principal por parte de las CPUs actuales. Por tanto, la localización de los datos en el disco es un *importante cuello de botella* en las aplicaciones de bases de datos. Las estructuras de ficheros las explicamos aquí y en el Capítulo 14 intentaremos *minimizar el número de transferencias de bloques* necesario para localizar y transferir los datos necesarios desde el disco a la memoria principal.

### 13.2.2 Dispositivos de almacenamiento en cinta magnética

Los discos son dispositivos de almacenamiento secundario de **acceso aleatorio** porque es posible acceder aleatoriamente a un bloque arbitrario del disco una vez especificada su dirección. Las cintas magnéticas son dispositivos de acceso secuencial; para acceder al bloque enésimo de una cinta, primero tenemos que escanear los $n - 1$ bloques precedentes. Los datos se almacenan en bobinas de cinta magnética de alta capacidad, parecidas a las cintas de audio o de vídeo. Para leer datos de, o escribir datos en, una **bobina de cinta** se necesita una unidad de cinta. Normalmente, cada grupo de bits que forma un byte se almacena a través de la cinta, y los bytes se almacenan consecutivamente en la cinta.

Se utiliza una cabeza de lectura/escritura para leer o escribir datos en la cinta. Los registros de datos también se almacenan en la cinta en bloques (aunque los bloques pueden ser sustancialmente más grandes que los de los discos, y los huecos entre los bloques también son bastante grandes). Con las densidades de cinta típicas de 1.600 a 6.250 bytes por pulgada, un hueco entre bloques típico[5] de 0,6 pulgadas corresponde a un espacio de almacenamiento desaprovechado de 960 a 3750 bytes. Es costumbre agrupar muchos registros juntos en un bloque para una mejor utilización del espacio.

La principal característica de una cinta es su requisito de acceder a los bloques de datos en **orden secuencial**. Para obtener un bloque de en medio de una bobina de cinta, se monta la cinta y se escanea hasta que el bloque requerido se coloca bajo la cabeza de lectura/escritura. Por esta razón, el acceso a la cinta puede ser lento

[5] Denominados *huecos entre registros* en la terminología de las cintas.

y las cintas no se utilizan para almacenar datos online, excepto para algunas aplicaciones especializadas. Sin embargo, las cintas sirven para una función muy importante, la **copia de seguridad** de la base de datos. Una razón para hacer una copia de seguridad es mantener copias de los ficheros en disco por si se pierden datos como consecuencia de un fallo del disco, que puede ocurrir si la cabeza de lectura/escritura toca la superficie del disco debido a un mal funcionamiento mecánico. Por esta razón, los ficheros en disco se copian periódicamente en una cinta. En muchas aplicaciones online críticas, como los sistemas de reserva en aerolíneas, y al objeto de evitar tiempos de inactividad, se utilizan sistemas de réplica para mantener tres conjuntos de discos idénticos (dos en funcionamiento online y uno como copia de seguridad). Aquí, los discos offline se convierten en dispositivos de copia de seguridad. Los tres giran para poder alternarse, por si uno de ellos falla. Las cintas también se pueden utilizar para almacenar los ficheros de bases de datos excesivamente grandes. Los ficheros de bases de datos que rara vez se utilizan o están desactualizados, pero requieren que se guarde un registro histórico, pueden **archivarse** en cinta. Recientemente, las cintas magnéticas de 8 mm más pequeñas (parecidas a las que se utilizan en las videocámaras) que permiten almacenar hasta 50 GB, así como los cartuchos de exploración helicoidal de 4 mm y los CDs y DVDs grabables, se han popularizado como medios para hacer una copia de seguridad de los ficheros de datos almacenados en los PCs y las estaciones de trabajo. También se utilizan para almacenar imágenes y librerías del sistema. Una tarea primordial es hacer una copia de seguridad de las bases de datos de la empresa para que no se pierda información. Actualmente, las librerías de cintas con slots para varios cientos de cartuchos se utilizan con las Cintas lineales digitales y superdigitales (DLTs y SDLTs) que tienen capacidades de cientos de gigabytes y que graban los datos en pistas lineales. Los brazos robotizados se utilizan para escribir en varios cartuchos en paralelo utilizando varias unidades de cinta y software de etiquetado automático para identificar los cartuchos de copia de seguridad. Un ejemplo de librería gigante es el modelo L5500 de Storage Technology, que puede almacenar hasta 13,2 petabytes (petabyte = 1000 TB), con un rendimiento de 55TB/hora. Dejamos la explicación de la tecnología RAID de almacenamiento en disco y de las redes de área de almacenamiento y del almacenamiento conectado a la red para el final del capítulo.

## 13.3 Almacenamiento de bloques en el búfer

Cuando es necesario transferir varios bloques desde el disco a la memoria principal y se conocen todas las direcciones de bloque, se pueden reservar varios búferes en la memoria principal para acelerar la transferencia. Mientras se está leyendo o escribiendo en un búfer, la CPU puede procesar datos en otro búfer porque hay un procesador (controlador) de E/S de disco independiente que, una vez iniciado, puede proseguir con la transferencia de un bloque de datos entre la memoria y el disco independientemente de, y en paralelo, al procesamiento de la CPU.

La Figura 13.3 ilustra cómo se pueden llevar a cabo dos procesos en paralelo. Los procesos A y B están ejecutándose **al mismo tiempo** de forma **interpolada**, mientras que los procesos C y D se están ejecutando **al mismo tiempo** pero **en paralelo**. Cuando una CPU controla varios procesos, la ejecución en paralelo no es posible. Sin embargo, los procesos todavía se pueden ejecutar al mismo tiempo de forma interpolada. El almacenamiento en búfer es más útil cuando los procesos se pueden ejecutar al mismo tiempo y en paralelo, porque un procesador de E/S de disco independiente esté disponible, o porque la CPU tenga varios procesadores.

La Figura 13.4 ilustra cómo se pueden efectuar en paralelo la lectura y el procesamiento, cuando el tiempo necesario para procesar un bloque de disco en la memoria es inferior al tiempo necesario para leer el siguiente bloque y llenar un búfer. La CPU puede iniciar el procesamiento de un bloque una vez completada su transferencia a la memoria principal; al mismo tiempo, el procesador de E/S de disco puede estar leyendo y transfiriendo el siguiente bloque a un búfer diferente. Esta técnica se denomina **doble búfer** y también puede utilizarse para leer un flujo continuo de bloques del disco a la memoria. El doble búfer permite la lectura o escritura continuas de datos en bloques de disco consecutivos, lo que elimina el tiempo de búsqueda y el retardo rotacional de la transferencia de todos los bloques, excepto del primero. Además, los datos quedan listos para su procesamiento, lo que reduce el tiempo de espera en los programas.

**Figura 13.3.** Concurrencia interpolada frente a ejecución en paralelo.

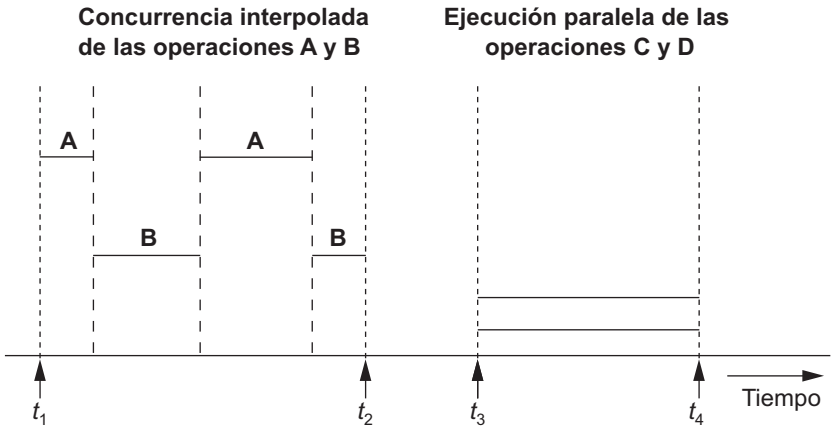
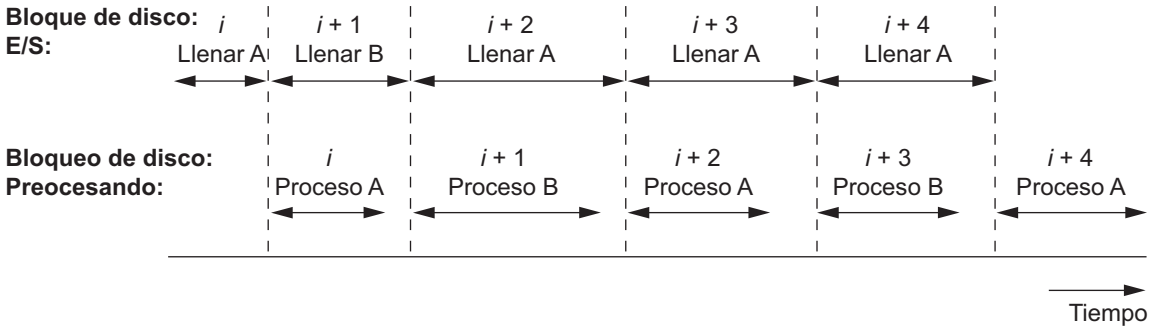



**Figura 13.4.** Uso de dos búferes, A y B, para leer desde el disco.




# 13.4 Ubicación de los registros de fichero en disco

En esta sección definimos los conceptos de registro, tipos de registros y fichero. Después, explicamos las técnicas para colocar los registros de un fichero en el disco.

## 13.4.1 Registros y tipos de registros

Los datos normalmente se almacenan en forma de **registros**. Un registro consta de una colección de **valores** o **elementos** de datos relacionados, donde cada valor está formado por uno o más bytes y corresponde a un **campo** concreto del registro. Los registros normalmente describen entidades y sus atributos. Por ejemplo, un registro EMPLEADO representa una entidad empleado, y el valor de cada campo del registro especifica algún atributo de ese empleado, como su Nombre, FechaNac, Sueldo o Supervisor. Una colección de nombres de campos y sus correspondientes tipos de datos constituyen un **tipo de registro** o una definición de **formato de registro**. Un **tipo de datos**, asociado a cada campo, especifica los tipos de valores que un campo puede tomar.

El tipo de datos de un campo normalmente es uno de los tipos de datos estándar que se utilizan en programación. Nos referimos a los tipos numéricos (entero, entero largo o de coma flotante), cadenas de caracteres (de longitud fija o variable), booleanos (que sólo pueden tener los valores 0 y 1, o TRUE y FALSE) y, en ocasio-

nes, tipos de datos de **fecha** y **hora** especialmente codificados. Un entero puede requerir 4 bytes, un entero largo 8 bytes, un número real 4 bytes, un booleano 1 byte, una fecha 10 bytes (asumiendo el formato AAAA-MM-DD), y una cadena de longitud fija de *k* caracteres *k* bytes. Las cadenas de longitud variable pueden requerir tantos bytes como caracteres tenga el valor del campo. Por ejemplo, un tipo de registro EMPLEADO puede definirse (utilizando la notación del lenguaje de programación C) como la siguiente estructura:

```
struct empleado{
     char nombre[30];
     char dni[9];
     int sueldo;
     int codtrabajo;
     char departamento[20];
} ;
```

En las aplicaciones de bases de datos recientes, puede darse la necesidad de almacenar elementos de datos consistentes en grandes objetos desestructurados, que representan imágenes, flujos de vídeo o audio digitalizados, o texto libre. Son los denominados **BLOBS** (Objetos binarios grandes, *Binary Large Objects*). Un elemento de datos BLOB normalmente se almacena independientemente de su registro en un almacén de bloques de disco, y en el registro se incluye un puntero al BLOB.

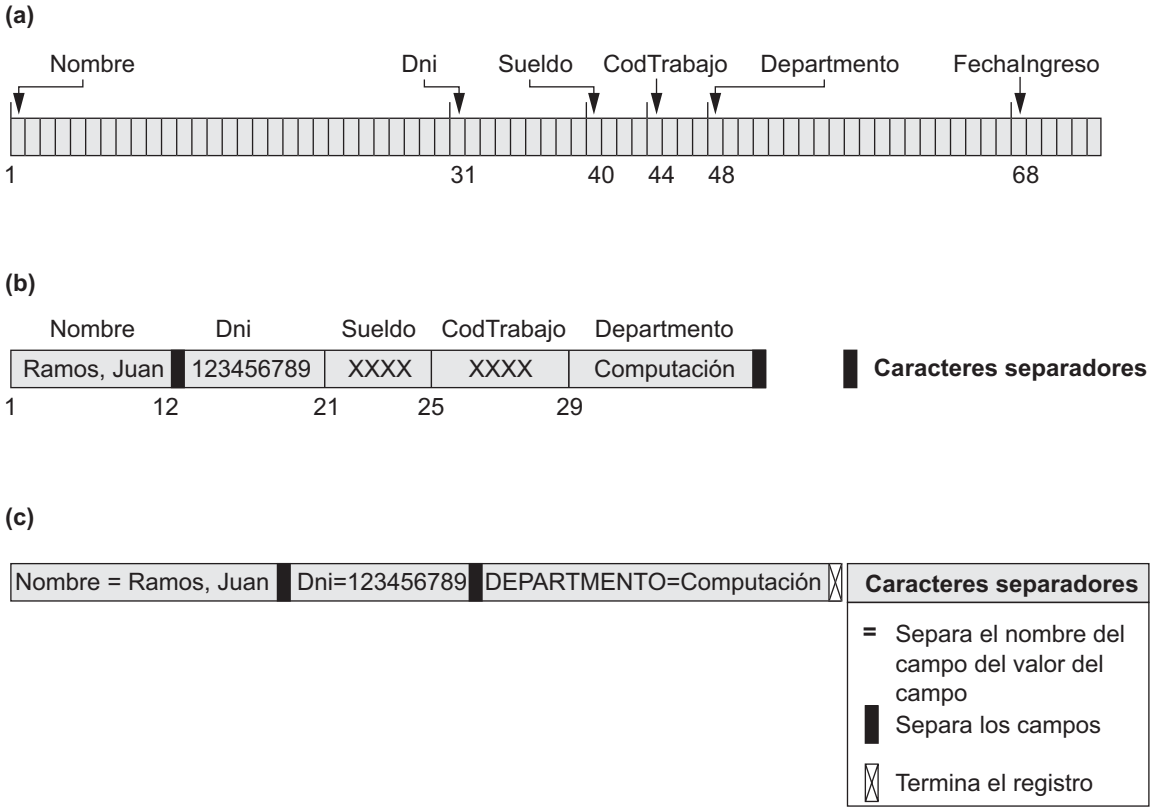
### 13.4.2 Ficheros, registros de longitud fija y registros de longitud variable

Un **fichero** es una *secuencia* de registros. En muchos casos, todos los registros de un fichero son del mismo tipo de registro. Si cada registro del fichero tiene exactamente el mismo tamaño (en bytes), se dice que el fichero está compuesto por **registros de longitud fija**. Si en el fichero hay registros que tienen tamaños diferentes, se dice que el fichero está compuesto por **registros de longitud variable**. Un fichero puede tener registros de longitud variable por varias razones:

- Los registros del fichero son del mismo tipo de registro, pero uno o más de los campos tienen un tamaño variable (**campos de longitud variable**). Por ejemplo, el campo Nombre de EMPLEADO puede ser un campo de longitud variable.
- Los registros del fichero son del mismo tipo de registro, pero uno o más de los campos pueden tener varios valores para registros individuales; un campo así se denomina **campo repetitivo** y un grupo de valores para el campo normalmente se conoce como **grupo repetitivo**.
- Los registros del fichero son del mismo tipo, pero uno o más de los campos son **opcionales**; es decir, pueden tener valores para algunos de los registros del fichero (**campos opcionales**).
- El fichero contiene registros de *diferentes tipos de registro* y, por tanto, de tamaño variable (**fichero mixto**). Esto suele ocurrir si se agruparon (se colocaron juntos) registros relacionados de tipos diferentes en bloques de disco; por ejemplo, los registros de INFORME_CALIF de un estudiante concreto pueden estar colocados a continuación del registro de ese ESTUDIANTE.

Los registros EMPLEADO de longitud fija de la Figura 13.5(a) tienen un tamaño de registro de 71 bytes. Cada registro tiene los mismos campos, y las longitudes de campo son fijas, por lo que el sistema puede identificar la posición del byte inicial de cada campo relativa a la posición inicial del registro. Esto facilita la localización de valores de campo por parte de los programas que acceden a dichos ficheros. Es posible representar un fichero que lógicamente debe tener registros de longitud variable como un fichero de registros de longitud fija. Por ejemplo, en el caso de los campos opcionales, podríamos incluir *todos los campos* en *cada registro del fichero*, pero almacenando un valor NULL especial si no existe un valor para esos campos. En un campo repetitivo, podríamos asignar tantos espacios en cada registro como el *número máximo de valores* que el campo puede tomar. En cualquier caso, el espacio se desaprovecha cuando ciertos registros no tienen especificados

**Figura 13.5.** Tres formatos de almacenamiento de un registro. (a) Un registro de longitud fija con seis campos y un tamaño de 71 bytes. (b) Un registro con dos campos de longitud variable y tres campos de longitud fija. (c) Un registro de campo variable con tres tipos de caracteres separadores.




valores para todos los espacios físicos. Ahora vamos a considerar otras opciones para formatear los registros de un fichero de registros de longitud variable.

Para los *campos de longitud variable*, cada registro tiene un valor para cada campo, pero no conocemos la longitud exacta de algunos valores de campo. Al objeto de determinar dentro de un registro en particular los bytes que representan a cada campo, podemos utilizar caracteres **separadores** especiales (por ejemplo,? o % o $), que no aparecen en ningún valor de campo, para finalizar los campos de longitud variable (véase la Figura 13.5[b]), o podemos almacenar la longitud en bytes del campo en el registro, por delante del valor del campo.

Un fichero de registros con *campos opcionales* puede formatearse de varias formas. Si el número total de campos del registro es grande, pero el número de campos que realmente aparecen en un registro típico es pequeño, podemos incluir en cada registro una secuencia de pares <nombre-del-campo, valor-del-campo> en lugar de únicamente los valores de campo. En la Figura 13.7(c) se utilizan tres tipos de caracteres separadores, aunque podemos utilizar el mismo separador para los dos primeros propósitos: separar el nombre del campo del valor del campo, y separar un campo del siguiente. Una opción más práctica es asignar un pequeño código de **tipo de campo** (por ejemplo, número entero) a cada campo e incluir en cada registro una secuencia de pares <tipo-de-campo, valor-del-campo>, en lugar de los pares <nombre-del-campo, valor-del-campo>.

Un *campo repetitivo* necesita un carácter separador para separar los valores repetitivos del campo y otro separador para indicar la terminación del campo. Por último, en el caso de un fichero que incluye *registros de diferentes tipos*, cada registro va precedido por un indicador de **tipo de registro**. Con razón, los programas que

procesan ficheros de registros de longitud variable (que normalmente son parte del sistema de ficheros y, por tanto, están ocultos a los programadores típicos) tienen que ser más complejos que los que procesan registros de longitud fija, en los que se conocen la posición de inicio y el tamaño de cada campo, que son fijos.[6]

### 13.4.3 Bloqueo de registros y registros extendidos frente a no extendidos

Los registros de un fichero deben asignarse a bloques de disco porque un bloque es la *unidad de transferencia de datos* entre el disco y la memoria. Cuando el tamaño del bloque es mayor que el tamaño del registro, un bloque puede contener varios registros, aunque algunos ficheros pueden tener registros que por su tamaño no entran en un bloque. Suponga que el tamaño de bloque es de $B$ bytes. En el caso de un fichero de registros con una longitud fija de $R$ bytes, siendo $B \geq R$, podemos encajar $bfr = \lfloor B/R \rfloor$ registros por bloque, donde $\lfloor(x)\rfloor$ (función suelo) redondea por abajo el número $x$ a un entero. El valor $bfr$ se conoce como **factor de bloqueo** del fichero. En general, $R$ no divide $B$ con exactitud, por lo que en cada bloque tenemos algo de espacio inutilizado igual a:

$$B - (bfr * R) \text{ bytes}$$

Para utilizar este espacio desaprovechado, podemos almacenar parte de un registro en un bloque y el resto en otro. Un **puntero** al final del primer bloque apunta al bloque que contiene el resto del registro, en caso de que no sean bloques consecutivos en el disco. Esta organización se denomina **extendida** porque los registros pueden abarcar más de un bloque. Siempre que un registro es más grande que un bloque, *debemos* utilizar una organización extendida. Si los registros no tienen permitido sobrepasar los límites de un bloque, se dice que la organización **no es extendida**. Ésta se utiliza con los registros de longitud fija cuando $B > R$, porque hace que cada registro empiece en una ubicación conocida del bloque, simplificándose el procesamiento del registro. En el caso de los registros de longitud variable, se puede utilizar cualquiera de las dos organizaciones, extendida o no extendida. Si el registro medio es grande, es recomendable utilizar la organización extendida para reducir la pérdida de espacio en cada bloque. La Figura 13.6 ilustra la organización extendida frente a la no extendida. Para los registros de longitud variable utilizando la organización extendida, cada bloque puede almacenar una cantidad diferente de registros. En este caso, el factor de bloqueo $bfr$ representa la cantidad media de registros por bloque de ese fichero. Podemos utilizar $bfr$ para calcular el número de bloques $b$ necesarios para un fichero de $r$ registros:

$$b = \lceil (r/bfr) \rceil \text{ bloques}$$

donde $\lceil(x)\rceil$ (función techo) redondea el valor $x$ al siguiente entero.

### 13.4.4 Asignación de bloques de fichero en un disco

Hay varias técnicas estándar para asignar los bloques de un fichero en disco. En una **asignación continua**, los bloques del fichero se asignan a bloques de disco consecutivos. De este modo, la lectura del fichero entero es mucho más rápida mediante el búfer doble, pero se hace más complejo expandir el fichero. En una **asignación enlazada**, cada bloque del fichero contiene un puntero al siguiente bloque del fichero. Esto hace más sencilla la expansión del fichero, pero más lenta la lectura del fichero entero. Una combinación de las dos asigna **clústeres** de bloques de disco consecutivos, y los clústeres se enlazan. En ocasiones, a los clústeres se les denomina **segmentos de fichero** o **extensiones**. Otra posibilidad es utilizar una **asignación indexada**, en la que uno o más **bloques de índice** contienen punteros a los bloques del fichero. También se utilizan combinaciones de estas técnicas.

---

[6] También son posibles otros esquemas para representar los registros de longitud variable.

**Figura 13.6.** Tipos de organización de registros. (a) No extendida. (b) Extendida.

### 13.4.5 Cabeceras de fichero

Una **cabecera de fichero** o **descriptor de fichero** contiene información sobre un fichero que los programas del sistema necesitan para acceder a los registros. La cabecera incluye información para determinar las direcciones de disco de los bloques del fichero y las descripciones de formato del registro (que pueden incluir longitudes de campo y orden de los campos dentro del registro en el caso de los registros no extendidos de longitud fija, y códigos de tipo, caracteres separadores y códigos de tipo de registro para los registros de longitud variable).

Para buscar un registro en el disco, se copian uno o más bloques en los búferes de la memoria principal. Los programas buscan después el registro o registros deseados en los búferes, utilizando la información de la cabecera del fichero. Si no se conoce la dirección del bloque que contiene el registro deseado, los programas de búsqueda deben realizar una **búsqueda lineal** a través de los bloques del fichero. Los bloques del fichero se van copiando en un búfer y se va buscando hasta localizar el registro o hasta que se ha buscado sin éxito en todos los bloques del fichero. Si el fichero es muy grande, esto puede llevar mucho tiempo. El objetivo de una buena organización del fichero es localizar el bloque que contiene el registro deseado con una mínima cantidad de transferencias de bloques.

## 13.5 Operaciones sobre ficheros

Las operaciones sobre ficheros se pueden agrupar en **operaciones de recuperación** y **operaciones de actualización**. Las primeras no cambian ningún dato del fichero, puesto que únicamente localizan ciertos registros para que los valores de sus campos se puedan examinar o procesar. Las segundas modifican el fichero mediante la inserción o la eliminación de registros, o modificando los valores de los campos. En cualquier caso, tenemos que **seleccionar** uno o más registros para la recuperación, la eliminación o la modificación, basándonos en una **condición de selección** (o **condición de filtrado**), que especifica los criterios que el(los) registro(s) debe(n) satisfacer.

Consideremos que el fichero EMPLEADO tiene los campos Nombre, Dni, Sueldo, CodTrabajo y Departamento. Una **condición de selección sencilla** puede implicar una comparación de igualdad sobre el valor de algún campo: por ejemplo, (Dni = '123456789') o (Departamento = 'Investigación'). Las condiciones más complejas pueden implicar otros tipos de operadores de comparación, como $>$ o $\geq$; por ejemplo, (Sueldo $\geq$ 30000). El caso general es tener una expresión booleana en los campos del fichero como condición de selección.

Cuando varios registros del fichero satisfacen una condición de búsqueda, el primer registro localizado (respecto a la secuencia física de los registros del fichero) es designado como **registro actual**. Las siguientes operaciones de búsqueda comienzan a partir de este registro y localizan el siguiente registro que satisface la condición.

Las operaciones de localización y acceso a los registros de un fichero varían de un sistema a otro. Presentamos a continuación un conjunto de operaciones representativas. Normalmente, los programas de alto nivel, como los programas DBMS, acceden a los registros utilizando estos comandos, por lo que a veces nos referimos a **variables de programa** en las siguientes descripciones:

- **Open (abrir).** Prepara el fichero para la lectura o escritura. Asigna los búferes apropiados (normalmente, al menos dos) para albergar los bloques del fichero, y recupera su cabecera. Establece el puntero del fichero al principio del mismo.
- **Reset (reiniciar).** Hace que el puntero de un fichero abierto apunte al principio del fichero.
- **Find (o Locate) (buscar).** Busca el primer registro que satisface una condición de búsqueda. Transfiere el bloque que contiene ese registro a un búfer de la memoria principal (si todavía no está en el búfer). El puntero del fichero apunta al registro del búfer, que se convierte en el *registro actual*. En ocasiones, se utilizan diferentes verbos para indicar si el registro localizado será recuperado o actualizado.
- **Read (o Get) (leer u obtener).** Copia el registro actual desde el búfer a una variable de programa del programa de usuario. Este comando también puede hacer avanzar el puntero del registro actual al siguiente registro del fichero, lo que puede hacer necesario leer del disco el siguiente bloque del fichero.
- **FindNext (buscar siguiente).** Busca el siguiente registro que satisface la condición de búsqueda. Transfiere el bloque en el que se encuentra el registro a un búfer de la memoria principal (si todavía no se encuentra aquí). El registro se almacena en el búfer y se convierte en el registro actual.
- **Delete (borrar).** Borra el registro actual y (finalmente) actualiza el fichero en disco para reflejar el borrado.
- **Modify (modificar).** Modifica los valores de algunos campos del registro actual y (finalmente) actualiza el fichero en disco para reflejar la modificación.
- **Insert (insertar).** Inserta un registro nuevo en el fichero localizando el bloque donde se insertará el registro, transfiriendo ese bloque a un búfer de la memoria principal (si todavía no se encuentra allí), escribiendo el registro en el búfer y (finalmente) escribiendo el búfer en el disco para reflejar la inserción.
- **Close (cerrar).** Completa el acceso al fichero liberando los búferes y ejecutando cualquier otra operación de limpieza necesaria.

Las operaciones anteriores (excepto las de abrir y cerrar) se conocen como operaciones de **un registro por vez**, porque cada operación se aplica a un solo registro. Es posible aglutinar las operaciones Find, FindNext y Read en una sola operación, Scan, cuya descripción es la siguiente:

- **Scan.** Si el fichero simplemente se ha abierto o reiniciado, *Scan* devuelve el primer registro; en caso contrario, devuelve el siguiente registro. Si con la operación especificamos una condición, el registro devuelto es el primero o el siguiente que satisface esa condición.

En los sistemas de bases de datos, se pueden aplicar **operaciones de grupo** al fichero, como, por ejemplo, las siguientes:

- **FindAll (buscar todo).** Localiza *todos* los registros del fichero que satisfacen una condición de búsqueda.
- **Find (o Locate) *n* (buscar *n*).** Busca el primer registro que satisface una condición de búsqueda y, después, continúa localizando los $n - 1$ siguientes registros que satisfacen la misma condición. Transfiere los bloques que contienen los $n$ registros a un búfer de la memoria principal (si todavía no están allí).

- **FindOrdered (buscar ordenados).** Recupera todos los registros del fichero en un orden específico.
- **Reorganize (reorganizar).** Inicia el proceso de reorganización. Como veremos, algunas organizaciones de ficheros requieren una reorganización periódica. Un ejemplo puede ser reordenar los registros del fichero ordenándolos por un campo específico.

En este punto, conviene observar la diferencia entre los términos **organización del fichero** y **método de acceso**. El primero se refiere a la organización de los datos de un fichero en registros, bloques y estructuras de acceso; esto incluye la forma en que los registros y los bloques se colocan e interconectan en el medio de almacenamiento. Un método de acceso, por el contrario, proporciona un grupo de operaciones (como las enumeradas anteriormente) que se pueden aplicar a un fichero. En general, es posible aplicar varios métodos de acceso a una organización de fichero. Algunos, no obstante, sólo se pueden aplicar a ficheros organizados de determinadas formas. Por ejemplo, no podemos aplicar un método de acceso indexado a un fichero que no tiene un índice (consulte el Capítulo 14).

Normalmente, esperamos utilizar unas condiciones de búsqueda más que otras. Algunos ficheros pueden ser **estáticos**, pues es raro ejecutar sobre ellos operaciones de actualización; otros, los denominados archivos **dinámicos**, pueden cambiar más a menudo pues se les están aplicando operaciones de actualización constantemente. Una organización de fichero satisfactoria debe ejecutar tan eficazmente como sea posible las operaciones que esperamos aplicar frecuentemente al fichero. Por ejemplo, el fichero EMPLEADO de la Figura 13.5(a) almacena los registros de los empleados actuales de la empresa. Esperamos insertar registros (cuando se contratan empleados), eliminar registros (cuando los empleados abandonan la empresa) y modificar registros (por ejemplo, cuando se modifica el sueldo o el puesto de trabajo de un empleado). La eliminación o modificación de un registro requiere una condición de selección para identificar un registro o un conjunto de registros concreto. La recuperación de uno o más registros también requiere una condición de selección.

Si los usuarios prevén aplicar una condición de búsqueda basada principalmente en el DNI, el diseñador debe elegir una organización de fichero que facilite la localización de un registro dado su valor de Dni. Esto puede implicar que los registros se ordenen físicamente por el valor de Dni o que se deba crear un índice con los valores de Dni (consulte el Capítulo 14). Vamos a suponer que una segunda aplicación utiliza el fichero para generar las nóminas de los empleados, y que requiere que esas nóminas estén agrupadas por departamento. Para esta aplicación, es recomendable almacenar contiguamente los registros de todos los empleados que tienen el mismo valor de departamento, agrupándolos en bloques y, quizá, ordenándolos por su nombre dentro de cada departamento. No obstante, esta distribución entra en conflicto con la ordenación de los registros por sus valores de DNI. Si las dos aplicaciones son importantes, el diseñador debe elegir una organización que permita llevar a cabo eficazmente ambas operaciones. Por desgracia, en muchos casos no puede haber una organización que permita la implementación eficaz de todas las operaciones necesarias. En estos casos, hay que tener en cuenta la importancia esperada y la mezcla de las operaciones de recuperación y actualización.

En las siguientes secciones y en el Capítulo 14 explicamos los métodos para organizar los registros de un fichero en el disco. Para crear los métodos de acceso se utilizan diversas técnicas generales, como la ordenación, la dispersión y la indexación. Además, con muchas de las organizaciones de ficheros funcionan diversas técnicas generales de manipulación de inserciones y borrados.

## 13.6 Ficheros de registros desordenados (ficheros heap)

Es el tipo de organización más sencillo y básico, según el cual los registros se guardan en el fichero en el mismo orden en que se insertan; es decir, los registros se insertan al final del fichero. Esta organización se conoce como **fichero heap o pila**.[7] Se utiliza a menudo con rutas de acceso adicionales, como los índices

[7] En ocasiones, a esta organización se la conoce como **fichero secuencial**.

secundarios que explicamos en el Capítulo 14. También se utiliza para recopilar y almacenar registros de datos para un uso futuro.

La inserción de un registro nuevo es muy *eficaz*. El último bloque de disco del fichero se copia en el búfer, se añade el registro nuevo y se **reescribe** el bloque de nuevo en el disco. En la cabecera del fichero se guarda la dirección del último bloque del fichero. Sin embargo, la búsqueda de un registro utilizando cualquier condición de búsqueda implica una **búsqueda lineal**, bloque a bloque, por todo el fichero (un procedimiento muy costoso). Si sólo un registro satisface la condición de búsqueda, entonces, por término medio, un programa leerá en memoria y buscará en la mitad de los bloques antes de encontrar el registro. En el caso de un fichero de $b$ bloques, esto requiere buscar de media en ($b$/2) bloques. Si ningún registro satisface la condición de búsqueda, el programa debe buscar en los $b$ bloques del fichero.

Para eliminar un registro, un programa primero debe buscar su bloque, copiar el bloque en un búfer, eliminar el registro del búfer y, por último, **reescribir el bloque** en el disco. Esto deja espacio inutilizado en el bloque de disco. La eliminación de una gran cantidad de registros de este modo supone un derroche de espacio de almacenamiento. Otra técnica que se utiliza para borrar registros es contar con un byte o un bit extra, denominado **marcador de borrado**, en cada registro. Un registro se borra estableciendo el marcador de borrado a un determinado valor. Un valor diferente del marcador indica un registro válido (no borrado). Los programas de búsqueda sólo consideran los registros válidos cuando realizan sus búsquedas. Estas dos técnicas de borrado requieren una **reorganización** periódica del fichero para reclamar el espacio inutilizado correspondiente a los registros borrados. Durante la reorganización, se accede consecutivamente a los bloques del fichero, y los registros se empaquetan eliminando los registros borrados. Tras la reorganización, los bloques se llenan de nuevo en toda su capacidad. Otra posibilidad es utilizar el espacio de los registros borrados al insertar registros nuevos, aunque esto requiere una contabilidad adicional para hacer un seguimiento de las ubicaciones vacías.

Podemos utilizar la organización extendida o la no extendida para un fichero desordenado, y podemos utilizarla con registros de longitud fija o de longitud variable. La modificación de un registro de longitud variable puede requerir el borrado del registro antiguo y la inserción del registro modificado, porque puede ser que este último no encaje en su antiguo espacio en disco.

Para leer todos los registros en orden por los valores de algún campo, debemos crear una copia ordenada del fichero. La ordenación es una operación costosa si el fichero es grande, y para ella se utilizan técnicas especiales de **ordenación externa** (consulte el Capítulo 15).

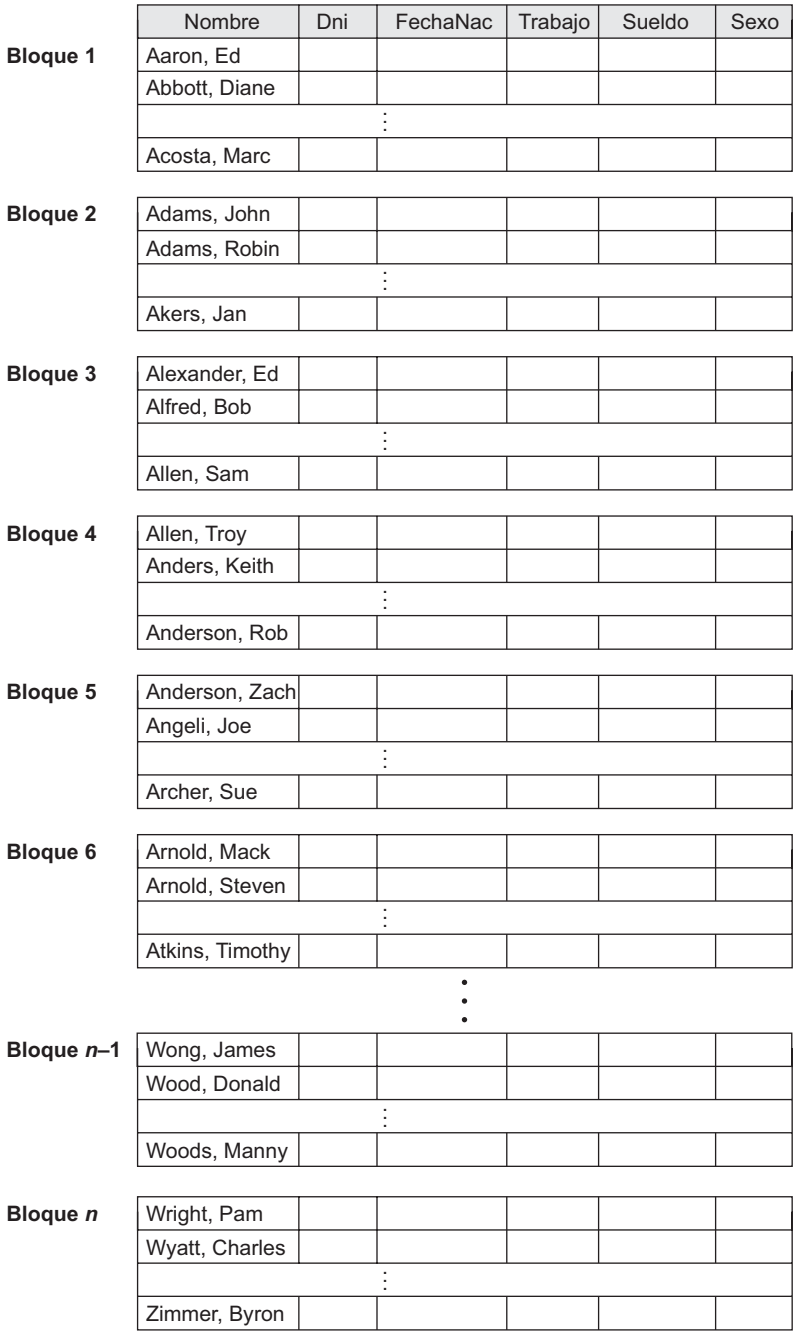
En un fichero desordenado de *registros de longitud fija* que utiliza *bloques extendidos* y la *asignación contigua*, es inmediato el acceso a cualquier registro por su **posición** en el fichero. Si los registros del fichero están numerados como 0, 1, 2,..., $r - 1$ y los registros de cada bloque están numerados como 0, 1,..., $bfr - 1$, donde $bfr$ es el factor de bloqueo, entonces el registro $i$-nésimo del fichero se encuentra en el bloque $\lfloor (i/bfr) \rfloor$ y es el registro número ($i$ mod $bfr$) de ese bloque. Este tipo de fichero suele recibir el nombre de **fichero relativo** o **directo** porque es fácil acceder directamente a los registros por sus posiciones relativas. El acceso a un registro por su posición no ayuda a localizar un registro basándose en una condición de búsqueda; sin embargo, facilita la construcción de rutas de acceso en el fichero, como los índices que explicamos en el Capítulo 14.

## 13.7 Ficheros de registros ordenados (ficheros ordenados)

Los registros de un fichero se pueden ordenar físicamente en el disco en función de los valores de uno de sus campos, denominado **campo de ordenación**. Esto conduce a un **fichero ordenado** o **secuencial**.[8] Si el campo

[8] El término *fichero secuencial* también se ha utilizado para referirse a los ficheros desordenados.

**Figura 13.7.** Algunos bloques de un fichero ordenado (secuencial) de registros EMPLEADO con Nombre como campo clave de ordenación.




de ordenación también es un **campo clave** (un campo que garantiza un valor exclusivo en cada registro) del fichero, entonces el campo se denomina **clave de ordenación** del fichero. La Figura 13.7 muestra un fichero ordenado con Nombre como campo clave de ordenación (asumiendo que los empleados tienen nombres distintos).

Los registros ordenados tienen algunas ventajas sobre los ficheros desordenados. En primer lugar, la lectura de los registros en el orden marcado por los valores de la clave de ordenación es extremadamente eficaz porque no se necesita una ordenación. En segundo lugar, encontrar el siguiente registro al actual según el orden de la clave de ordenación normalmente no requiere acceder a bloques adicionales porque el siguiente registro se encuentra en el mismo bloque que el actual (a menos que el registro actual sea el último del bloque). En tercer lugar, el uso de una condición de búsqueda basándose en el valor de un campo clave de ordenación ofrece un acceso más rápido cuando se utiliza la técnica de búsqueda binaria, que constituye una mejora respecto a las búsquedas lineales, aunque no se utiliza habitualmente para los ficheros de disco.

Una **búsqueda binaria** en ficheros de disco puede realizarse en los bloques en lugar de en los registros. Vamos a suponer que el fichero tiene $b$ bloques numerados como 1, 2,..., $b$; los registros están ordenados ascendentemente por el valor de su campo clave de ordenación; y estamos buscando un registro cuyo campo clave de ordenación tenga el valor $K$. Asumiendo que las direcciones de disco de los bloques del fichero están disponibles en la cabecera del fichero, la búsqueda binaria puede describirse con el Algoritmo 13.1. Una búsqueda binaria normalmente accede a $\log2(b)$ bloques, se encuentre o no el registro [una mejora respecto a las búsquedas lineales, en las que, por término medio, se accede a ($b$/2) bloques cuando se encuentra el registro y a $b$ bloques cuando el registro no se encuentra].

**Algoritmo 13.1. Búsqueda binaria en una clave de ordenación de un fichero de disco.**

```
l ← 1; u ← b; (* b es el número de bloques del fichero *)
while (u ≥ l ) do
   begin i ← (l + u) div 2;
   leer el bloque i del fichero en el búfer;
   if K < (valor del campo clave de ordenación del primer registro en el bloque i)
      then u ← i − 1
   else if K > (valor del campo clave de ordenación del último registro en el bloque i)
      then l ← i + 1
   else if el registro con el campo clave de ordenación = K está en el búfer
      then goto encontrado
   else goto noencontrado;
   end;
goto noencontrado;
```

Un criterio de búsqueda que implica las condiciones >, <, ≥ y ≤, es muy eficaz, porque la ordenación física de los registros significa que todos los registros que satisfacen la condición están contiguos en el fichero. Por ejemplo, y en referencia a la Figura 13.9, si el criterio de búsqueda es (Nombre < 'G'), donde < significa *alfabéticamente anterior*, los registros que satisfacen el criterio de búsqueda son los que se encuentran entre el principio del fichero y el primer registro cuyo campo Nombre tiene un valor que empieza por la letra 'G'.

La ordenación no ofrece ninguna ventaja para el acceso aleatorio u ordenado de los registros basándose en los valores de otros *campos no ordenados* del fichero. En estos casos, realizamos una búsqueda lineal para el acceso aleatorio. Para acceder a los registros en orden basándose en un campo desordenado, es necesario crear otra copia ordenada (con un orden diferente) del fichero.

La inserción y eliminación de registros son operaciones costosas para un archivo ordenado, porque los registros deben permanecer físicamente ordenados. Para insertar un registro debemos encontrar su posición correcta en el fichero, basándonos en el valor de su campo de ordenación, y después hacer espacio en el fichero para insertar el registro en esa posición. Si se trata de un fichero grande, esta operación puede consumir mucho tiempo porque, por término medio, han de moverse la mitad de los registros del fichero para hacer espacio al nuevo registro. Esto significa que deben leerse y reescribirse la mitad de los bloques del fichero después de

haber movido los bloques entre sí. En la eliminación de un registro, el problema es menos grave si se utilizan marcadores de borrado y una reorganización periódica.

Una opción para que la inserción sea más eficaz es reservar algo de espacio sin usar en cada bloque de cara a los registros nuevos. Sin embargo, una vez que se agota este espacio, resurge el problema original. Otro método que se utiliza con frecuencia es crear un fichero temporal desordenado denominado **fichero de desbordamiento** (***overflow***) o de **transacciones**. Con esta técnica, el fichero ordenado actual se denomina **fichero principal** o **maestro**. Los registros nuevos se insertan al final del fichero de desbordamiento, en lugar de hacerlo en su posición correcta en el fichero principal. Periódicamente, el fichero de desbordamiento se ordena y mezcla con el maestro durante la reorganización del fichero. La inserción es muy eficaz, pero a costa de aumentar la complejidad del algoritmo de búsqueda. La búsqueda en el fichero de desbordamiento debe hacerse con una búsqueda lineal si, después de la búsqueda binaria, no se encuentra el registro en el fichero principal. En las aplicaciones que no requieren la mayoría de la información actualizada pueden ignorarse los registros de desbordamiento durante una búsqueda.

La modificación del valor de un campo depende de dos factores: la condición de búsqueda para localizar el registro y el campo que se va a modificar. Si la condición de búsqueda involucra al campo clave de ordenación, podemos localizar el registro utilizando una búsqueda binaria; en caso contrario, debemos hacer una búsqueda lineal. Un campo no ordenado se puede modificar cambiando el registro y reescribiendo en la misma ubicación física del disco (si los registros son de longitud fija). La modificación del campo de ordenación significa que el registro puede cambiar su posición en el fichero. Esto requiere la eliminación del registro antiguo, seguida por la inserción del registro modificado.

La lectura de los registros del fichero en el orden especificado por el campo de ordenación es bastante eficaz si ignoramos los registros desbordados, puesto que con el doble búfer podemos leer los bloques consecutivamente. Para incluir los registros desbordados, debemos mezclarlos en sus posiciones correctas; en este caso, primero tenemos que reorganizar el fichero y después leer sus bloques secuencialmente. Para reorganizar el fichero, primero ordenamos los registros del fichero de desbordamiento, y después los mezclamos con el fichero maestro. Los registros marcados para su borrado se eliminan durante la reorganización.

La Tabla 13.2 resume el tiempo de acceso medio a los bloques para encontrar un registro específico en un fichero que tiene *b* bloques.

Los ficheros ordenados raramente se utilizan en las aplicaciones de bases de datos, a menos que se utilice una ruta de acceso adicional, denominada **índice principal**; el resultado es un **fichero indexado-secuencial**. Esto mejora el tiempo de acceso aleatorio al campo clave de ordenación. En el Capítulo 14 explicamos los índices. Si el atributo de ordenación no es una clave, el fichero se denomina **fichero agrupado**.

## 13.8 Técnicas de dispersión

Otro tipo de organización de ficheros está basado en la dispersión, que proporciona un acceso muy rápido a los registros bajo ciertas condiciones de búsqueda. Esta organización se denomina normalmente **fichero disperso** o fichero *hash*.[9] La condición de búsqueda debe ser una condición de igualdad sobre un solo campo, denominado **campo de dispersión** o campo *hash*. En la mayoría de los casos, el campo de dispersión también es un campo clave del fichero, en cuyo caso se denomina **clave de dispersión** (o clave *hash*). La idea que hay detrás de la dispersión es proporcionar una función *h*, denominada **función de dispersión** (o *hash*) o **función de aleatorización**, que se aplica al valor del campo de dispersión de un registro y produce la *dirección* del bloque de disco en el que está almacenado el registro. Una búsqueda del registro dentro del bloque puede llevarse a cabo en un búfer de la memoria principal. En la mayoría de los registros, sólo necesitamos acceder a un bloque para recuperar el registro.

[9] Un fichero disperso también recibe el nombre de *fichero directo*.

**Tabla 13.2.** Tiempos de acceso medio para un fichero de *b* bloques bajo organizaciones de fichero básicas.

| Tipo de organización | Método de acceso/búsqueda | Bloques promedio para acceder a un registro concreto |
|---|---|---|
| *Heap* (desordenado) | Exploración secuencial (búsqueda lineal) | $b/2$ |
| Ordenado | Exploración secuencial | $b/2$ |
| Ordenado | Búsqueda binaria | $\log_2 b$ |

La dispersión también se utiliza como una estructura de búsqueda interna dentro de un programa, siempre que se acceda a un grupo de registros exclusivamente utilizando el valor de un campo. En la Sección 13.8.1 describimos el uso de la dispersión para los ficheros internos; después, en la Sección 13.8.2, mostramos cómo se modifica para almacenar ficheros externos en disco. En la Sección 13.8.3 explicamos las técnicas para extender la dispersión a los ficheros de crecimiento dinámico.

## 13.8.1 Dispersión interna

En los ficheros internos, la dispersión normalmente se implementa como una **tabla de dispersión** mediante el uso de un array de registros. Vamos a suponer que el índice del array va de 0 a $M - 1$ (véase la Figura 13.8[a]); por tanto, tenemos $M$ **slots** cuyas direcciones corresponden a los índices del array. Elegimos una función de dispersión que transforma el valor del campo de dispersión en un entero entre 0 y $M - 1$. Una función de dispersión común es la función $h(K) = K$ **mod** $M$, que devuelve el resto del valor entero de un campo de dispersión $K$ después de dividirlo por $M$; este valor se utiliza después para la dirección del registro.

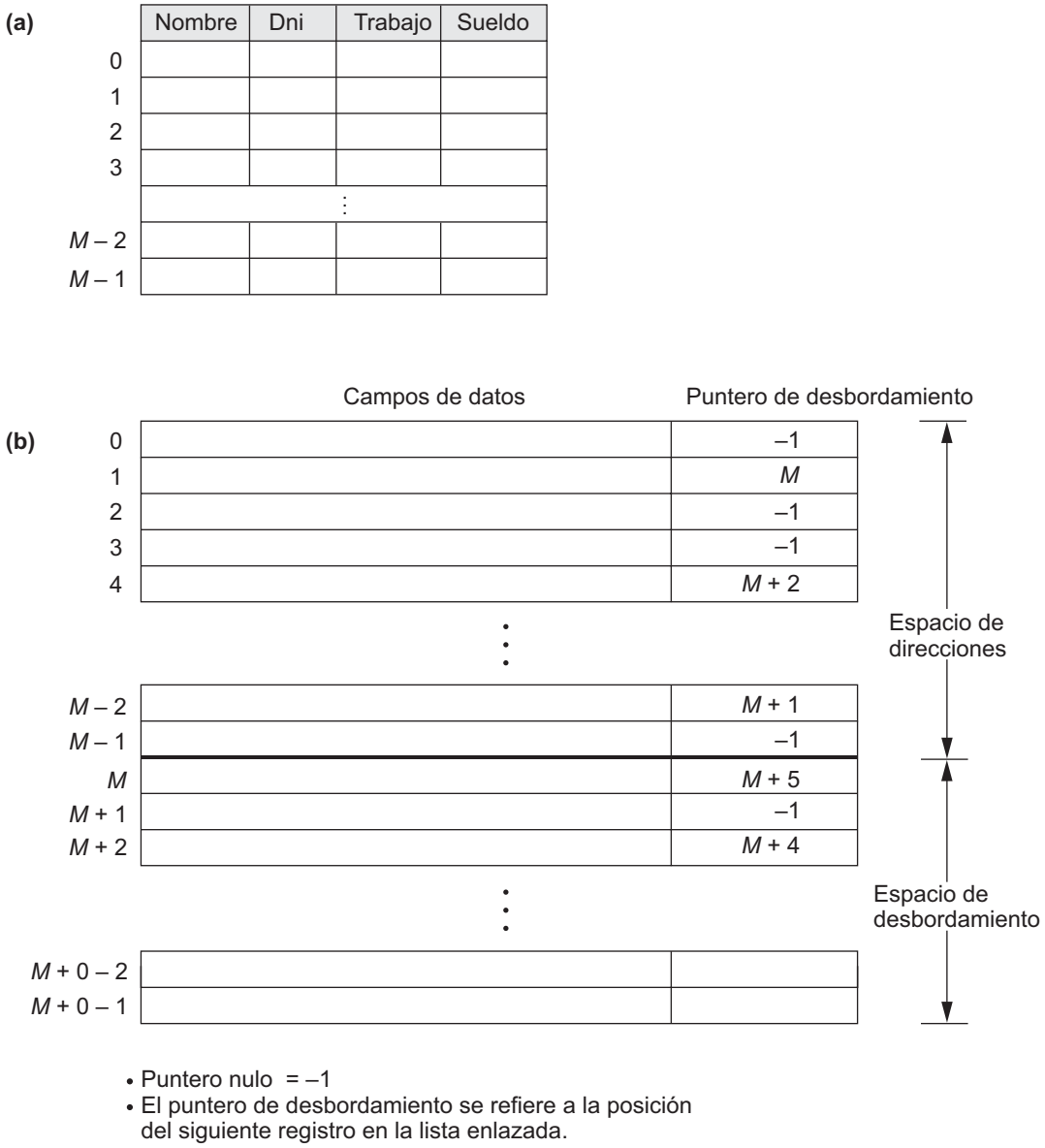
Los valores no enteros de un campo de dispersión se pueden transformar en enteros antes de aplicar la función mod. En las cadenas de caracteres, podemos utilizar en la transformación los códigos numéricos (ASCII) asociados a los caracteres (por ejemplo, multiplicar estos códigos). Para un campo de dispersión cuyo tipo de datos es una cadena de 20 caracteres, es posible utilizar el Algoritmo 13.2(a) para calcular la dirección de dispersión. Asumimos que la función devuelve el código numérico de un carácter y que tenemos un valor de campo de dispersión $K$ de tipo *K: array* [1..20] *of char* (en PASCAL) o *char K*[20] (en C).

**Algoritmo 13.2. Dos algoritmos de dispersión simples.** (a) Aplicación de la función de dispersión mod a una cadena de caracteres *K*. (b) Resolución de la colisión por direccionamiento abierto.

```
(a) temp ← 1;
    for i ← 1 to 20 do temp ← temp * code(K[i ] ) mod M ;
    dirección_dispersión ← temp mod M;
(b) i ← dirección_dispersión(K); a ← i;
    if ubicación i está ocupada
       then begin i ← (i + 1) mod M;
        while (i ≠ a) y la ubicación i esté ocupada
          do i ← (i + 1) mod M;
          if (i = a) then todas las posiciones están llenas
          else nueva_dirección_dispersión ← i;
          end;
```

También se pueden utilizar otras funciones de dispersión. Una técnica, denominada **reversible**, implica la aplicación de una función aritmética como una *suma* o una función lógica como un *or exclusivo* a diferentes porciones del valor del campo de dispersión para calcular la dirección de dispersión. Otra técnica implica

**Figura 13.8.** Estructuras de datos de dispersión interna. (a) Array de *M* posiciones para su uso en la dispersión interna. (b) Resolución de la colisión mediante encadenación de registros.




tomar algunos dígitos del valor del campo de dispersión (por ejemplo, los dígitos tercero, quinto y octavo) para formar la dirección de dispersión.[10] El problema con la mayoría de las funciones de dispersión es que no garantizan que valores distintos se dispersen a direcciones distintas, porque el **espacio del campo de dispersión** (el número de valores posibles que un campo de dispersión puede tomar) es normalmente mucho más grande que el **espacio de direcciones** (el número de direcciones disponibles para los registros). La función de dispersión mapea el espacio del campo de dispersión al espacio de direcciones.

[10] Una explicación detallada de las funciones de dispersión queda fuera del objetivo de nuestra presentación.

Una **colisión** se produce cuando el valor del campo de dispersión de un registro que se está insertando se dispersa a una dirección que ya contiene un registro diferente. En esta situación, debemos insertar el registro nuevo en alguna otra posición, puesto que su dirección de dispersión está ocupada. El proceso de encontrar otra posición se denomina **resolución de colisiones**. Hay varios métodos para resolver una colisión:

- **Direccionamiento abierto.** A partir de la posición ocupada especificada por la dirección de dispersión, el programa comprueba las posiciones subsiguientes en orden hasta encontrar una posición sin utilizar (vacía). El Algoritmo 13.2(b) puede utilizarse con este propósito.
- **Encadenamiento.** Para este método, se conservan varias ubicaciones de dispersión, normalmente extendiendo el array con algunas posiciones de desbordamiento. Adicionalmente, se añade un campo puntero a cada ubicación de registro. Una colisión se resuelve colocando el registro nuevo en una ubicación de desbordamiento sin utilizar y estableciendo el puntero de la ubicación de la dirección de dispersión ocupada a la dirección de esa ubicación de desbordamiento. Se conserva entonces una lista enlazada de registros de desbordamiento por cada dirección de dispersión, como se muestra en la Figura 13.8(b).
- **Dispersión múltiple.** El programa aplica una segunda función de dispersión si la primera desemboca en una colisión. Si se produce otra colisión, el programa utiliza el desbordamiento abierto o aplica una tercera función de dispersión y utiliza después el direccionamiento abierto si es necesario.

Todo método de resolución de colisiones requiere sus propios algoritmos para insertar, recuperar y borrar registros. Los algoritmos de encadenamiento son los más sencillos. Los de borrado para el direccionamiento abierto son bastante difíciles. Los libros sobre estructuras de datos explican más en profundidad los algoritmos de dispersión interna.

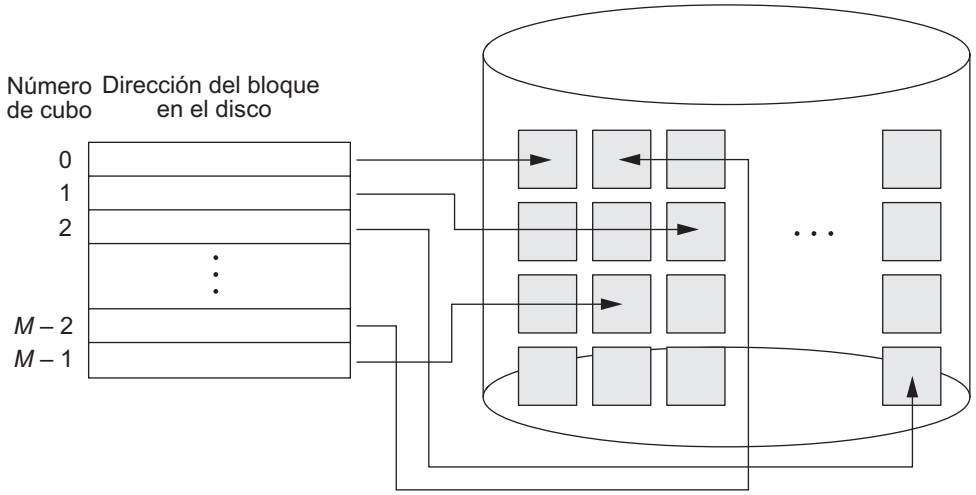
El objetivo de una buena función de dispersión es distribuir uniformemente los registros sobre el espacio de direcciones para minimizar las colisiones sin dejar demasiadas ubicaciones sin utilizar. Los estudios de simulación y análisis han mostrado que normalmente es mejor mantener una tabla de dispersión entre el 70 y el 90 por ciento llena, para que el número de colisiones sea bajo y no tengamos que malgastar demasiado espacio. Por tanto, si esperamos almacenar $r$ registros en la tabla, debemos elegir $M$ ubicaciones para el espacio de direcciones de tal modo que ($r$/$M$) esté entre 0,7 y 0,9. También puede ser útil elegir un número primo para $M$, ya que se ha demostrado que esto distribuye mejor las direcciones de dispersión sobre el espacio de direcciones cuando se utiliza la función de dispersión mod. Otras funciones de dispersión pueden requerir que $M$ sea una potencia de 2.

## 13.8.2 Dispersión externa para los ficheros de disco

La dispersión para los ficheros de disco se denomina **dispersión externa**. Para ajustar las características del almacenamiento en disco, el espacio de direcciones de destino se compone de **cubos**, cada uno de los cuales almacena varios registros. Un cubo puede ser un bloque de disco o un grupo de bloques contiguos. La función de dispersión mapea una clave a un número de cubo relativo, en lugar de asignar una dirección de bloque absoluta al cubo. Una tabla almacenada en la cabecera del fichero convierte el número de cubo en la correspondiente dirección de bloque del disco (véase la Figura 13.9).

El problema de las colisiones es menos grave con los cubos, porque cuantos más registros encajen en un cubo más posible será que se dispersen al mismo cubo sin causar problemas. Sin embargo, debemos prever que el cubo puede estar lleno y que un registro nuevo se esté dispersando a ese cubo. Podemos utilizar una variación del encadenamiento según la cual se mantiene un puntero en cada cubo apuntando a una lista enlazada de registros de desbordamiento para el cubo (véase la Figura 13.10). Los punteros de la lista enlazada deben ser **punteros de registro**, que incluyen tanto una dirección de bloque como una posición relativa de registro dentro de bloque.

**Figura 13.9.** Emparejamiento de los números de cubo con direcciones de bloques de disco.



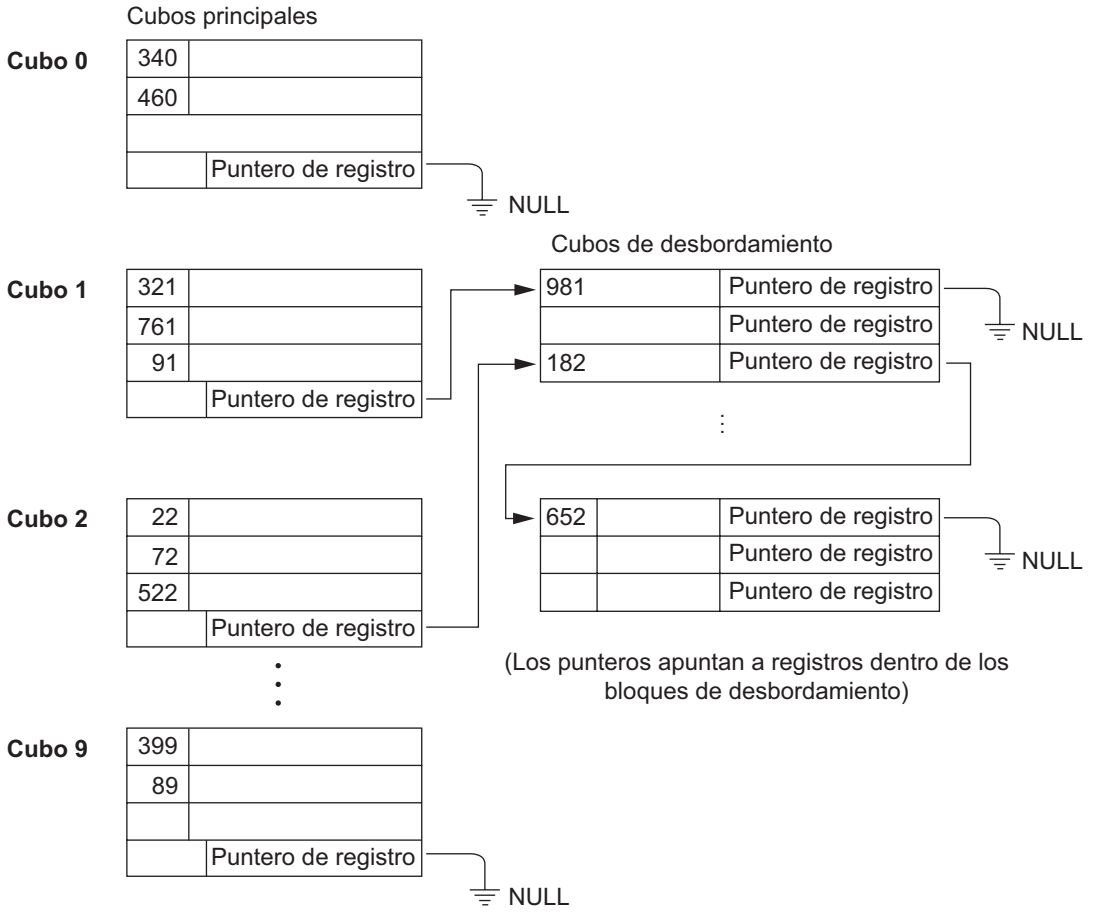

La dispersión proporciona el acceso posible más rápido para recuperar un registro arbitrario dado el valor de su campo de dispersión. Aunque la mayoría de las mejores funciones de dispersión no mantienen ordenados los registros por los valores del campo de dispersión, algunas funciones (denominadas de **preservación del orden**) sí lo hacen. Un ejemplo sencillo de función de dispersión con conservación del orden es tomar los tres dígitos situados a la izquierda de un campo de número de factura como dirección de dispersión, y mantener ordenados los registros por el número de factura dentro de cada cubo. Otro ejemplo es utilizar una clave de dispersión entera directamente como índice a un fichero relativo, si los valores de la clave de dispersión llenan un intervalo concreto; por ejemplo, si los números de empleado de una empresa son asignados como 1, 2, 3,... hasta el número total de empleados, podemos utilizar la función de dispersión de identidad que mantiene el orden. Por desgracia, esto sólo funciona si las claves son generadas en orden por alguna aplicación.

El esquema de dispersión descrito se denomina **dispersión estática** porque se asigna un número fijo de $M$ cubos. Esto puede ser un gran inconveniente para los ficheros dinámicos. Suponga que asignamos $M$ cubos para el espacio de direcciones y permitimos que $m$ sea el número máximo de registros que podemos encajar en un cubo; por tanto, en el espacio asignado encajarán a lo sumo ($m$ * $M$) registros. Si el número de registros fuera a ser sustancialmente inferior a ($m$ * $M$), acabaríamos con mucho espacio desaprovechado. Por el contrario, si el número de registros sobrepasa sustancialmente la cifra de ($m$ * $M$), tendremos numerosas colisiones y la recuperación será más lenta debido a las largas listas de registros desbordados. En cualquier caso, puede que tengamos que cambiar el número de $M$ bloques asignado y utilizar entonces una nueva función de dispersión (basándonos en el nuevo valor de $M$) para redistribuir los registros. Estas reorganizaciones pueden consumir mucho tiempo si el fichero es grande. Las organizaciones de fichero dinámicas más nuevas basadas en la dispersión permiten que los cubos varíen dinámicamente sólo con la reorganización localizada (consulte la Sección 13.8.3).

Cuando utilizamos la dispersión externa, la búsqueda de un registro dado el valor de algún otro campo distinto al campo de distribución es tan costosa como en el caso de un fichero desordenado. La eliminación de un registro se puede implementar eliminando el registro de su cubo. Si el cubo tiene una cadena de desbordamiento, podemos mover uno de los registros de desbordamiento del cubo para reemplazar el registro borrado. Si el registro que vamos a borrar ya está desbordado, simplemente lo eliminamos de la lista enlazada. La eliminación de un registro desbordado implica hacer un seguimiento de las posiciones vacías del desbordamiento. Esto se consigue fácilmente manteniendo una lista enlazada de ubicaciones de desbordamiento inutilizadas.

La modificación del valor de un campo del registro depende de dos factores: la condición de búsqueda para localizar el registro y el campo que se va a modificar. Si la condición de búsqueda es una comparación de

**Figura 13.10.** Manipulación del desbordamiento de cubos por encadenamiento.




igualdad en el campo de dispersión, podemos localizar el registro eficazmente utilizando la función de dispersión; en caso contrario, debemos realizar una búsqueda lineal. Un campo que no es de dispersión se puede modificar cambiando el registro y reescribiéndolo en el mismo cubo. La modificación del campo de dispersión significa que el registro se puede mover a otro cubo, lo que requiere la eliminación del registro antiguo seguida de la inserción del registro modificado.

### 13.8.3 Técnicas de dispersión que permiten la expansión dinámica del fichero

Un importante inconveniente del esquema de dispersión *estática* que acabamos de ver es que el espacio de direcciones de dispersión es fijo. Por tanto, resulta difícil expandir o contraer dinámicamente el fichero. Los esquemas descritos en esta sección intentan remediar esta situación. El primer esquema (la dispersión extensible) almacena una estructura de acceso además del fichero, por lo que se parece a la indexación (consulte el Capítulo 14). La principal diferencia es que la estructura de acceso está basada en los valores resultantes después de aplicar la función de dispersión al campo de búsqueda. En la indexación, la estructura de acceso está basada en los valores del propio campo de búsqueda. La segunda técnica, denominada dispersión lineal, no requiere estructuras de acceso adicionales.

Estos esquemas de dispersión tienen la ventaja de que el resultado de aplicar una función de dispersión es un entero positivo y que puede representarse como un número binario. La estructura de acceso se fundamenta en la **representación binaria** del resultado de la función de dispersión, que es un cadena de **bits**; lo llamamos **valor de dispersión** de un registro. Los registros se distribuyen entre cubos basándose en los valores de los *primeros bits* de sus valores de dispersión.

**Dispersión extensible**. En esta dispersión, se conserva un tipo de directorio (un array de $2^d$ direcciones de cubos), donde $d$ se conoce como **profundidad global** del directorio. El valor entero correspondiente a los $d$ primeros (orden superior) bits de un valor de dispersión se utiliza como índice del array para determinar una entrada del directorio, y la dirección de esa entrada determina el cubo en el que se almacenan los registros correspondientes.

Sin embargo, no hay que tener un cubo distinto para cada una de las $2^d$ ubicaciones de directorio. Varias ubicaciones de directorio con los mismos $d'$ primeros bits para sus valores de dispersión pueden contener la misma dirección de cubo si todos los registros que se dispersan a estas ubicaciones encajan en un solo cubo. Una **profundidad local d'** (almacenada con cada cubo) especifica el número de bits en los que está basado el contenido del cubo. La Figura 13.13 muestra un directorio con un profundidad local $d = 3$.

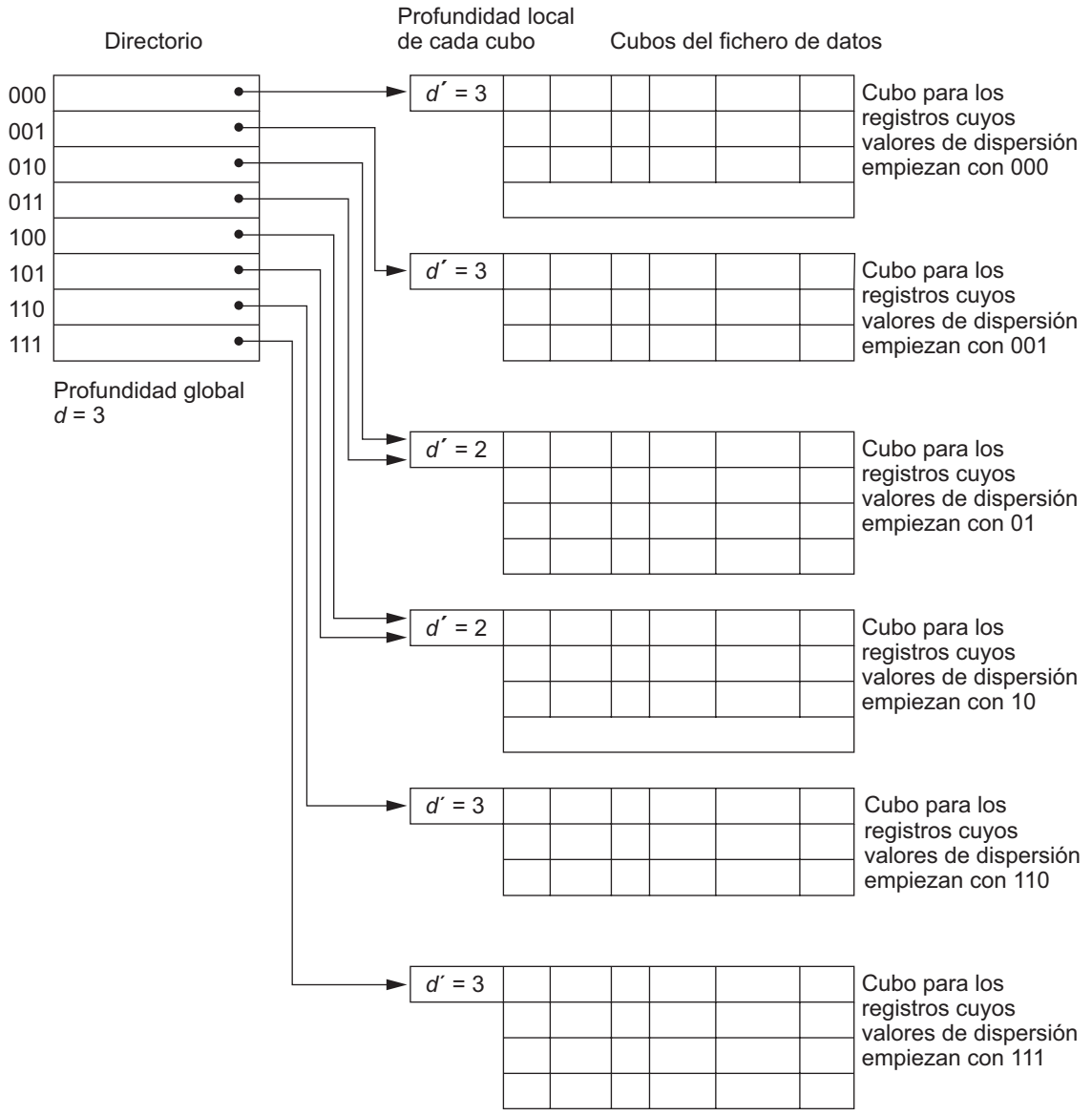
El valor de $d$ se puede incrementar o reducir de uno en uno, doblando o reduciendo a la mitad de este modo el número de entradas del array de directorio. La opción de la duplicación se necesita si un cubo, cuya profundidad local *d'* es igual a la profundidad global $d$, desborda. La reducción a la mitad se da si $d >$ d' para todos los cubos después de producirse algunos borrados. La mayoría de las recuperaciones de registros requieren dos accesos de bloque: uno al directorio y otro al cubo.

Para ilustrar la división del cubo, suponga que la inserción de un registro nuevo provoca el desbordamiento en el cubo cuyos valores de dispersión empiezan con 01 (el tercer cubo de la Figura 13.13). Los registros se distribuirán entre dos cubos: el primero contiene todos los registros cuyos valores de dispersión empiezan por 010, y el segundo todos los que sus valores de dispersión empiezan con 011. Ahora, las dos ubicaciones de directorio para 010 y 011 apuntan a los dos cubos nuevos y distintos. Antes de la división, apuntaban al mismo cubo. La profundidad local *d'* de los dos cubos nuevos es 3, uno más que la profundidad local del cubo antiguo.

Si un cubo que se desborda y se divide tenía una profundidad local $d'$ igual a la profundidad global $d$ del directorio, entonces el tamaño de este último debe duplicarse ahora para que podamos utilizar un bit adicional a fin de distinguir los dos cubos nuevos. Por ejemplo, si el cubo de los registros cuyos valores de dispersión empiezan con 111 de la Figura 13.11 se desborda, los dos cubos nuevos necesitan un directorio con una profundidad global de $d = 4$, porque los dos cubos aparecen ahora etiquetados como 1110 y 1111, y por tanto sus profundidades locales son 4. El tamaño del directorio se ha doblado, y cada una de las ubicaciones originales del directorio también se ha dividido en dos ubicaciones, y ambas tienen el mismo valor de puntero que la ubicación original.

La principal ventaja de la dispersión extensible es que el rendimiento del fichero no se degrada al crecer, al contrario de lo que ocurre con la dispersión externa estática, con la que aumentan las colisiones y el encadenamiento correspondiente provoca accesos adicionales. Además, en la dispersión extensible no se asigna espacio para un crecimiento futuro, sino que se asignan dinámicamente cubos adicionales a medida que se necesitan. La sobrecarga de espacio para la tabla de directorio es insignificante. El tamaño máximo del directorio es $2^k$, donde $k$ es el número de bits del valor de dispersión. Otra ventaja es que la división provoca una menor reorganización en la mayoría de los casos, por lo que únicamente los registros de un cubo se redistribuyen a los dos cubos nuevos. La reorganización más costosa en cuanto a tiempo es la de duplicar el directorio (o dividirlo en dos). Un inconveniente es que antes de acceder a los propios cubos, debe buscarse el directorio, lo que da lugar a dos accesos de bloque en lugar de uno, como ocurre en la dispersión estática. Esta penalización en el rendimiento se considera menor y, por tanto, se considera que este esquema es muy recomendable para los ficheros dinámicos.

**Figura 13.11.** Estructura del esquema de dispersión extensible.




**Dispersión lineal.** La idea que hay detrás de la dispersión lineal es permitir el aumento y la reducción dinámica del número de cubos de un fichero de dispersión *sin* necesidad de un directorio. Supongamos que el fichero empieza con $M$ cubos numerados como 0, 1,...,$M - 1$ y utiliza la función de dispersión mod $h(K) = K$ mod $M$; esta función de dispersión se denomina función de dispersión lineal $h_i$. Todavía se necesita el desbordamiento debido a las colisiones y puede manipularse manteniendo cadenas de desbordamiento individuales para cada cubo. Sin embargo, cuando una colisión lleva a un registro desbordado en *cualquier* cubo del fichero, el *primer* cubo del fichero (el cubo 0) se divide en dos cubos: el cubo 0 original y un nuevo cubo $M$ al final del fichero. Los registros que se encontraban originalmente en el cubo 0 se distribuyen entre los dos cubos basándose en una función de dispersión diferente, $h_{i+1}(K) = K$ mod $2M$. Una propiedad importante de las dos

funciones de dispersión $h_i$ y $h_{i+1}$ es que cualquier registro que se disperse al cubo 0 basándose en $h_i$ se dispersará al cubo 0 o al cubo $M$ basándose en $h_{i+1}$; esto es necesario para que funcione la dispersión lineal.

Cuando colisiones posteriores llevan al desbordamiento de registros, se dividen cubos adicionales en orden *lineal*, 1, 2, 3,.... Si se producen bastantes desbordamientos, todos los cubos del fichero original, 0, 1,..., $M$ – 1, se habrán dividido, por lo que el fichero tendrá ahora $2M$ cubos en lugar de $M$, y todos los cubos utilizarán la función de dispersión $h_{i+1}$. Por tanto, los registros desbordados se redistribuyen en cubos normales, utilizando la función $h_{i+1}$ a través de una *división retardada* de sus cubos. No hay ningún directorio; sólo se necesita un valor $n$ (que inicialmente se establece a 0 y se incrementa en una unidad siempre que se produce una división) para determinar los cubos que se han dividido. Para recuperar un registro con el valor de clave de dispersión $K$, primero aplicamos la función $h_i$ a $K$; si $h_i(K) < n$, entonces aplicamos la función $h_{i+1}$ sobre $K$ porque el cubo ya está dividido. Inicialmente, $n = 0$, lo que indica que la función $h_i$ se aplica a todos los cubos; $n$ aumenta linealmente a medida que se dividen los cubos.

Cuando $n = M$ después de haberse incrementado, significa que todos los cubos originales se han dividido y que la función de dispersión $h_{i+1}$ se ha aplicado a todos los registros del fichero. En este punto, $n$ se reinicia a 0 (cero), y cualquier colisión nueva que provoque un desbordamiento conduce al uso de una nueva función de dispersión, $h_{i+2}(K) = K \bmod 4M$. En general, se utiliza una secuencia de funciones de dispersión $h_{i+j}(K) = K \bmod (2^j M)$, donde $j = 0, 1, 2,...$; se necesita una nueva función de dispersión $h_{i+j+1}$ siempre que todos los cubos 0, 1,..., $(2^j M)$-1 se hayan dividido y $n$ se haya reiniciado a 0. El Algoritmo 13.3 muestra la búsqueda de un registro con un valor de clave de dispersión de $K$.

La división puede controlarse monitorizando el factor de carga del fichero, en lugar de aplicar una división siempre que se produzca un desbordamiento. En general, el **factor de carga del fichero** $l$ se puede definir como $l = r/(bfr * N)$, donde $r$ es el número actual de registros del fichero, $bfr$ es el número máximo de registros que pueden encajar en un cubo, y $N$ es el número actual de cubos del fichero. Los cubos que se han dividido también se pueden recombinar si el factor de carga del fichero cae por debajo de un determinado umbral. Los bloques se combinan linealmente y $N$ se reduce en consecuencia. La carga del fichero se puede utilizar para lanzar tanto divisiones como combinaciones; de este modo, la carga del fichero puede mantenerse dentro del rango deseado. Las divisiones se pueden lanzar cuando la carga excede un determinado umbral (por ejemplo, 0,9) y las combinaciones se pueden lanzar cuando la carga cae por debajo de otro umbral (por ejemplo, 0,7).

**Algoritmo 13.3. Procedimiento de búsqueda para la dispersión lineal.**

```
if n = 0
   then m ← h_j (K) (* m es el valor de dispersión del registro con la clave de dispersión K *)
   else begin
        m ← h_j (K);
        if m < n then m ← h_{j+1} (K )
        end;
buscar el cubo cuyo valor de dispersión es m (y su desbordamiento, si lo hay);
```

# 13.9 Otras organizaciones principales de ficheros

## 13.9.1 Ficheros de registros mezclados

Las organizaciones de ficheros que hemos estudiado hasta ahora asumen que todos los registros de un fichero en particular son del mismo tipo de registro. Los registros pueden ser de EMPLEADOs, PROYECTOs, ESTUDIANTEs o DEPARTAMENTOs, pero cada fichero contiene registros de un solo tipo. En la mayoría de las aplicaciones de bases de datos nos encontramos con situaciones en las que varios tipos de entidades se

interrelacionan de distintas formas, como vimos en el Capítulo 3. Las relaciones entre registros de varios ficheros se pueden representar mediante **campos de conexión**.[11] Por ejemplo, un registro ESTUDIANTE puede tener un campo de conexión DptoEspecialidad cuyo valor proporciona el nombre del DEPARTAMENTO en el que el estudiante se está especializando. Este campo DptoEspecialidad se *refiere* a una entidad DEPARTAMENTO, que debe estar representada por un registro propio en el fichero DEPARTAMENTO. Si queremos recuperar los valores de campo de dos registros relacionados, primero debemos recuperar uno de los registros. Después, podemos utilizar el valor de su campo de conexión para recuperar el registro relacionado del otro fichero. Por tanto, las relaciones se implementan mediante **referencias de campo lógicas** entre los registros de ficheros distintos.

Las organizaciones de ficheros en los DBMSs de objetos, así como los sistemas heredados (por ejemplo, los DBMSs jerárquicos y de red), a menudo implementan las relaciones entre los registros como **relaciones físicas** materializadas por la contigüidad física (o agrupamiento) de los registros relacionados o por los punteros físicos. Estas organizaciones de fichero normalmente asignan un **área** del disco para albergar registros de más de un tipo, para que esos registros de tipos distintos puedan **agruparse físicamente** en el disco. Si se espera utilizar con frecuencia una relación particular, la implementación de una relación física puede aumentar la eficacia del sistema al recuperar registros relacionados. Por ejemplo, si es frecuente la consulta para recuperar un registro DEPARTAMENTO y los registros de todas las especialidades de los ESTUDIANTEs de ese departamento, sería deseable colocar cada registro de DEPARTAMENTO y su grupo de registros de ESTUDIANTE contiguamente en disco en un fichero mixto. En los DBMSs de objetos se utiliza el concepto de **agrupamiento físico** de tipos de objetos para almacenar juntos los objetos relacionados en un fichero mixto.

Para distinguir los registros de un fichero mixto, cada registro tiene (además de los valores de sus campos) un campo con el **tipo de registro**. Normalmente es el primer campo de cada registro y lo utiliza el software del sistema para determinar el tipo del registro que está a punto de procesar. Con la información del catálogo, el DBMS puede determinar los campos de ese tipo de registro y sus tamaños, a fin de interpretar los valores del registro.

### 13.9.2 Árboles B y otras estructuras de datos como organización primaria

Se pueden utilizar otras estructuras de datos como organizaciones primarias de ficheros. Por ejemplo, si tanto el tamaño del registro como el número de registros de un fichero son pequeños, algunos DBMSs ofrecen la opción de una estructura de árbol B como organización primaria de fichero. En la Sección 14.3.1 describiremos los árboles B cuando expliquemos el uso de la estructura de datos de árbol B para la indexación. En general, cualquier estructura de datos que pueda adaptarse a las características de los dispositivos de disco puede utilizarse como organización primaria de fichero para la ubicación de registros en el disco.

## 13.10 Paralelismo del acceso al disco mediante la tecnología RAID

Con el crecimiento exponencial del rendimiento y la capacidad de los dispositivos semiconductores y de las memorias, aparecen continuamente microprocesadores más rápidos con memorias primarias más y más grandes. Para igualar este crecimiento, es natural esperar que la tecnología del almacenamiento secundario también avance a la par que la tecnología de los procesadores en cuanto a rendimiento y fiabilidad.

Una importante ventaja de la tecnología de almacenamiento secundario está representada por el desarrollo de **RAID**, que originariamente significaba **Matriz redundante de discos baratos** (*Redundant Arrays of*

[11] El concepto de *foreign keys* o claves externas en el modelo relacional (Capítulo 5) y las referencias entre objetos en los modelos orientados a objetos (Capítulo 20) son ejemplos de campos de conexión.

*Inexpensive Disks*). Últimamente, la *I* de RAID ha cambiado su significado por el de "independientes". La idea tras RAID recibió un respaldo muy positivo por parte de la industria y se ha convertido en un conjunto elaborado de arquitecturas RAID alternativas (RAID de niveles 0 a 6). Destacaremos las principales características de esta tecnología.

El objetivo principal de RAID es igualar las amplias diferencias de rendimiento y velocidad entre los discos y la memoria y los microprocesadores.[12] Mientras que la capacidad de la RAM se cuadriplica cada dos a tres años, los *tiempos de acceso* al disco se mejoran menos de un 10% por año, y las *velocidades de transferencia* al disco se mejoran apenas un 20% al año. Las *capacidades* de los discos están mejorando por encima del 50% anual, pero las mejoras en la velocidad y el tiempo de acceso son de una magnitud mucho más pequeña. La Tabla 13.3 compara la tecnología de los discos con una diferencia de diez años, así como las tasas de mejora.

Existe una segunda disparidad cualitativa entre la capacidad de los microprocesadores especiales que satisfacen las necesidades de las aplicaciones que deben procesar vídeo, audio, imagen y datos especiales (consulte los Capítulos 24 y 29 si desea más información sobre estas aplicaciones), con la correspondiente carencia de un acceso rápido a los conjuntos más grandes de datos compartidos.

La solución natural es una matriz grande de pequeños discos independientes que actúan como un solo disco lógico de alto rendimiento. Se utiliza un concepto denominado **segmentación (*striping*) de datos**, que utiliza el *paralelismo* para mejorar el rendimiento del disco. La segmentación de datos distribuye los datos de forma transparente por varios discos para hacerlos aparecer como un solo disco más grande y rápido. La Figura 13.12 muestra un fichero segmentado o distribuido por cuatro discos. La segmentación mejora el rendimiento de E/S global al permitir varias operaciones de E/S en paralelo, lo que proporciona unas altas velocidades de transferencia global. La segmentación de los datos también acomete el equilibrado de la carga entre los discos. Además, al almacenar información redundante en los discos mediante la paridad o algún otro código de corrección de errores se puede mejorar la fiabilidad. En las Secciones 13.3.1 y 13.3.2, explicamos cómo RAID logra los importantes objetivos de mejorar la fiabilidad y el rendimiento. La Sección 13.3.3 explica las organizaciones RAID.

**Tabla 13.3.** Tendencias en la tecnología del disco.

| | Valores en 1993* | Tasa histórica de mejora anual (%)* | Valores en 2003** |
|---|---|---|---|
| Densidad superficial | 50–150 Mb/pulgada cuadrada | 27 | 36 Gb/pulgada cuadrada |
| Densidad lineal | 40.000–60.000 bits/pulgada | 13 | 570 Kb/pulgada |
| Densidad entre pistas | 1.500–3.000 pistas/pulgada | 10 | 64,000 pistas/pulgada |
| Capacidad (factor de forma de 3,5 pulgadas) | 100–2.000 MB | 27 | 146 GB |
| Velocidad de transferencia | 3–4 MB/s | 22 | 43–78 MB/seg |
| Tiempo de búsqueda | 7–20 ms | 8 | 3,5–6 ms |

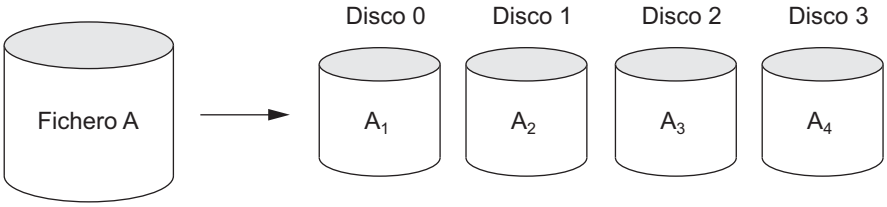
**Fuente*: De Chen, Lee, Gibson, Katz y Patterson (1994), *ACM Computing Surveys,* Vol. 26, No. 2 (junio de 1994).

Reproducido con permiso.

***Fuente*: Unidades de disco duro IBM Ultrastar 36XP y 18ZX.

[12] Gordon Bell predijo que sería aproximadamente el 40 por ciento por año entre 1974 y 1984, y hoy se supone que excede el 50 por ciento anual.

**Figura 13.12.** Segmentación de datos. El Fichero A está dividido en cuatro discos.




## 13.10.1 Mejora de la fiabilidad con RAID

Para un array de *n* discos, la probabilidad de un fallo es *n* veces superior que para un solo disco. Por tanto, si asumimos que el MTTF (tiempo medio entre fallos) de una unidad de disco es de 200.000 horas o aproximadamente 22,8 años (los tiempos normales llegan al millón de horas), el de un banco de 100 unidades de disco es sólo de 2.000 horas u 83,3 días. Guardar una sola copia de los datos en una matriz semejante de discos provocará una significativa pérdida de fiabilidad. Una solución obvia es emplear la redundancia de datos a fin de poder tolerar los fallos de disco. Los inconvenientes son muchos: operaciones de E/S adicionales para escribir, cálculos extra para mantener la redundancia y recuperarse de los errores, y capacidad de disco adicional para almacenar la información redundante.

Una técnica para introducir la redundancia se denomina **espejo** o **reflejo** (*mirroring*) o ***shadowing***. Los datos se escriben redundantemente en dos discos físicos idénticos que se tratan como un solo disco lógico. Cuando se leen los datos, se pueden recuperar del disco con retardos de cola, búsqueda y rotación más cortos. Si falla un disco, se utiliza el otro hasta que el primero es reparado. Supongamos que el tiempo medio de reparación es de 24 horas, entonces el tiempo medio de que se pierdan datos en un sistema de disco en espejo que utiliza 100 discos con un MTTF de 200.000 horas cada uno es de $(200.000)^2/(2 * 24) = 8{,}33 * 10^8$ horas, que son 95.028 años.[13] Los discos en espejo también duplican la velocidad a la que se manipulan las solicitudes de lectura, puesto que la lectura puede ir a cualquier disco. Sin embargo, la velocidad de transferencia de cada lectura sigue siendo la misma que para un solo disco.

Otra solución al problema de la fiabilidad es almacenar información extra que normalmente no es necesaria, pero que puede utilizarse para reconstruir la información perdida en caso de un fallo en el disco. La incorporación de la redundancia debe tener en consideración dos problemas: seleccionar una técnica para calcular la información redundante, y seleccionar un método de distribución de la información redundante por la matriz de discos. El primer problema se resuelve utilizando códigos de corrección de errores que implican el uso de bits de paridad, o códigos especializados como los códigos Hamming. Si optamos por el esquema de paridad, podemos tener un disco redundante con la suma de todos los datos de los otros discos. Cuando un disco falla, la información perdida puede reconstruirse con un proceso parecido a la substracción.

Para resolver el segundo problema, los dos métodos principales consisten en almacenar información redundante en un pequeño número de discos o distribuirla uniformemente por todos los discos. Este último método ofrece un mejor equilibrado de la carga. Los diferentes niveles de RAID eligen una combinación de estas opciones para implementar la redundancia y mejorar la fiabilidad.

## 13.10.2 Mejora del rendimiento con RAID

Las matrices de discos emplean la técnica de la segmentación de los datos para lograr unas velocidades de transferencia más altas. Los datos sólo se pueden leer o escribir de bloque en bloque, por lo que una transferencia típica contiene de 512 a 8.192 bytes. La segmentación del disco puede aplicarse con una granularidad

[13] Las fórmulas para calcular el MTTF aparecen en Chen y otros (1994).

más final dividiendo un byte de datos en bits y diseminando los bits por diferentes discos. De este modo, la **segmentación de datos a nivel de bit** consiste en dividir un byte de datos y escribir el bit *j* en el disco número *j*. Con bytes de 8 bits, podemos considerar que ocho discos físicos son un disco lógico, con una velocidad de transferencia de datos aumentada ocho veces. Cada disco participa en cada solicitud de E/S y la cantidad total de datos leída por solicitud es casi ocho veces superior. La segmentación a nivel de bit puede generalizarse a un número de discos que sea múltiplo o factor de ocho. Así, en una matriz de cuatro discos, el bit *n* va al disco (*n* mod 4).

La granularidad de la intercalación o interpolación de datos puede ser superior a un bit; por ejemplo, los bloques de un fichero se pueden segmentar por varios discos, dando lugar a la **segmentación a nivel de bloque**. La Figura 13.12 muestra la segmentación de datos a nivel de bloque asumiendo que el fichero de datos está contenido en cuatro bloques. Con esta segmentación, varias solicitudes independientes que acceden a bloques sencillos (solicitudes pequeñas) pueden ser servidas en paralelo por discos separados, reduciéndose de este modo el tiempo de espera en cola de las solicitudes de E/S. Las solicitudes que acceden a varios bloques (solicitudes grandes) se pueden hacer en paralelo, lo que reduce su tiempo de respuesta. En general, cuantos más discos haya en una matriz, mayor es la posibilidad de mejorar el rendimiento. Sin embargo, y asumiendo fallos independientes, una matriz de 100 discos tiene una fiabilidad colectiva que es 1/100 de la fiabilidad de un solo disco. Por tanto, son necesarios la redundancia mediante códigos de corrección de errores y discos reflejados (en espejo) para mejorar la fiabilidad, junto con un rendimiento más alto.

### 13.10.3 Organizaciones y niveles de RAID

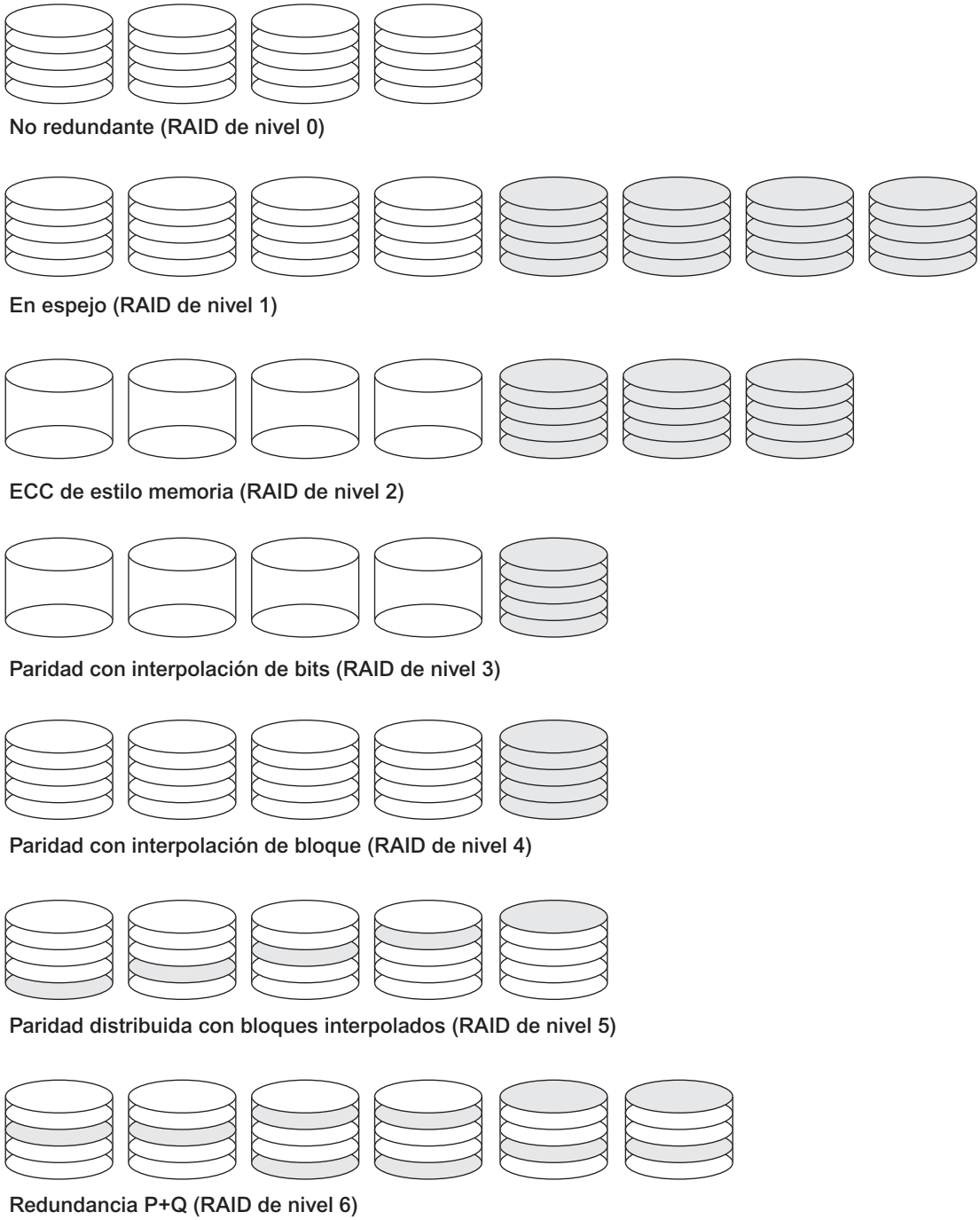
Las diferentes organizaciones de RAID se definieron basándose en diferentes combinaciones de los dos factores de granularidad del interpolado de datos (segmentación) y del patrón utilizado para calcular la información redundante. En la propuesta inicial aparecían 5 niveles de RAID (del 1 al 5), y más tarde se añadieron los niveles 0 y 6.

El nivel 0 de RAID utiliza la segmentación de datos, no tiene datos redundantes y, por tanto, ofrece un mejor rendimiento de escritura porque las actualizaciones no tienen que duplicarse. Sin embargo, su rendimiento en la lectura no es tan bueno como en el nivel 1 de RAID, que utiliza discos reflejados. En este último, es posible mejorar el rendimiento planificando una solicitud de lectura del disco con un retardo de búsqueda y rotacional más corto de lo esperado. El nivel 2 de RAID utiliza una redundancia al estilo de la memoria utilizando códigos Hamming, que contienen bits de paridad para distintos subconjuntos superpuestos de componentes. Por tanto, en una versión particular de este nivel, tres discos redundantes serían suficientes para cuatro discos originales, mientras que con los discos en espejo (como en el nivel 1) se necesitarían cuatro discos. El nivel 2 incluye la detección de errores y la corrección, aunque normalmente no se requiere la detección porque los discos dañados quedan identificados.

El nivel 3 de RAID utiliza un disco de paridad sencilla que cuenta con el controlador de disco para determinar el disco que ha fallado. Los niveles 4 y 5 utilizan la segmentación de datos a nivel de bloque; el nivel 5 distribuye los datos y la información de paridad por todos los discos. Por último, el nivel 6 de RAID aplica lo que se conoce como esquema de redundancia $P + Q$ que utiliza los códigos Reed-Soloman para protegerse contra el fallo de hasta dos discos, utilizando dos discos redundantes. En la Figura 13.13 se ilustran los siete niveles de RAID (0 a 6).

Con el nivel 1 de RAID la reconstrucción es sencilla en caso de que falle un disco. Los otros niveles necesitan leer varios discos para poder reconstruir un disco dañado. El nivel 1 se utiliza para aplicaciones críticas, como el almacenamiento de los registros de transacciones. Los niveles 3 y 5 son los preferidos para el almacenamiento de grandes volúmenes, mientras que el nivel 3 proporciona unas velocidades de transferencia más altas. Los niveles RAID más utilizados actualmente son el 0 (con segmentación), el 1 (con espejo) y el 5 con una unidad adicional para la paridad. Los diseñadores de una configuración RAID para una aplicación mixta dada deben confrontar muchas decisiones de diseño, como el nivel de RAID, el número de discos, la elección

**Figura 13.13.** Niveles de RAID. Información extraída de Chen, Lee, Gibson, Katz y Patterson (1994), ACM Computing Survey, Vol. 26, No. 2 (junio de 1994). Reproducido con permiso.




de los esquemas de paridad, y el agrupamiento de discos para la segmentación a nivel de bloque. Se han realizado estudios detallados del rendimiento sobre pequeñas cantidades de lecturas y escrituras (en referencia a solicitudes de E/S para una unidad de segmentación) y grandes cantidades de lecturas y escrituras

(en referencia a solicitudes de E/S para una unidad de segmentación de cada disco en un grupo de corrección de errores).

# 13.11 Nuevos sistemas de almacenamiento

En esta sección describimos dos desarrollos recientes en los sistemas de almacenamiento que se han convertido en parte integral de la mayoría de arquitecturas de los sistemas de información empresariales.

## 13.11.1 Redes de área de almacenamiento

Con el rápido crecimiento del comercio electrónico, los sistemas de Planificación de recursos empresariales (ERP, *Enterprise Resource Planning*) que integran datos de una aplicación a través de varias empresas, y almacenes de datos que almacenan información agregada histórica (consulte el Capítulo 27), la demanda de almacenamiento ha crecido sustancialmente. Las empresas actuales orientadas a Internet se han visto en la necesidad de moverse de un centro de datos fijo y estático orientado a las operaciones, a una infraestructura más flexible y dinámica para sus requisitos de procesamiento de información. El coste total de administración de todos los datos crece tan rápidamente que en muchos casos el coste de administrar el almacenamiento conectado a un servidor excede el coste del propio servidor. Además, el coste de adquisición del almacenamiento sólo es una pequeña fracción (normalmente, sólo del 10 al 15% del coste global de administración del almacenamiento). Muchos usuarios de sistemas RAID no pueden utilizar eficazmente la capacidad porque tienen que estar conectados de un modo fijo a uno o más servidores. Por consiguiente, las empresas grandes se están moviendo a un concepto denominado **Redes de área de almacenamiento (SAN**, *Storage Area Networks*). En una SAN, los periféricos de almacenamiento online están configurados como nodos en una red de alta velocidad y se pueden conectar y desconectar de los servidores con mucha flexibilidad. Varias empresas han surgido como proveedores SAN y proporcionan sus propias topologías patentadas. Estas redes permiten que los sistemas de almacenamiento se coloquen a mayor distancia de los servidores y proporcionan diferentes opciones de rendimiento y conectividad. Las aplicaciones de administración del almacenamiento existentes se pueden portar a configuraciones de SAN mediante redes de canal de fibra (Fiber Channel) que encapsulan el protocolo SCSI heredado. En consecuencia, los dispositivos conectados a la SAN aparecen como dispositivos SCSI.

Las alternativas arquitectónicas actuales para SAN incluyen lo siguiente: conexiones punto a punto entre servidores y sistemas de almacenamiento a través de un canal de fibra, el uso de un *switch* de canal de fibra para conectar varios sistemas RAID, librerías de cinta, etcétera a los servidores, y el uso de *hubs* y *switches* de canal de fibra para conectar los servidores y los sistemas de almacenamiento en diferentes configuraciones. Las empresas pueden moverse lentamente de unas topologías más sencillas a otras más complejas añadiendo servidores y dispositivos de almacenamiento según lo necesiten. No ofreceremos aquí más detalles porque varían mucho de unos proveedores de SAN a otros. Las principales ventajas exigibles son las siguientes:

- Conectividad "varios a varios" flexible entre los servidores y los dispositivos de almacenamiento utilizando *hubs* y *switches* de canal de fibra.
- Hasta 10 kilómetros de separación entre un servidor y un sistema de almacenamiento utilizando los cables de fibra óptica apropiados.
- Buenas capacidades de aislamiento que permitan añadir sin complicaciones nuevos periféricos y servidores.

Las SANs están creciendo muy rápidamente, pero todavía se enfrentan a muchos problemas, como la combinación de opciones de almacenamiento de diferentes fabricantes y el desarrollo de estándares para el software y el hardware de administración del almacenamiento. La mayoría de las principales empresas están evaluando SAN como una opción viable para el almacenamiento de bases de datos.

### 13.11.2 Almacenamiento conectado a la red

Con el fenomenal crecimiento de los datos digitales, en particular los generados por las aplicaciones multimedia y otras aplicaciones empresariales, la necesidad de soluciones de almacenamiento de alto rendimiento y bajo coste se ha convertido en algo extremadamente importante. Los dispositivos **NAS** (**Almacenamiento conectado a la red**, *Network-Attached Storage*) se encuentran entre los últimos dispositivos de almacenamiento que se están utilizando con este propósito. Estos dispositivos son, de hecho, servidores que no proporcionan ninguno de los servicios de servidor comunes, sino que simplemente permiten la adición de almacenamiento para compartir ficheros. Los dispositivos NAS permiten añadir vastas cantidades de espacio de almacenamiento en disco duro a una red y hacen que ese espacio esté disponible para varios servidores sin tener que pararlos para realizar el mantenimiento y las actualizaciones. Los dispositivos NAS pueden residir en cualquier parte de una red de área local (LAN) y pueden combinarse en diferentes configuraciones. Un único dispositivo hardware, a menudo denominado **caja NAS** o **cabecera NAS**, actúa como interfaz entre el sistema NAS y los clientes de la red. Estos dispositivos NAS no requieren monitor, teclado o ratón. Es posible conectar una o más unidades de disco o cinta a varios sistemas NAS para aumentar la capacidad total. Los clientes se conectan a la cabecera NAS, en lugar de hacerlo a los dispositivos de almacenamiento individuales. Un NAS puede almacenar cualquier dato que aparezca en forma de ficheros, como buzones de correo electrónico, contenido web, copias de seguridad remotas del sistema, etcétera. En este sentido, los dispositivos NAS se están implantando como un sustituto de los servidores de ficheros tradicionales.

Los sistemas NAS se esfuerzan por gozar de un funcionamiento fiable y una administración sencilla. Incluyen características integradas como la autenticación segura o el envío automático de alertas por email en caso de un error en el dispositivo. Los dispositivos NAS se están ofreciendo con un alto grado de escalabilidad, fiabilidad, flexibilidad y rendimiento. Normalmente soportan los niveles 0, 1 y 5 de RAID. Las SANs tradicionales difieren de los NASs en varias cosas. En concreto, las SANs a menudo utilizan Fiber Channel en lugar de Ethernet, y una SAN a menudo incorpora varios dispositivos de red o *puntos finales* en una LAN autocontenida o *privada*, mientras que un NAS depende de dispositivos individuales conectados directamente a una LAN pública existente. Mientras que los servidores de ficheros Windows, UNIX y NetWare demandan cada uno el soporte de un protocolo específico en el lado del cliente, los sistemas NAS exigen una independencia mayor del sistema operativo por parte de los clientes.

## 13.12 Resumen

Empezamos este capítulo explicando las características de las jerarquías de la memoria y después nos centramos en los dispositivos de almacenamiento secundario. En particular, nos centramos en los discos magnéticos porque son los que más se utilizan para almacenar los ficheros de bases de datos online.

Los datos se almacenan en los bloques de los discos; el acceso a un bloque de un disco es costoso debido al tiempo de búsqueda, el retardo rotacional y el tiempo de transferencia del bloque. Para reducir el tiempo medio de acceso a un bloque, se puede utilizar el doble búfer para acceder a bloques de disco consecutivos. En el Apéndice B explicamos otros parámetros relacionados con un disco. Presentamos las diferentes formas de almacenar los registros de un fichero en un disco. Los registros de un fichero se agrupan en bloques del disco y pueden ser de longitud fija o variable, extendida o no extendida, y del mismo tipo de registro o de una mezcla de tipos. Explicamos la cabecera de un fichero, que describe el formato del registro y alberga las direcciones de disco de los bloques del fichero. El software del sistema utiliza la información de esta cabecera para acceder a los registros del fichero.

Después, presentamos un conjunto de comandos típicos para acceder a los registros individuales de un fichero, y explicamos el concepto de registro actual de un fichero. También explicamos cómo las condiciones de búsqueda de registros complejas se pueden transformar en condiciones de búsqueda sencillas que se utilizan para localizar registros en un fichero.

A continuación explicamos las tres principales organizaciones de un fichero: desordenado, ordenado y disperso. Los ficheros desordenados requieren una búsqueda lineal para localizar registros, pero la inserción de un registro es muy sencilla. Explicamos el problema del borrado y el uso de los marcadores de borrado.

Los ficheros ordenados reducen el tiempo necesario para leer registros en el orden estipulado por el campo de ordenación. El tiempo necesario para realizar la búsqueda de un registro arbitrario, dado el valor de su campo clave de ordenación, también se reduce si se utiliza una búsqueda binaria. Sin embargo, el mantenimiento de un orden en los registros hace que la inserción sea muy costosa; por eso explicamos la técnica de utilizar un fichero de desbordamiento desordenado para reducir el coste que supone insertar un registro. Los registros de desbordamiento se mezclan periódicamente con el fichero maestro durante la reorganización del fichero.

La dispersión proporciona un acceso muy rápido a un registro arbitrario de un fichero, dado el valor de su clave de dispersión. El método más adecuado para la dispersión externa es la técnica de los cubos, en la que a cada cubo le corresponden uno o más bloques contiguos. Las colisiones provocadas por un desbordamiento del cubo se manipulan con el encadenamiento. El acceso a cualquier campo no disperso es lento. Explicamos dos técnicas de dispersión para los ficheros cuya cantidad de registros crece o disminuye dinámicamente: dispersión extensible y lineal.

Explicamos brevemente otras posibilidades para la reorganización primaria de un fichero, como los árboles B y los ficheros de registros mixtos, que implementan relaciones físicas entre registros de diferentes tipos como parte de la estructura de almacenamiento. Por último, repasamos los avances más recientes representados por la tecnología RAID.

## Preguntas de repaso

**13.1.** ¿Cuál es la diferencia entre almacenamiento primario y secundario?

**13.2.** ¿Por qué se utilizan discos, y no cintas, para almacenar los ficheros de bases de datos online?

**13.3.** Defina los siguientes términos: *disco, paquete de discos, pista, bloque, cilindro, sector, hueco entre bloques, cabeza de lectura/escritura.*

**13.4.** Explique el proceso de inicialización de un disco.

**13.5.** Explique el mecanismo que se utiliza para leer los datos o escribir datos en el disco.

**13.6.** ¿Cuáles son los componentes de la dirección de un bloque del disco?

**13.7.** ¿Por qué resulta costoso acceder a un bloque de un disco? Explique los distintos tiempos implicados en el acceso a un bloque del disco.

**13.8.** ¿Cómo mejora el tiempo de acceso a un bloque utilizando el doble búfer?

**13.9.** ¿Cuáles son las razones que llevan a tener registros de longitud variable? ¿Qué tipos de caracteres separadores se necesitan para cada uno?

**13.10.** Explique las técnicas de asignación de bloques de fichero en el disco.

**13.11.** ¿Cuál es la diferencia entre una organización de fichero y un método de acceso?

**13.12.** ¿Cuál es la diferencia entre un fichero estático y uno dinámico?

**13.13.** ¿Cuáles son las operaciones típicas de un registro por vez para acceder a un fichero? ¿Cuáles de ellas dependen del registro actual del fichero?

**13.14.** Explique las técnicas para borrar un registro.

**13.15.** Explique las ventajas y los inconvenientes de utilizar (a) un fichero desordenado, (b) un fichero ordenado, y (c) un fichero disperso estático con cubos y encadenamiento. ¿Qué operaciones se pueden llevar a cabo eficazmente en cada una de estas organizaciones, y qué operaciones resultan costosas?

**13.16.** Explique las técnicas para permitir que un fichero de dispersión crezca o disminuya dinámicamente. ¿Cuáles son las ventajas y los inconvenientes de cada una de ellas?

**13.17.** ¿Para qué se utilizan los ficheros mixtos? ¿Cuáles son los otros tipos de organizaciones primarias de fichero?

**13.18.** Describa la incompatibilidad entre el procesador y las tecnologías de disco.

**13.19.** ¿Cuáles son los principales objetivos de la tecnología RAID? ¿Cómo se consiguen?

**13.20.** ¿Cómo ayuda un disco reflejado a mejorar la fiabilidad? Ofrezca un ejemplo cuantitativo.

**13.21.** ¿Qué técnicas se utilizan para mejorar el rendimiento de los discos en RAID?

**13.22.** ¿Qué caracteriza los niveles en la organización RAID?

## Ejercicios

**13.23.** Piense en un disco con las siguientes características (no son parámetros de una unidad de disco específica): tamaño de bloque $B = 512$ bytes; tamaño del hueco entre bloques $G = 128$ bytes; número de bloques por pista = 20; número de pistas por superficie = 400. Un paquete de discos consta de 15 discos de doble cara.

- **a.** ¿Cuál es la capacidad total de una pista, y cuál es su capacidad útil (excluyendo los huecos entre bloques)?
- **b.** ¿Cuántos cilindros tiene?
- **c.** ¿Cuál es la capacidad total y la capacidad útil de un cilindro?
- **d.** ¿Cuál es la capacidad total y la capacidad útil de un paquete de discos?
- **e.** Suponga que la unidad de disco gira el paquete de discos a una velocidad de 2.400 rpm (revoluciones por minuto); ¿cuál es la velocidad de transferencia (*tr*) en bytes/mseg y el tiempo de transferencia de un bloque (*btt*) en mseg? ¿Cuál es el retardo rotacional medio (*rd*) en mseg? ¿Cuál es la velocidad de transferencia en masa? (Consulte el Apéndice B.)
- **f.** Suponga que el tiempo de búsqueda medio es de 30 mseg. ¿Cuánto tiempo tarda (por término medio) en mseg en localizar y transferir un solo bloque, dada la dirección de ese bloque?
- **g.** Calcule el tiempo medio que tardaría en transferir 20 bloques aleatorios, y compárelo con el tiempo que tardaría en transferir 20 bloques consecutivos utilizando el doble búfer para ahorrar tiempo de búsqueda y retardo rotacional.

**13.24.** Un fichero tiene $r = 20.000$ registros ESTUDIANTE de *longitud fija*. Cada registro tiene los siguientes campos: Nombre (30 bytes), Dni (9 bytes), Direcc (40 bytes), Tlf (9 bytes), FechaNac (8bytes), Sexo (1 byte), CodDptoEspecialidad (4 bytes), CodDptoEspecSec (4 bytes), CodClase (4bytes, entero) y ProgramaGrado (3 bytes). Se utiliza un byte adicional como marcador de borrado. El fichero está almacenado en un disco cuyos parámetros son los facilitados en el Ejercicio 13.23.

- **a.** Calcule el tamaño de registro $R$ en bytes.
- **b.** Calcule el factor de bloqueo *bfr* y el número de bloques del fichero $b$, asumiendo una organización no extendida.
- **c.** Calcule el tiempo medio que se tardaría en encontrar un registro realizando una búsqueda lineal sobre el fichero si (i) los bloques del fichero están almacenados contiguamente y se utiliza el doble búfer; (ii) los bloques del fichero no están almacenados contiguamente.
- **d.** Asuma que el fichero está ordenado por Dni; con una búsqueda binaria, calcule el tiempo que se tardaría en encontrar un registro dado el valor de su Dni.

**13.25.** Suponga que sólo el 80% de los registros de ESTUDIANTE del Ejercicio 13.24 tienen un valor para Tlf, el 85% para CodDptoEspecialidad, el 15% para CodDptoEspecSec y el 90% para ProgramaGrado; y suponga que utilizamos un fichero de registros de longitud variable. Cada registro tiene 1 byte como *tipo de campo* por cada campo del registro, más 1 byte del marcador de borrado y 1 byte como marcador de fin de registro. Suponga además que utilizamos una organización de

registros extendida, donde cada bloque tiene un puntero de 5 bytes apuntando al siguiente bloque (este espacio no se utiliza para el almacenamiento del registro).

**a.** Calcule la longitud media del registro *R* en bytes.

**b.** Calcule el número de bloques necesarios para el fichero.

**13.26.** Suponga que una unidad de disco tiene los siguientes parámetros: tiempo de búsqueda $s = 20$ mseg; retardo rotacional $rd = 10$ mseg; tiempo de transferencia de un bloque $btt = 1$ mseg; tamaño de bloque $B = 2.400$ bytes; tamaño de hueco entre bloques $G = 600$ bytes. Un fichero EMPLEADO tiene los siguientes campos: Dni, 9 bytes; Nombre, 20 bytes; Apellidos, 20 bytes; Inicial, 1 byte; FechaNac, 10 bytes; Direcc, 35 bytes; Tlf, 12 bytes; DniSupervisor, 9 bytes; Departamento, 4 bytes; CodTrabajo, 4 bytes; *marcador de borrado,* 1 byte. El fichero EMPLEADO tiene $r = 30.000$ registros, formato de longitud fija, y bloqueo no extendido. Escriba las fórmulas apropiadas *y* calcule los siguientes valores para este fichero EMPLEADO:

**a.** Tamaño de registro *R* (incluyendo el marcador de borrado), factor de bloqueo *bfr*, y número de bloques de disco *b*.

**b.** Calcule el espacio desperdiciado en cada bloque de disco debido a la organización no extendida.

**c.** Calcule la velocidad de transferencia *tr* y la velocidad de transferencia en masa *btr* de esta unidad de disco (consulte el Apéndice B si desea conocer las definiciones de *tr* y *btr*).

**d.** Calcule la media de *accesos a bloques* necesarios para buscar un registro arbitrario en el fichero, utilizando una búsqueda lineal.

**e.** Calcule en mseg el *tiempo medio* necesario para buscar un registro arbitrario en el fichero, utilizando la búsqueda lineal, si los bloques del fichero están almacenados en bloques de disco consecutivos y utilizando el doble búfer.

**f.** Calcule en mseg el *tiempo medio* necesario para buscar un registro arbitrario en el fichero, utilizando la búsqueda lineal, si los bloques del fichero no están almacenados en bloques de disco consecutivos.

**g.** Asuma que los registros están ordenados mediante algún campo clave. Calcule el *promedio de accesos a bloques* y el *tiempo medio* necesario para buscar un registro arbitrario en el disco utilizando una búsqueda binaria.

**13.27.** Un fichero REPUESTOS con NumRep como clave de dispersión incluye registros con los siguientes valores de NumRep: 2369, 3760, 4692, 4871, 5659, 1821, 1074, 7115, 1620, 2428, 3943, 4750, 6975, 4981, 9208. El fichero utiliza ocho cubos, numerados de 0 a 7. Cada cubo es un bloque de disco y almacena dos registros. Cargue estos registros en el fichero en el orden dado, utilizando la función de dispersión $h(K) = K \bmod 8$. Calcule la cantidad media de accesos a bloque para una recuperación aleatoria sobre NumRep.

**13.28.** Cargue los registros del Ejercicio 13.27 en ficheros de dispersión expansibles basándose en la dispersión extensible. Muestre la estructura del directorio en cada paso, así como las profundidades global y local. Utilice la función de dispersión $h(K) = K \bmod 128$.

**13.29.** Cargue los registros del Ejercicio 13.27 en un fichero de dispersión expansible, utilizando la dispersión lineal. Empiece con un solo bloque de disco, utilizando la función de dispersión $h_0 = K \bmod 2^0$, y muestre cómo crece el fichero y cómo las funciones de dispersión cambian a medida que se insertan registros. Asuma que los bloques se dividen siempre que se produce un desbordamiento, y muestre el valor de *n* en cada etapa.

**13.30.** Compare los comandos de fichero de la Sección 13.6 con los disponibles en un método de acceso a ficheros con el que esté familiarizado.

**13.31.** Suponga que tenemos un fichero desordenado de registros de longitud fija que utiliza una organización de registros no extendida. Diseñe unos algoritmos para insertar, borrar y modificar un registro del fichero. Especifique las suposiciones que asuma.

**13.32.** Suponga que tenemos un fichero ordenado de registros de longitud fija y un fichero de desbordamiento desordenado para manipular la inserción. Los dos ficheros utilizan registros no extendidos. Diseñe unos algoritmos para insertar, borrar y modificar un registro del fichero y para reorganizar el fichero. Especifique las suposiciones que asuma.

**13.33.** ¿Puede pensar en otras técnicas diferentes a usar un fichero de desbordamiento desordenado que puedan utilizarse para realizar inserciones en un fichero ordenado de un modo más eficaz?

**13.34.** Suponga que tenemos un fichero de dispersión de registros de longitud fija, y suponga también que el desbordamiento lo manipulamos con el encadenamiento. Diseñe unos algoritmos para insertar, borrar y modificar un registro del fichero. Especifique las suposiciones que asuma.

**13.35.** Escriba el código de un programa para acceder a los campos individuales de los registros bajo cada una de las siguientes circunstancias. En cada caso, enuncie las suposiciones que está haciendo en relación a punteros, caracteres de separación, etcétera. Determine el tipo de información necesaria en la cabecera del fichero para que su código sea general en cada caso.

- **a.** Registros de longitud fija con bloqueo no extendido.
- **b.** Registros de longitud fija con bloqueo extendido.
- **c.** Registros de longitud variable con campos de longitud variable y bloqueo extendido.
- **d.** Registros de longitud variable con repetición de grupos y bloqueo extendido.
- **e.** Registros de longitud variable con campos opcionales y bloqueo extendido.
- **f.** Registros de longitud variable que permite los tres casos indicados en las partes c, d, y e.

## Bibliografía seleccionada

Wiederhold (1983) tiene una explicación y un análisis detallados de los dispositivos de almacenamiento secundario y de las organizaciones de ficheros. Los discos ópticos se describen en Berg y Roth (1989) y se analizan en Ford y Christodoulakis (1991). La memoria flash se explica en Dippert y Levy (1993). Ruemmler y Wilkes (1994) presenta un estudio de la tecnología de disco magnético. La mayoría de los libros sobre bases de datos incluyen explicaciones del material aquí presentado. Casi todos los libros sobre estructuras de datos, incluyendo Knuth (1973), explican la dispersión estática más en detalle; Knuth tiene una explicación completa de las funciones de dispersión y de las técnicas de resolución de colisiones, así como de la comparación de su rendimiento. Entre los libros sobre estructuras de datos citamos Claybrook (1983), Smith y Barnes (1987) y Salzberg (1988); explican organizaciones adicionales de ficheros, como los ficheros estructurados en árbol, y tienen algoritmos detallados para las operaciones sobre ficheros. Algunas obras sobre las organizaciones de los ficheros son Miller (1987) y Livadas (1989). Salzberg y otros (1990) describe un algoritmo de ordenación externa distribuida. Las organizaciones de ficheros con un alto grado de tolerancia a los fallos se describe en Bitton and Gray (1988) y en Gray y otros (1990). La segmentación de disco se propone en Salem y García Molina (1986). El primer ensayo sobre las matrices redundantes de discos baratos (RAID) lo realizó Patterson y otros (1988). Chen y Patterson (1990) y el excelente estudio sobre RAID realizado por Chen y otros (1994) son otras referencias. Grochowski y Hoyt (1996) explica las tendencias futuras en lo que se refiere a las unidades de disco. En Chen y otros (1994) aparecen varias fórmulas para la arquitectura RAID.

Morris (1968) es un antiguo ensayo sobre la dispersión. La dispersión extensible se describe en Fagin y otros (1979). La dispersión lineal se describe en Litwin (1980). La dispersión dinámica, que no hemos explicado en detalle, se propuso en Larson (1978). Se han propuesto muchas variaciones de la dispersión extensible y lineal; a modo de ejemplo, consulte Cesarini y Soda (1991), Du y Tong (1991), y Hachem y Berra (1992).

En los sitios de los fabricantes puede encontrar detalles sobre los dispositivos de almacenamiento (por ejemplo, http://www.seagate.com, *http://www.ibm.com*, *http://www.ecm.com*, *http://www.hp.com*, y *http://www.storagetek.com*). IBM dispone de un centro de investigación de tecnología de almacenamiento en IBM Almaden (*http://www.almaden.ibm.com/sst/*).

CAPÍTULO 14

# Estructuras de indexación para los ficheros

En este capítulo asumimos que ya existe un fichero con alguna de las organizaciones primarias: desordenado, ordenado o disperso (consulte el Capítulo 13). En este capítulo vamos a describir unas estructuras de acceso auxiliares denominadas índices, que se utilizan para acelerar la recuperación de registros en respuesta a ciertas condiciones de búsqueda. Las estructuras de índice normalmente proporcionan rutas de acceso, que ofrecen formas alternativas de acceder a los registros sin que se vea afectada la ubicación física de los registros en el disco. Permiten un acceso eficaz a los registros basándose en la indexación de los campos que se utilizan para construir el índice. Básicamente, es posible utilizar *cualquier campo* del fichero para crear un índice y se pueden construir *varios índices* con diferentes campos en el mismo fichero. También son posibles varios tipos de índices; cada uno de ellos utiliza una estructura de datos en particular para acelerar la búsqueda. Para encontrar uno o varios registros del fichero basándose en ciertos criterios de selección de un campo indexado, primero hay que acceder al índice, que apunta a uno o más bloques del fichero donde están almacenados los registros requeridos. Los tipos de índices predominantes están basados en los ficheros ordenados (índices de un nivel) y en estructuras de datos en forma de árbol (índices multinivel, árboles $B^+$). Los índices también pueden construirse basándose en la dispersión o en otras estructuras de datos de búsqueda.

En la Sección 14.1 describimos los diferentes tipos de índices ordenados de un nivel: primario, secundario y agrupado (o de agrupamiento). Si vemos un índice de un nivel como un fichero ordenado, podemos desarrollar índices adicionales para él, lo que nos lleva al concepto de índices multinivel. En esta idea está basado el popular esquema de indexación ISAM (Método de acceso secuencial indexado, *Indexed Sequential Access Method*). En la Sección 14.2 explicamos los índices multinivel. En la Sección 14.3 describimos los árboles B y $B^+$, que son estructuras de datos que se utilizan con frecuencia en los DBMSs para implementar índices multinivel que cambian dinámicamente. Los árboles $B^+$ se han convertido en una estructura predeterminada comúnmente aceptada en la mayoría de los DBMSs relacionales para la generación de índices bajo demanda. La Sección 14.4 está dedicada a las distintas alternativas de acceso a datos en base a una combinación de varias claves. En la Sección 14.5 explicamos cómo podemos utilizar otras estructuras de datos (por ejemplo, la dispersión) para construir índices. También ofrecemos una breve introducción al concepto de los índices lógicos, que dotan de un nivel adicional de indirección a los índices físicos, ya que les permite ser flexibles y extensibles en su organización. La Sección 14.6 resume el capítulo.

# 14.1 Tipos de índices ordenados de un nivel

La idea tras la estructura de acceso de un índice ordenado es parecida a la que hay tras el índice de un libro, que enumera al final de la obra los términos importantes ordenados alfabéticamente, junto con las páginas en las que aparecen. Podemos buscar en un índice en busca de una lista de *direcciones* (números de páginas en este caso) y utilizar esas direcciones para localizar un término en el libro, *buscando* en las páginas especificadas. La alternativa, de no indicarse lo contrario, sería desplazarse lentamente por todo el libro, palabra por palabra, hasta encontrar el término en el que estuviéramos interesados; sería como hacer una búsqueda lineal en un fichero. Por supuesto, la mayoría de los libros tienen información adicional, como títulos de capítulo y de sección, que nos ayuda a encontrar el término sin tener que buscar por todo el libro. Sin embargo, el índice es la única indicación exacta del lugar del libro donde se encuentra cada término.

En un fichero con una estructura de registro dada compuesta por varios campos (o atributos), normalmente se define una estructura de índice con un solo campo del fichero, que se conoce como **campo de indexación** (o **atributo de indexación**).[1] Normalmente, el índice almacena todos los valores del campo de índice, junto con una lista de punteros a todos los bloques de disco que contienen los registros con ese valor de campo. Los valores del índice están ordenados, de modo que podemos hacer una búsqueda binaria en el índice. El fichero de índice es mucho más pequeño que el fichero de datos, por lo que la búsqueda en el índice mediante una búsqueda binaria es razonablemente eficaz. La indexación multinivel (consulte la Sección 14.2) anula la necesidad de una búsqueda binaria a expensas de crear índices al propio índice.

Hay varios tipos de índices ordenados. Un **índice principal** o **primario** se especifica en el *campo clave de ordenación* de un fichero ordenado de registros. Como recordará de la Sección 13.7, se utiliza un campo clave de ordenación para *ordenar físicamente* los registros del fichero en el disco, y cada registro tiene un *valor único* para ese campo. Si el campo de ordenación no es un campo clave (es decir, si varios registros del fichero tienen el mismo valor para el campo de ordenación), podemos utilizar otro tipo de índice, denominado **índice agrupado**. Un fichero puede tener como máximo un campo de ordenación física, por lo que puede tener a lo sumo un índice principal o un índice agrupado, *pero no ambos*. Un tercer tipo de índice, el **índice secundario**, se puede especificar sobre cualquier campo no ordenado de un fichero. Un fichero puede tener varios índices secundarios además de su método de acceso principal. Explicamos estos tipos de índices de un nivel en las siguientes tres subsecciones.

## 14.1.1 Índices principales

Un **índice principal** es un fichero ordenado cuyos registros son de longitud fija con dos campos. El primer campo es del mismo tipo de datos que el campo clave de ordenación (denominado **clave principal**) del fichero de datos, y el segundo campo es un puntero a un bloque del disco (una dirección de bloque). En el fichero índice hay una **entrada de índice** (o **registro de índice**) por cada *bloque* del fichero de datos. Cada entrada del índice tiene dos valores: el valor del campo clave principal para el *primer* registro de un bloque, y un puntero a ese bloque. Nos referiremos a los dos valores de la entrada $i$ del índice como $<K(i), P(i)>$.

Para crear un índice principal en el fichero ordenado de la Figura 13.7, utilizamos el campo Nombre como clave principal, porque es el campo clave de ordenación del fichero (asumiendo que cada valor de Nombre es único). Cada entrada del índice tiene un valor Nombre y un puntero. Las tres primeras entradas del índice son las siguientes:

$<K(1) = (\text{Aaron, Ed}), P(1) = \text{dirección de bloque 1}>$

$<K(2) = (\text{Adams, John}), P(2) = \text{dirección de bloque 2}>$

$<K(3) = (\text{Alexander, Ed}), P(3) = \text{dirección de bloque 3}>$

[1] En este capítulo utilizamos indistintamente los términos *campo* y *atributo*.

La Figura 14.1 ilustra este índice principal. El número total de entradas del índice coincide con el *número de bloques de disco* del fichero de datos ordenado. El primer registro de cada bloque del fichero de datos es el **registro ancla** del bloque, o simplemente **ancla de bloque**.[2]

Los índices se pueden clasificar como densos o escasos. Un **índice denso** tiene una entrada de índice por *cada valor de la clave de búsqueda* (y, por tanto, cada registro) del fichero de datos. Un **índice escaso** (o **no denso**) sólo tiene entradas para algunos de los valores de búsqueda. Por consiguiente, un índice principal es un índice escaso, porque incluye una entrada por cada bloque de disco del fichero de datos y las claves de su registro ancla, en lugar de una entrada por cada valor de búsqueda (es decir, por cada registro).

El fichero de índice para un índice principal necesita menos bloques que el fichero de datos, por dos razones. En primer lugar, hay *menos entradas de índice* que registros en el fichero de datos. En segundo lugar, cada entrada del índice tiene normalmente un *tamaño más pequeño* que un registro de datos, porque sólo tiene dos campos; en consecuencia, en un bloque entran más entradas de índice que registros de datos. Por tanto, una búsqueda binaria en el fichero de índice requiere menos accesos a bloques que una búsqueda binaria en el fichero de datos. En referencia a la Tabla 13.2, observe que la búsqueda binaria en un fichero de datos ordenado requiere $\log_2 b$ accesos a bloques. Pero si el fichero de índice principal contiene $b_i$ bloques, la localización de un registro con un valor de clave de búsqueda requiere una búsqueda binaria de ese índice y acceder al bloque que contiene ese registro: un total de $\log_2 b_i + 1$ accesos.

Un registro cuyo valor de la clave principal es $K$ consiste en el bloque cuya dirección es $P(i)$, donde $K(i) \leq K < K(i+1)$. El bloque número $i$ del fichero de datos contiene tales registros debido a la ordenación física de los registros del fichero por el campo clave principal. Para recuperar un registro, dado el valor $K$ de su campo clave principal, hacemos una búsqueda binaria en el fichero índice para encontrar la entrada de índice $i$ apropiada, y después recuperar el bloque del fichero de datos cuya dirección es $P(i)$.[3] El Ejemplo 1 ilustra el ahorro en accesos a bloques que se logra cuando se utiliza un índice principal para buscar un registro.

**Ejemplo 1**. Supongamos que tenemos un fichero ordenado con $r = 30.000$ registros almacenados en un disco cuyo tamaño de bloque es $B = 1.024$ bytes. Los registros del fichero son de tamaño fijo y no están extendidos, con una longitud de registro $R = 100$ bytes. El factor de bloqueo para el fichero sería $bfr = \lfloor (B/R) \rfloor = \lfloor (1.024/100) \rfloor = 10$ registros por bloque. El número de bloques necesarios para el fichero es $b = \lceil (r/bfr) \rceil = \lceil (30.000/10) \rceil = 3.000$ bloques. Una búsqueda binaria en el fichero de datos necesitaría aproximadamente $\lceil \log_2 b \rceil = \lceil (\log_2 3.000) \rceil = 12$ accesos a bloques.

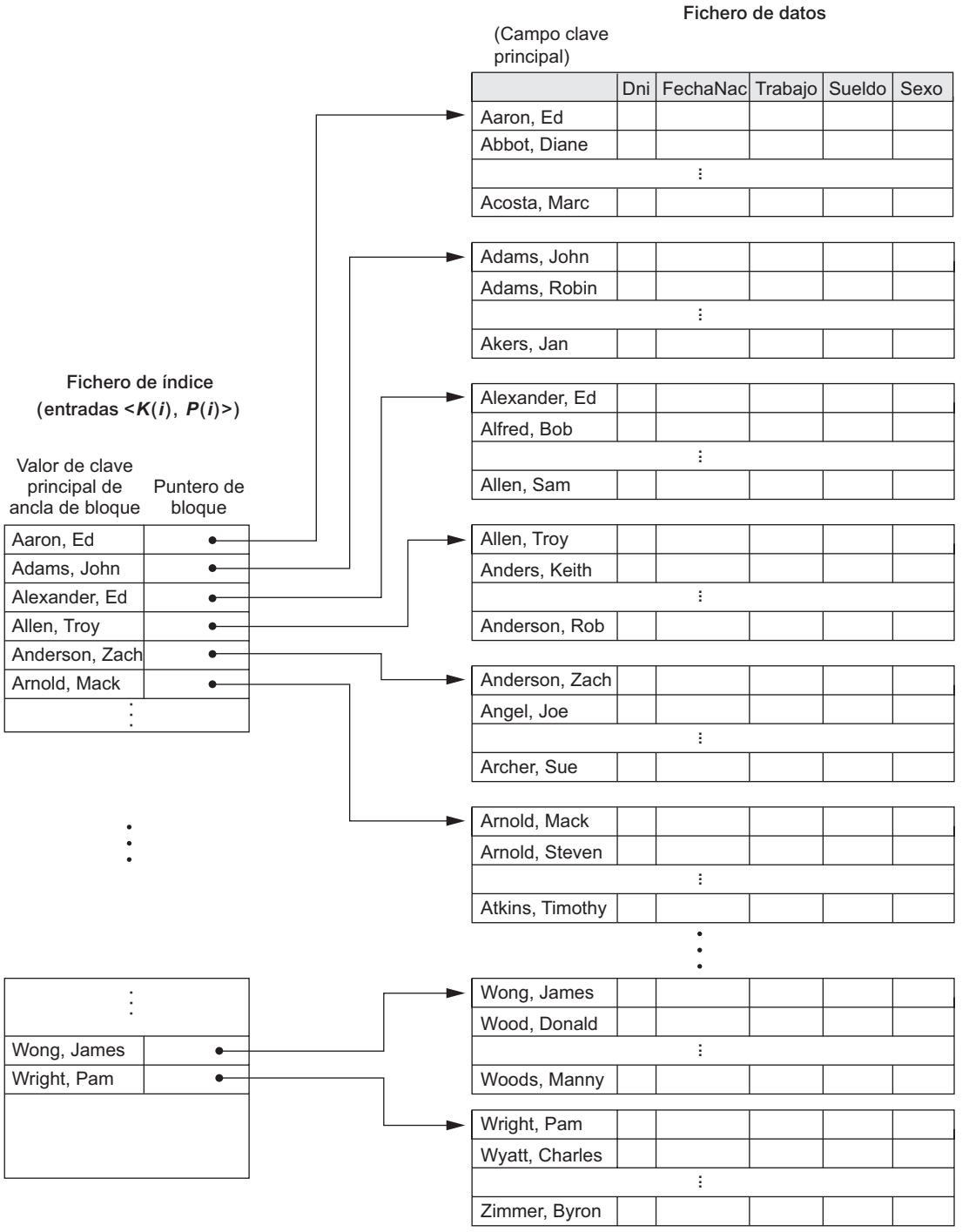
Ahora, suponga que el campo clave de ordenación del fichero tiene una longitud de $V = 9$ bytes, una longitud de puntero de bloque de $P = 6$ bytes, y que hemos construido un índice principal para el fichero. El tamaño de cada entrada del índice es $R_i = (9 + 6) = 15$ bytes, por lo que el factor de bloqueo para el índice es $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (1.024/15) \rfloor = 68$ entradas por boque. El número total de entradas del índice $r_i$ es igual al número de bloques del fichero de datos, que son 3.000. El número de bloques del índice es, por tanto, $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (3.000/68) \rceil = 45$ bloques. Para realizar una búsqueda binaria en el fichero índice necesitaríamos $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 45) \rceil = 6$ accesos a bloques. Para buscar un registro utilizando el índice, necesitamos un acceso a bloque adicional al fichero de datos para un total de $6 + 1 = 7$ accesos a bloques (una mejora sobre la búsqueda binaria en el fichero de datos, que requería 12 accesos a bloques).

La inserción y el borrado de registros supone un gran problema con un índice principal (como con cualquier fichero ordenado). Con un índice principal, el problema se complica porque si queremos insertar un registro en su posición correcta dentro del fichero de datos, no sólo tenemos que mover los registros para hacer sitio al registro nuevo, sino que también tenemos que cambiar algunas entradas del índice, puesto que al mover los

[2] Podemos utilizar un esquema parecido al descrito aquí, con el último registro de cada bloque (en lugar del primero) como ancla de bloque. Esto mejora ligeramente la eficacia del algoritmo de búsqueda.

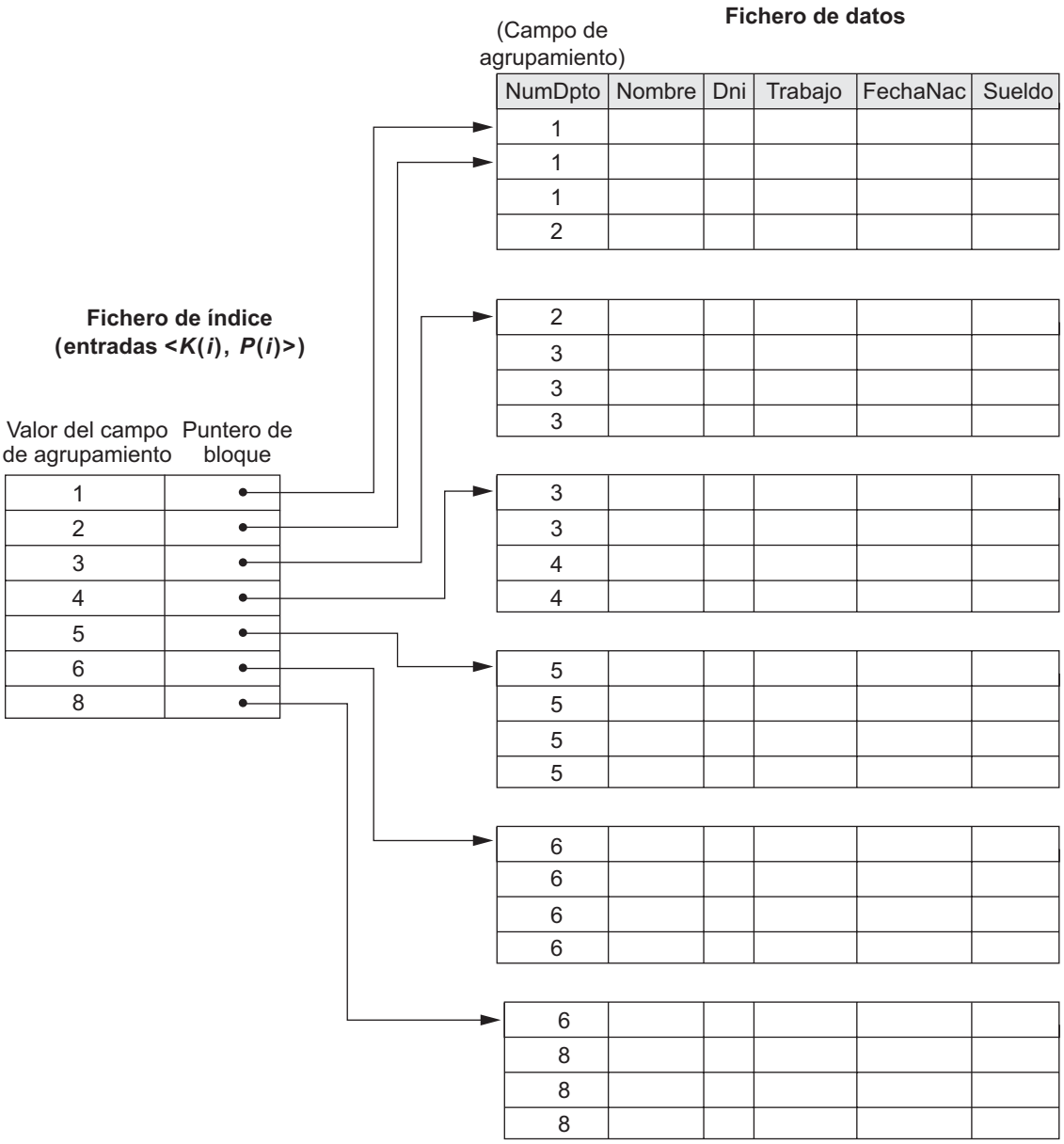
[3] Observe que la fórmula anterior no sería correcta si el fichero de datos estuviera ordenado por un campo no clave; en tal caso, el mismo valor de índice en el ancla de bloque podría repetirse en los últimos registros del bloque anterior.

**Figura 14.1.** El índice principal en el campo clave de ordenación del fichero de la Figura 13.7.




registros modificaremos los registros ancla de algunos bloques. El uso de un fichero de desbordamiento desordenado (consulte la Sección 13.7), puede minimizar este problema. Otra posibilidad es utilizar una lista de enlaces de registros desbordados por cada bloque del fichero de datos. Se parece al método de tratar con los

**Figura 14.2.** Un índice agrupado en el campo no clave de ordenación NumDpto de un fichero EMPLEADO.




registros desbordados con dispersión descrito en la Sección 13.8.2. Los registros de cada bloque y su lista enlazada de desbordamiento se pueden ordenar para mejorar el tiempo de recuperación. El borrado de un registro se realiza con marcadores de borrado.

## 14.1.2 Índices agrupados

Si los registros del fichero están ordenados físicamente por un campo no clave (es decir, un campo que no tiene un valor distinto para cada registro), este campo se denomina **campo agrupado**. Podemos crear un tipo de índice diferente, denominado **índice agrupado**, para acelerar la recuperación de los registros que tienen el

mismo valor para el campo agrupado. Esto difiere de un índice principal, que requiere que el campo de ordenación del fichero de datos tenga un *valor distinto* para cada registro. Un índice agrupado también es un fichero ordenado con dos campos; el primero es del mismo tipo que el campo agrupado del fichero de datos, y el segundo es un puntero a un bloque. En el índice agrupado hay una entrada por cada *valor distinto* del campo agrupado, que contiene el valor y un puntero al *primer bloque* del fichero de datos que tiene un registro con ese valor para su campo agrupado. La Figura 14.2 muestra un ejemplo. La inserción y el borrado de un registro todavía provoca problemas, porque los registros de datos están ordenados físicamente. Para aliviar el problema de la inserción, se suele reservar un bloque entero (o un grupo de bloques contiguos) para *cada valor* del campo agrupado; todos los registros con ese valor se colocan en el bloque (o grupo de bloques). Esto hace que la inserción y el borrado sean relativamente directos. La Figura 14.3 muestra este esquema.

Un índice agrupado es otro ejemplo de índice no denso porque tiene una entrada por cada *valor distinto* del campo de indexación, que no es una clave por definición y, por tanto, tiene valores duplicados en lugar de un valor único por cada registro del fichero. Hay alguna similitud entre las Figuras 14.1 a 14.3, por un lado, y la Figura 13.11, por otro. Un índice es algo parecido a las estructuras de directorio utilizadas para la dispersión extensible que explicamos en la Sección 13.8.3. Ambas estructuras se exploran para encontrar un puntero al bloque de datos que contiene el registro deseado. Una diferencia importante es que una búsqueda en un índice utiliza los valores del propio campo de búsqueda, mientras que una búsqueda en un directorio disperso utiliza el valor de dispersión que se calcula aplicando la función de dispersión al campo de búsqueda.

### 14.1.3 Índices secundarios

Un **índice secundario** proporciona un medio secundario de acceso a un fichero para el que ya existe algún acceso principal. El índice secundario puede ser un campo que es una clave candidata y tiene un valor único en cada registro, o puede ser una "no clave" con valores duplicados. El índice es un fichero ordenado con dos campos. El primero es del mismo tipo de datos que algún *campo no ordenado* del fichero de datos que es un **campo de indexación**. El segundo campo es un puntero de *bloque* o un puntero de *registro*. Puede haber muchos índices secundarios (y, por tanto, campos de indexación) para el mismo fichero.

Primero consideramos que la estructura de acceso de un índice secundario en un campo clave tiene un *valor distinto* por cada registro. Dicho campo recibe a veces el nombre de **clave secundaria**. En este caso, hay una entrada de índice por *cada registro* del fichero de datos, que contiene el valor de la clave secundaria para el registro y un puntero al bloque en el que está almacenado el registro o al propio registro. Por tanto, dicho índice es **denso**.

De nuevo nos referimos a los dos valores de la entrada de índice $i$ como $<K(i), P(i)>$. Las entradas están **ordenadas** por el valor de $K(i)$, por lo que podemos efectuar una búsqueda binaria. Como los registros del fichero de datos no están físicamente ordenados por los valores del campo clave secundario, *no podemos utilizar* las anclas de bloque. Por eso creamos una entrada de índice por cada registro del fichero de datos, en lugar de hacerlo por cada bloque, como en el caso de un índice principal. La Figura 14.4 ilustra un índice secundario en el que los punteros $P(i)$ de las entradas de índice son *punteros de bloque*, y no punteros de registro. Una vez transferido el bloque adecuado a la memoria principal, puede efectuarse una búsqueda del registro deseado dentro del bloque.

Un índice secundario normalmente necesita más espacio de almacenamiento y un tiempo de búsqueda mayor que un índice principal, debido a su mayor cantidad de entradas. Sin embargo, la *mejora* en el tiempo de búsqueda de un registro arbitrario es mucho mayor para un índice secundario que para un índice principal, puesto que tendríamos que hacer una *búsqueda lineal* en el fichero de datos si no existiera el índice secundario. En el caso de un índice principal, todavía podríamos utilizar una búsqueda binaria en el fichero principal, aun cuando no existiera el índice. El Ejemplo 2 ilustra la mejora en el número de bloques accedidos.

**Ejemplo 2**. Considere el fichero del Ejemplo 1 con $r = 30.000$ registros de longitud fija y un tamaño $R = 100$ bytes almacenados en un disco con un tamaño de bloque de $B = 1.024$ bytes. El fichero tiene $b = 3.000$

**Figura 14.3.** Índice agrupado con un grupo de bloques separados por cada grupo de registros que comparten el mismo valor del campo agrupado.

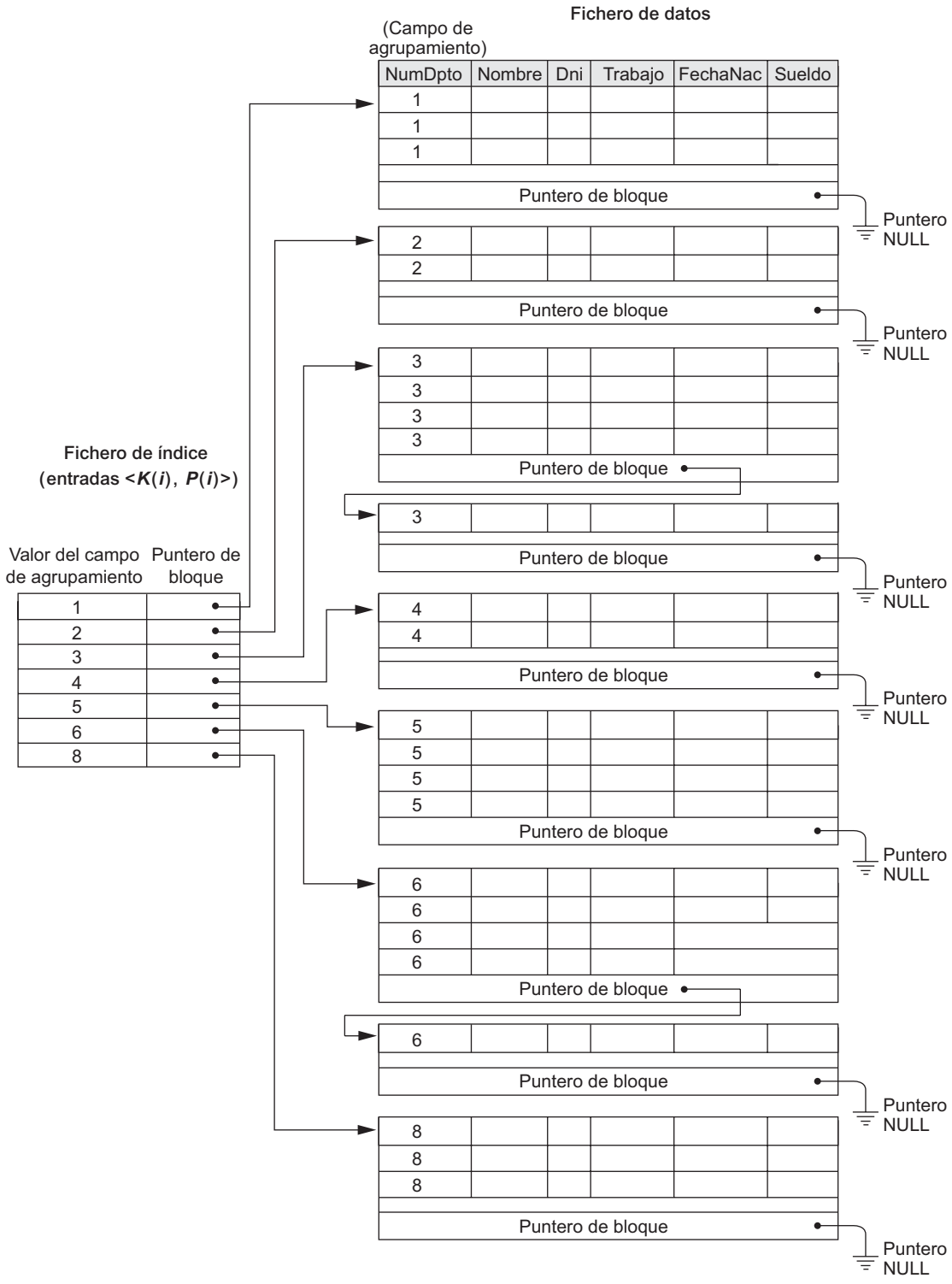
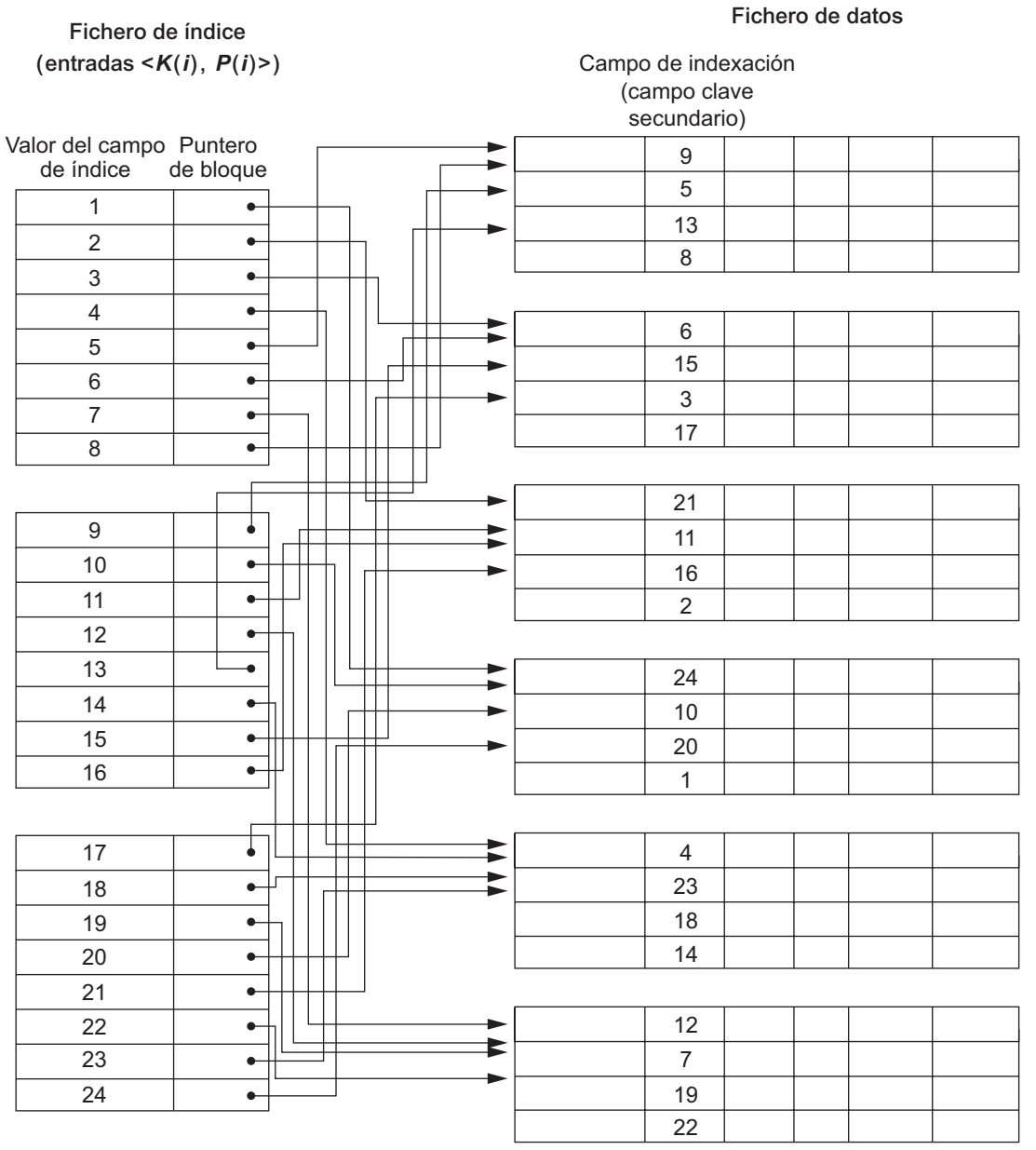


**Figura 14.4.** Índice secundario denso (con punteros de bloque) en un campo clave desordenado de un fichero.




bloques, según calculamos en el Ejemplo 1. Para efectuar una búsqueda lineal en el fichero, necesitaríamos acceder a b/2 = 3.000/2 = 1.500 bloques por término medio. Suponga que creamos un índice secundario en un campo clave desordenado del fichero y que tiene una longitud de $V = 9$ bytes. Como en el Ejemplo 1, un puntero de bloque tiene una longitud $P = 6$ bytes, por lo que cada entrada de índice es de $R_i = (9 + 6) = 15$ bytes, y el factor de bloqueo para el índice es de $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (1024/15) \rfloor = 68$ entradas por bloque. En un índice secundario denso como éste, la cantidad total de entradas de índice $r_i$ es igual al *número de registros* del fichero de datos, es decir, 30.000. El número de bloques necesarios para el índice es, por tanto, $b_i = \lceil (r_i / bfr_i) \rceil = \lceil (3.000/68) \rceil = 442$ bloques.

Una búsqueda binaria en este índice secundario necesita acceder a $\lceil(\log_2 b_i)\rceil = \lceil(\log_2 442)\rceil = 9$ bloques. Para buscar un registro utilizando el índice, necesitamos un acceso a bloque adicional al fichero de datos para un total de 9 + 1 = 10 accesos a bloques (una enorme mejora respecto a los 1.500 accesos a bloques necesarios por término medio para una búsqueda lineal, pero es una cifra ligeramente peor que los siete accesos a bloques requeridos por un índice principal).

También podemos crear un índice secundario en un campo que no sea una clave del fichero. En este caso, varios registros del fichero de datos pueden tener el mismo valor para el campo de indexación. Hay varias opciones para implementar dicho índice:

- Opción 1. Incluir varias entradas de índice con el mismo valor $K(i)$, una por cada registro. Sería un índice denso.
- Opción 2. Tener registros de longitud variable para las entradas del índice, con un campo repetitivo para el puntero. Guardamos una lista de punteros $<P(i,1), ..., P(i, k)>$ en la entrada de índice para $K(i)$ [un puntero a cada bloque que contiene un registro cuyo valor del campo de indexación es igual a $K(i)$]. En cualquiera de estas dos opciones que hemos visto por ahora, es preciso modificar apropiadamente el algoritmo de búsqueda binaria en el índice.
- Opción 3. Es la más utilizada y consiste en mantener las entradas del índice con una longitud fija y disponer de una sola entrada por cada *valor del campo de índice*, pero crear un nivel de indirección extra para manipular los múltiples punteros. En este esquema no denso, el puntero $P(i)$ en la entrada de índice $<K(i), P(i)>$ apunta a un *bloque de punteros de registro*; cada puntero de registro de ese bloque apunta a uno de los registros del fichero de datos cuyo campo de indexación tiene el valor K(i). Si algún valor $K(i)$ se repite en demasiados registros, de modo que sus punteros de registro no pueden encajar en un solo bloque de disco, se utiliza un grupo o lista enlazada de bloques. Esta técnica se ilustra en la Figura 14.5. La recuperación a través de un índice requiere uno o más accesos a bloques debido al nivel extra, pero los algoritmos de búsqueda en el índice y (lo que es más importante) de inserción de registros nuevos en el fichero de datos son directos. Además, las recuperaciones en condiciones de selección complejas pueden manipularse haciendo referencia a los punteros de registro, sin tener que recuperar muchos registros innecesarios del fichero (consulte el Ejercicio 14.19).

Un índice secundario proporciona una **ordenación lógica** de los registros por el campo de indexación. Si accedemos a los registros según el orden de las entradas del índice secundario, los obtendremos ordenados por el campo de indexación.

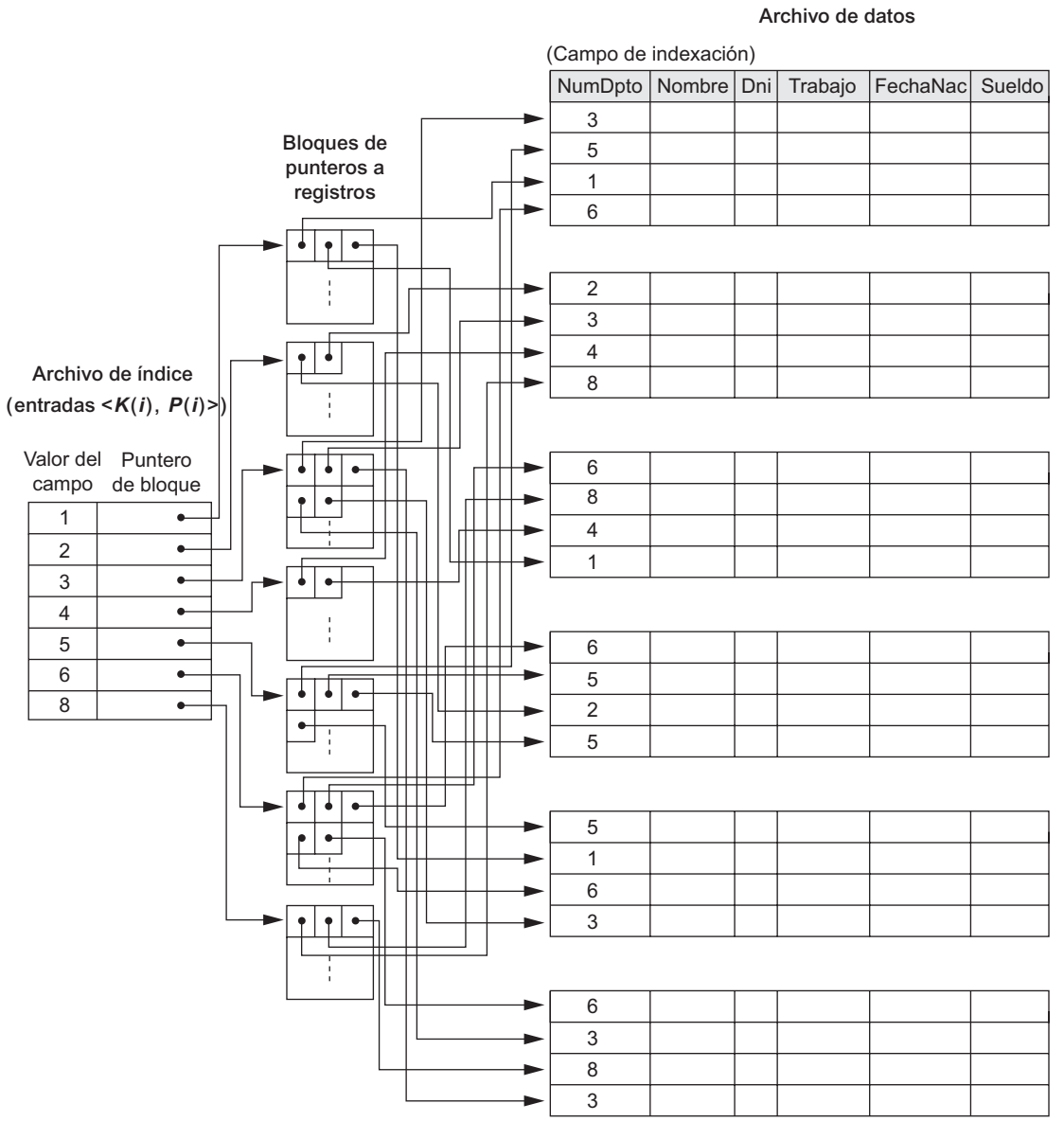
### 14.1.4 Resumen

Para concluir esta sección, resumimos la explicación de los tipos de índices en dos tablas. La Tabla 14.1 muestra las características del campo de índice de cada tipo de índice ordenado de un solo nivel: principal, agrupado y secundario. La Tabla 14.2 resume las propiedades de cada tipo de índice comparando el número de entradas de índice y especificando los índices que son densos y los que utilizan anclas de bloque del fichero de datos.

**Tabla 14.1.** Tipos de índices basados en las propiedades del campo de indexación.

| | **Campo de índice utilizado para ordenar el fichero** | **Campo de índice no usado para ordenar el fichero** |
|---|---|---|
| El campo de indexación es una clave | Índice principal | Índice secundario (clave) |
| El campo de indexación no es una clave | Índice agrupado | Índice secundario (no clave) |

**Figura 14.5.** Índice secundario (con punteros de registro) en un campo que no es índice e implementado utilizando un nivel de indirección, de modo que las entradas del índice son de longitud fija y tienen valores de campo únicos.




# 14.2 Índices multinivel

Los esquemas de indexación hasta ahora descritos implican un fichero de índice ordenado. Al índice se le aplica una búsqueda binaria para localizar los punteros a un bloque de disco o a un registro (o registros) del fichero que tiene un valor de campo de índice específico. Una búsqueda binaria requiere aproximadamente ($\log_2 b_i$) accesos a bloques para un índice con $b_i$ bloques, porque cada paso del algoritmo reduce por un factor de 2 la parte del fichero de índice que continuamos explorando. Por eso, tomamos la función log en base 2. La idea

**Tabla 14.2.** Propiedades de los tipos de índice.

| Tipo de índice | Número de entradas de índice (primer nivel) | Denso no denso | Bloque de anclaje en el fichero de datos |
|---|---|---|---|
| Principal | Número de bloques en el fichero de datos | No denso | Sí |
| Agrupado | Número de valores del campo de índice distintos | No denso | Sí/no[a] |
| Secundario (clave) | Número de registros del fichero de datos | Denso | No |
| Secundario (no clave) | Número de registros[b] o número de valores del campo de índice distintos[c] | Denso o no denso | No |

[a] Sí, si cada valor distinto del campo de ordenación inicia un nuevo bloque; no, en cualquier otro caso.

[b] Para la opción 1.

[c] Para las opciones 2 y 3.

que hay detrás de un **índice multinivel** es reducir la parte del índice que continuamos explorando por $bfr_i$, el factor de bloqueo para el índice, que es mayor que 2. Por tanto, el espacio de búsqueda se reduce mucho más rápidamente. El valor $bfr_i$ se denomina ***fan-out*** del índice multinivel, al que nos referiremos mediante el símbolo ***fo***. La búsqueda en un índice multinivel requiere aproximadamente $(\log_{fo} b_i)$ accesos a bloques, que es un número más pequeño que para la búsqueda binaria si el *fan-out* es mayor que 2.

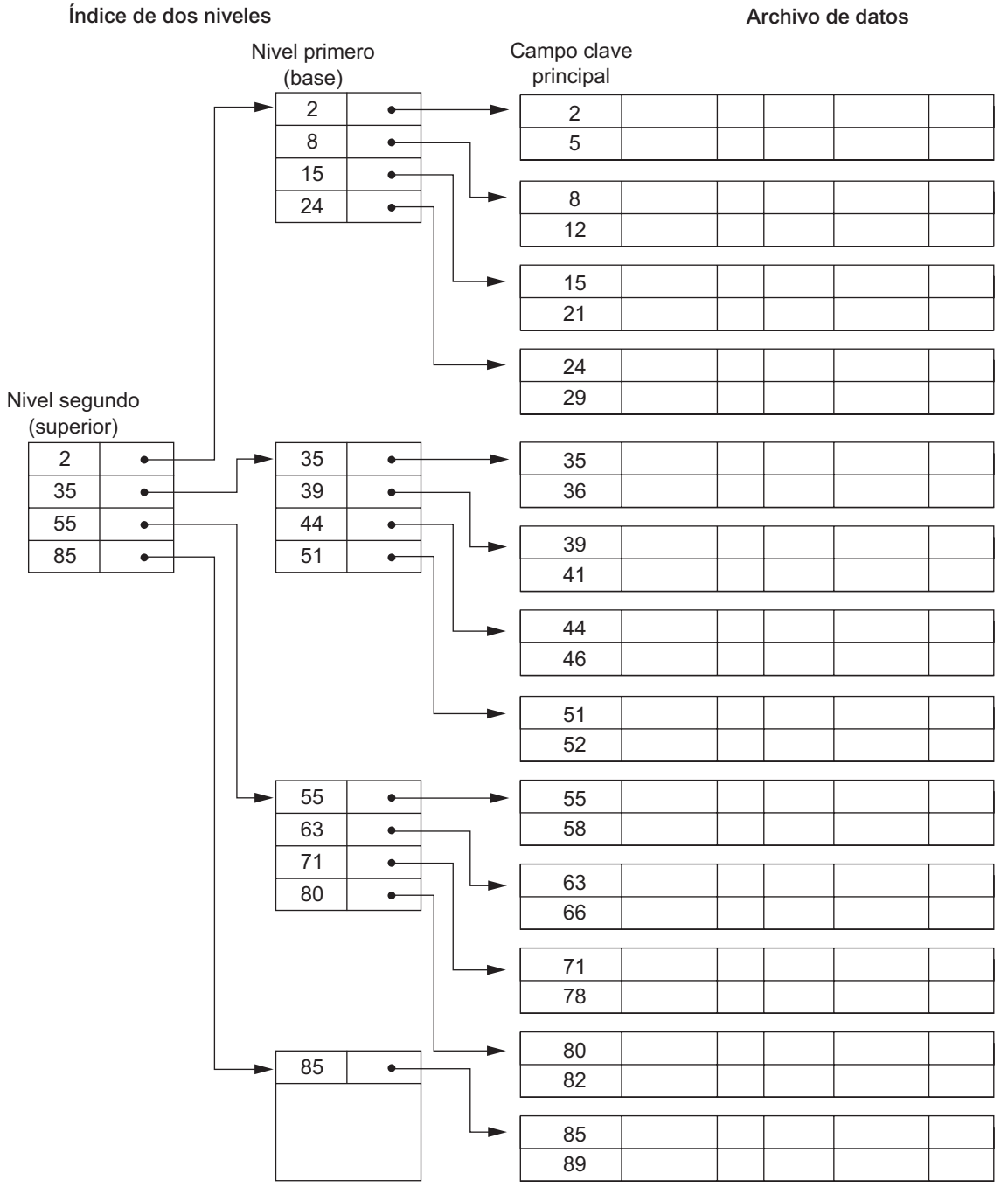
Un índice multinivel considera el fichero de índice, al que ahora nos referiremos como **primer nivel** (o **base**) de un índice multinivel, como un *fichero ordenado* con un *valor distinto* por cada K(i). Por consiguiente, podemos crear un índice principal para el primer nivel; este índice al primer nivel se denomina **segundo nivel** del índice multinivel. Como el segundo nivel es un índice principal, podemos utilizar anclas de bloque de modo que el segundo nivel tenga una sola entrada por *cada bloque* del primer nivel. El factor de bloqueo $bfr_i$ para el segundo nivel (y para todos los niveles subsiguientes) es el mismo que para el índice de primer nivel porque todas las entradas del índice tienen el mismo tamaño; cada una con un valor de campo y una dirección de bloque. Si el primer nivel tiene $r_1$ entradas, y el factor de bloqueo (que también es el *fan-out*) para el índice es $bfr_i = fo$, entonces el primer nivel necesita $\lceil (r_1/fo) \rceil$ bloques, que es por consiguiente el número de entradas $r_2$ necesarias en el segundo nivel del índice.

Podemos repetir este proceso para el segundo nivel. El **tercer nivel**, que es un índice principal para el segundo nivel, tiene una entrada por cada bloque de segundo nivel, por lo que el número de entradas de tercer nivel es $r_3 = \lceil (r_2/fo) \rceil$. Observe que necesitamos un segundo nivel sólo si el primer nivel necesita más de un bloque de almacenamiento en disco y, de forma parecida, requerimos un tercer nivel sólo si el segundo nivel necesita más de un bloque. Podemos repetir el proceso anterior hasta que todas las entradas del mismo nivel del índice $t$ encajen en un solo bloque. Este bloque en el nivel $t$ se denomina nivel de índice **superior**.[4] Cada nivel reduce el número de entradas en el nivel previo por un factor de $fo$ (el índice *fan-out*) por lo que podemos utilizar la fórmula $1 \leq (r_1/((fo)^t))$ para calcular $t$. Por tanto, un índice multinivel con $r_1$ entradas de primer nivel tendrá aproximadamente $t$ niveles, donde $t = \lceil (\log_{fo}(r_1)) \rceil$.

El esquema multinivel aquí descrito puede utilizarse en cualquier tipo de índice, sea principal, agrupado o secundario (siempre y cuando el índice de primer nivel tenga *valores distintos para K(i) y entradas de longitud fija*). La Figura 14.6 muestra un índice multinivel construido sobre un índice principal. El Ejemplo 3 ilustra la mejora en el número de bloques accedidos cuando se utiliza un índice multinivel para buscar un registro.

[4] El esquema de numeración aquí utilizado para los niveles del índice es lo contrario a como normalmente se definen los niveles para las estructuras de datos en árbol. En estas últimas, $t$ es como el nivel 0 (cero), $t–1$ es el nivel 1, etcétera.

**Figura 14.6.** Índice principal de dos niveles que se parece a la organización ISAM.




**Ejemplo 3**. Supongamos que el índice secundario denso del Ejemplo 2 se convierte en un índice multinivel. Calculamos el factor de bloqueo del índice como $bfr_i = 68$ entradas de índice por bloque, que también es el *fan-out fo* para el índice multinivel; también hemos calculado el número de bloques de primer nivel $b_1 = 442$ bloques. El número de bloques de segundo nivel será $b_2 = \lceil (b_1/fo) \rceil = \lceil (442/68) \rceil = 7$ bloques, y el número

de bloques de tercer nivel será $b_3 = \lceil (b_2/fo) \rceil = \lceil (7/68) \rceil = 1$ bloque. Por tanto, el tercer nivel es el nivel superior del índice, y $t = 3$. Para acceder a un registro buscando en el índice multinivel, debemos acceder a un bloque en cada nivel más un bloque del fichero de datos, por lo que necesitamos $t + 1 = 3 + 1 = 4$ accesos a bloques. Compare esto con el Ejemplo 2, donde necesitábamos 10 accesos a bloques cuando utilizábamos un índice de un solo nivel y una búsqueda binaria.

Observe que también podríamos tener un índice principal multinivel, que no sería denso. El Ejercicio 14.14(c) ilustra este caso, donde *debemos* acceder al bloque de datos del fichero antes de poder determinar si el registro buscado está en el fichero. En el caso de un índice denso, esto puede determinarse accediendo al primer nivel del índice (sin tener que acceder a un bloque de datos), puesto que hay una entrada de índice por *cada* registro del fichero.

Una organización de fichero bastante común en el procesamiento de datos empresarial es un fichero ordenado con un índice principal multinivel en su campo clave de ordenación. Una organización de este tipo se conoce como **fichero secuencial indexado** y se utilizó en un buen número de los antiguos sistemas IBM.

La inserción se realiza con alguna forma de fichero de desbordamiento que se combina periódicamente con el fichero de datos. El índice se vuelve a crear durante la reorganización del fichero. La organización **ISAM** de IBM incorpora un índice de dos niveles que está estrechamente relacionado con la organización del disco. El primer nivel es un índice de cilindro, que tiene el valor clave de un registro ancla por cada cilindro del paquete de discos y un puntero al índice de pista para el cilindro. El índice de pista tiene el valor clave de un registro de ancla por cada pista del cilindro y un puntero a la pista. Después se puede buscar secuencialmente en la pista el registro o bloque deseado.

El Algoritmo 14.1 esboza el procedimiento de búsqueda de un registro en un fichero de datos que utiliza un índice principal multinivel no denso con $t$ niveles. Nos referimos a la entrada $i$ del nivel $j$ del índice como $<K_j(\text{i}), P_j(\text{i})>$, y buscamos un registro cuyo valor de clave principal es $K$. Asumimos que se ignoran los registros desbordados. Si el registro está en el fichero, debe haber alguna entrada a nivel 1 con $K_1(\text{i}) \leq K < K_1(i+1)$ y el registro estará en el bloque del fichero de datos cuya dirección es $P_1(i)$. El Ejercicio 14.19 explica la modificación del algoritmo de búsqueda para otros tipos de índices.

**Algoritmo 14.1. Buscando un índice principal multinivel no denso con *t* niveles.**

```
p ← dirección del bloque de nivel superior del índice;
for j ← t step − 1 to 1 do
    begin
        leer el bloque de índice (en el nivel de índice número j) cuya dirección es p;
        buscar bloque p para la entrada i tal que Kj(i) ≤ K < Kj(i + 1) (if Kj(i)
        es la última entrada del bloque, es suficiente satisfacer Kj(i) ≤ K);
        p ← Pj(i) (∗ elegir el puntero apropiado en el nivel de índice j ∗)
    end;
    leer el bloque del fichero de datos cuya dirección es p;
    buscar en el bloque p el registro con la clave = K;
```

Como hemos visto, un índice multinivel reduce el número de accesos a bloques cuando se busca un registro, dado el valor de su campo de indexación. Todavía nos enfrentamos con los problemas de tratar con las inserciones y los borrados en el índice, porque todos los niveles de índice son *ficheros ordenados físicamente*. Para mantener los beneficios de utilizar la indexación multinivel a la vez que se reducen los problemas de la inserción y la eliminación en el índice, los diseñadores adoptaron un índice multinivel denominado **índice multinivel dinámico** que deja algo de espacio en cada uno de sus bloques para insertar nuevas entradas. Esto se implementa normalmente utilizando estructuras de datos denominadas árboles B y árboles $B^+$, que describimos en la siguiente sección.

# 14.3 Índices multinivel dinámicos utilizando árboles B y B+

Los árboles B y B+ son casos especiales de estructuras de datos en forma de árbol bien conocidas, de cuya terminología hacemos una introducción. Un **árbol** está formado por **nodos**. Cada nodo del árbol, excepto el nodo especial denominado **raíz**, tiene un nodo **padre** y varios (ninguno o más) nodos **hijo**. El nodo raíz no tiene padre. Un nodo que no tiene ningún nodo hijo se llama nodo **hoja**; un nodo que no es hoja es un nodo **interno**. El **nivel** de un nodo siempre es uno más que el nivel de su padre, siendo cero el nivel del nodo raíz.[5] Un **subárbol** de un nodo consta de ese nodo y de todos sus nodos **descendientes** (sus nodos hijo, los nodos hijo de sus nodos hijo, etcétera). Una definición recursiva precisa de subárbol es que consta de un nodo *n* y de los subárboles de todos los nodos hijo de *n*. La Figura 14.7 ilustra una estructura de datos en árbol. En esta figura, el nodo raíz es A, y sus nodos hijo son B, C y D. Los nodos E, J, C, G, H y K son nodos hoja.

Normalmente, mostramos un árbol con el nodo raíz en la parte superior (véase la Figura 14.7). Una forma de implementar un árbol es con tantos punteros en cada nodo como nodos hijo tiene ese nodo. En algunos casos, también se almacena un puntero padre en cada nodo. Además de los punteros, un nodo normalmente contiene algún tipo de información almacenada. Cuando un índice multinivel se implementa como una estructura en árbol, esa información incluye los valores del campo de indexación del fichero que se utiliza para guiar la búsqueda de un determinado registro.

En la Sección 14.3.1 introducimos los árboles de búsqueda y después explicamos los árboles B, que pueden utilizarse como índices multinivel dinámicos para guiar la búsqueda de los registros en un fichero de datos. Los nodos de un árbol B se mantienen llenos entre un 50% y un 100%, y los punteros a los bloques de datos se almacenan tanto en los nodos internos como en los nodos hoja de la estructura de árbol B. En la Sección 14.3.2 explicamos los árboles B+, una variación de los árboles B en los que los punteros a los bloques de datos de un fichero se almacenan únicamente en los nodos hoja, lo que puede llevar a índices de menos niveles y mayor capacidad.

## 14.3.1 Árboles de búsqueda y árboles B

Un árbol de búsqueda es un tipo especial de árbol que sirve para guiar la búsqueda de un registro, dado el valor de uno de sus campos. Los índices multinivel explicados en la Sección 14.2 pueden imaginarse como una variante de un árbol de búsqueda; cada nodo del índice multinivel puede tener hasta *fo* punteros y *fo* valores clave, donde *fo* es el índice *fan-out*. Los valores del campo de índice de cada nodo nos guía hasta el siguiente nodo, hasta que alcanzamos el bloque del fichero de datos que contiene el registro deseado. Siguiendo un puntero, restringimos nuestra búsqueda en cada nivel a un subárbol del árbol de búsqueda e ignoramos todos los nodos que no están en este subárbol.

**Árboles de búsqueda.** Un árbol de búsqueda es ligeramente distinto a un índice multinivel. Un **árbol de búsqueda** de orden p es un árbol tal que cada nodo contiene, *a lo sumo*, $p-1$ valores de búsqueda y $p$ punteros en el orden $< P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q >$, donde $q \leq$ p; cada $P_i$ es un puntero a un nodo hijo (o puntero NULL o nulo); y cada $K_i$ es un valor de búsqueda proveniente de algún conjunto ordenado de valores. Se supone que todos los valores de búsqueda son únicos.[6] La Figura 14.8 ilustra un nodo de un árbol de búsqueda. Un árbol de búsqueda debe cumplir, en todo momento, las siguientes restricciones:

1. Dentro de cada nodo, $K_1 < K_2 < ... < K_{q-1}$

[5] Esta definición estándar del nivel de un nodo de un árbol, que utilizamos a lo largo de la Sección 14.3, es diferente de la que ofrecimos para los índices multinivel en la Sección 14.2.

[6] Esta restricción se puede relajar algo. Si el índice se encuentra en un campo que no es clave, pueden existir valores de búsqueda duplicados y pueden modificarse la estructura de nodos y las reglas de navegación del árbol.

**Figura 14.7.** Estructura de datos en árbol que muestra un árbol desequilibrado.

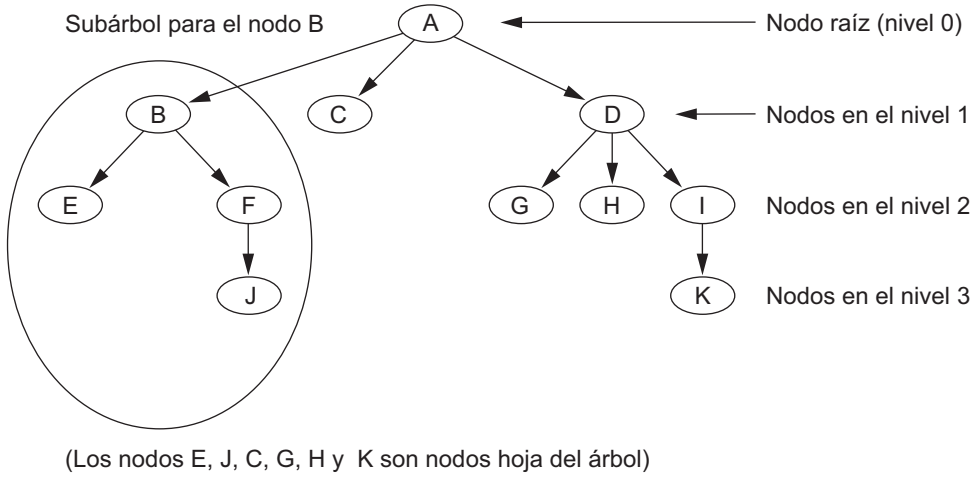
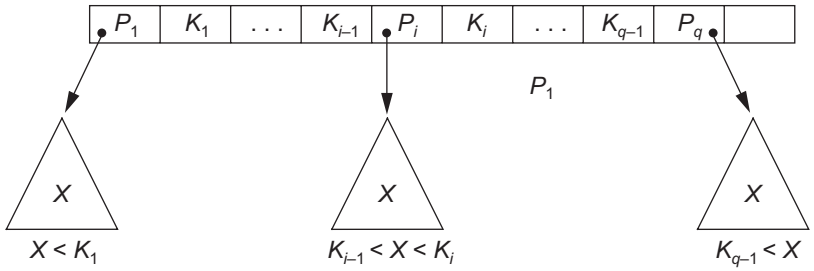



**Figura 14.8.** Nodo de un árbol de búsqueda con punteros a los subárboles que tiene por debajo.




**2.** Para todos los valores de $X$ del subárbol al cual apunta $P_i$, tenemos que $K_{i-1} < X < K_i$ para $1 < i < q$; $X < K_i$ para $i = 1$; y $K_{i-1} < X$ para $i = q$ (véase la Figura 14.8).

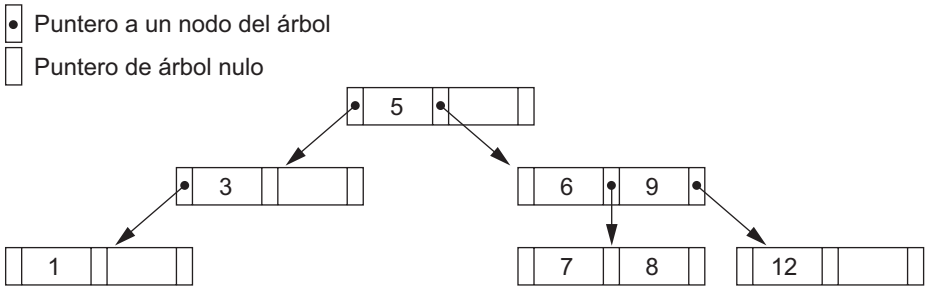
Siempre que buscamos un valor $X$, seguimos el puntero $P_i$ apropiado, de acuerdo con las fórmulas de la condición 2. La Figura 14.9 ilustra un árbol de búsqueda de orden $p = 3$ y valores de búsqueda enteros. Observe que algunos de los punteros $P_i$ de un nodo pueden ser punteros nulos.

Podemos usar un árbol de búsqueda como mecanismo para buscar registros almacenados en un fichero de disco. Los valores del árbol pueden ser los valores de uno de los campos del registro, el llamado **campo de búsqueda** (que es el mismo que el campo de índice si un índice multinivel guía la búsqueda). Cada valor clave del árbol está asociado a un puntero de registro que tiene ese valor en el fichero de datos. Como alternativa, puede apuntar al bloque de disco que contiene ese registro. El árbol de búsqueda en sí puede almacenarse en disco, asignando cada nodo del árbol a un bloque del disco. Cuando se inserta un registro nuevo, es preciso actualizar el árbol de búsqueda incluyendo en él una entrada con el valor del campo de búsqueda del nuevo registro y un puntero a éste.

Para insertar valores de búsqueda en el árbol y eliminarlos, sin violar las dos restricciones anteriores, se utilizan algoritmos que, por regla general, no garantizan que el árbol de búsqueda esté equilibrado (es decir, que todas las hojas estén al mismo nivel).[7] El árbol de la Figura 14.7 no está equilibrado porque tiene nodos hoja en los niveles 1, 2 y 3. Es importante mantener equilibrados los árboles de búsqueda porque esto garantiza

[7] La definición de *equilibrado* es diferente para los árboles binarios. Los árboles binarios equilibrados se conocen como árboles AVL.

**Figura 14.9.** Árbol de búsqueda de orden $p = 3$.




que no habrá nodos en niveles muy profundos que requieran muchos accesos a bloques durante una búsqueda. Los árboles equilibrados procuran una velocidad de búsqueda uniforme independientemente del valor de la clave de búsqueda. Además, las eliminaciones de registros pueden hacer que queden nodos casi vacíos, con lo que hay un desperdicio de espacio importante y un aumento en el número de niveles. El árbol B hace frente a estos dos problemas especificando restricciones adicionales en el árbol de búsqueda.

**Árboles B.** El árbol B tiene restricciones adicionales que garantizan que el árbol siempre estará equilibrado y que el espacio desperdiciado por la eliminación, si lo hay, nunca será excesivo. Los algoritmos para insertar y eliminar son más complejos para poder mantener estas restricciones. No obstante, la mayor parte de las inserciones y eliminaciones son procesos simples que se complican sólo en circunstancias especiales (por ejemplo, siempre que intentamos una inserción en un nodo que ya está lleno o una eliminación de un nodo que está a menos de la mitad de su capacidad). De manera más formal, un **árbol B** de **orden *p***, utilizado como estructura de acceso según un *campo clave* para buscar registros en un fichero de datos, se puede definir de este modo:

1. Cada nodo interno del árbol B (véase la Figura 14.10[a]) tiene la forma:

   $<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>,..., <K_{q-1}, Pr_{q-1}>, P_q>$

   donde $q \leq p$. Cada $P_i$ es un **puntero de árbol** (un puntero a otro nodo del árbol B). Cada $P_{ri}$ es un **puntero de datos**[8] (un puntero al registro cuyo valor del campo clave de búsqueda es igual a $K_i$ [o al bloque del fichero de datos que contiene ese registro]).

2. Dentro de cada nodo, $K_1 < K_2 < \ldots < K_{q-1}$.

3. Para todos los valores del campo clave de búsqueda $X$ del subárbol al que apunta $P_i$ (el subárbol $i$; véase la Figura 14.10[a]), tenemos:

   $K_{i-1} < X < K_i$ para $1 < i < q$; $X < K_i$ para $i = 1$; y $K_{i-1} < X$ para $i = q$.

4. Cada nodo tiene a lo sumo $p$ punteros de árbol.

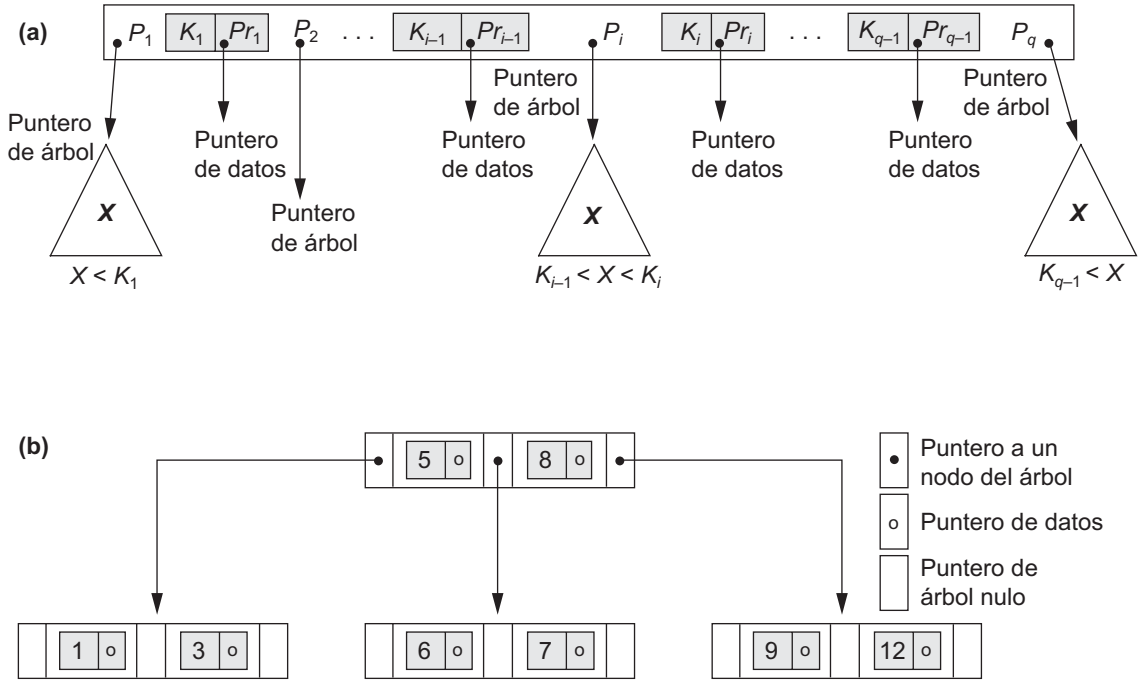
5. Cada nodo, excepto los nodos raíz y hoja, tienen por lo menos $\lceil (p/2) \rceil$ punteros de árbol. El nodo raíz tiene como mínimo dos punteros de árbol, a menos que sea el único nodo del árbol.

6. Un nodo con $q$ punteros de árbol, $q \leq p$, tiene $q - 1$ valores del campo clave de búsqueda (y, por tanto, tiene $q - 1$ punteros de datos).

7. Todos los nodos hoja se encuentran en el mismo nivel. Los nodos hoja tienen la misma estructura que los internos, excepto que todos sus *punteros de árbol* $P_i$ son nulos.

[8] Un puntero de datos es una dirección de bloque, o una dirección de registro; este último es esencialmente una dirección de bloque y un desplazamiento de registro dentro del bloque.

**Figura 14.10.** Estructuras de árbol B. (a) Nodo A en un árbol B con $q - 1$ valores de búsqueda. (b) Un árbol B de orden $p = 3$. Los valores se insertaron en este orden: 8, 5, 1, 7, 3, 12, 9, 6.




La Figura 14.10(b) ilustra un árbol B de orden $p = 3$. Observe que todos los valores de búsqueda $K$ del árbol B son únicos porque asumimos que el árbol se utiliza como una estructura de acceso en un campo clave. Si utilizamos un árbol B *en un campo que no es clave*, debemos cambiar la definición de los punteros de fichero $Pr_i$ para que apunten a un bloque (o grupo de bloques) que contienen los punteros a los registros del fichero. Este nivel adicional de indirección es parecido a la Opción 3, explicada en la Sección 14.1.3, para los índices secundarios.

Un árbol B empieza con un solo nodo raíz (que también es un nodo hoja) en el nivel 0 (cero). Una vez que el nodo raíz está lleno con $p - 1$ valores de clave de búsqueda e intentamos insertar otra entrada en el árbol, el nodo raíz se divide en dos nodos en el nivel 1. En el nodo raíz sólo queda el valor central, y el resto de valores se dividen uniformemente entre los otros dos nodos. Cuando un nodo no raíz está lleno y se inserta en él una entrada nueva, dicho nodo se divide en otros dos al mismo nivel, y la entrada central se mueve al nodo padre junto con dos punteros a los nuevos nodos divididos. Si el nodo padre está lleno, también es dividido. La división puede propagarse hasta llegar al nodo raíz, creando un nuevo nivel si se divide la raíz. Aquí no explicamos en profundidad los algoritmos para los árboles B; en cambio, en la siguiente sección hacemos una introducción de los procedimientos de búsqueda e inserción para los árboles $B^+$.

Si la eliminación de un valor provoca que un nodo no llegue a la mitad de su capacidad, es combinado con sus nodos vecinos, algo que también puede propagarse hasta la raíz. Por tanto, la eliminación puede reducir el número de niveles del árbol. Con ayuda de análisis y simulaciones se ha demostrado que, después de numerosas inserciones y eliminaciones aleatorias en un árbol B, los nodos quedan ocupados aproximadamente al 69% de su capacidad cuando se estabiliza el número de valores en el árbol. Esto también es cierto con los árboles $B^+$. Si ocurre esto, la división y combinación de nodos sólo se dará raramente, por lo que la inserción y la eliminación son muy eficaces. Si el número de valores aumenta, el árbol se expandirá sin problemas (aunque podría darse la división de nodos, de modo que algunas inserciones tardarían algo más de tiempo). El Ejemplo 4 ilustra el cálculo del orden $p$ de un árbol B almacenado en disco.

**Ejemplo 4.** Supongamos que el campo de búsqueda tiene una longitud de $V = 9$ bytes, que el tamaño del bloque de disco es $B = 512$ bytes, que un puntero de registro (datos) es $P_r = 7$ bytes, y que un puntero de bloque es $P = 6$ bytes. Cada nodo del árbol B puede tener *como máximo p* punteros de árbol, $p - 1$ punteros de datos, y $p - 1$ valores de campo clave de búsqueda (véase la Figura 14.10[a]). Deben encajar en un solo bloque de disco si cada nodo del árbol B corresponde a un bloque de disco. Por tanto, tenemos:

$$(p * P) + ((p - 1) * (P_r + V)) \leq \text{B}$$
$$(p * 6) + ((p - 1) * (7 + 9)) \leq 512$$
$$(22 * p) \leq 528$$

Podemos elegir que $p$ sea un valor mayor que satisfaga la desigualdad anterior, que da $p = 23$ (no se elige $p = 24$ debido a las razones que exponemos a continuación).

En general, un nodo de un árbol B puede contener información adicional que los algoritmos que manipulan el árbol necesitan, como el número de entradas $q$ en el nodo y un puntero al nodo padre. Por tanto, antes de que hagamos el cálculo precedente para $p$, debemos reducir el tamaño del bloque para que pueda entrar dicha información. A continuación veremos cómo calcular el número de bloques y niveles para un árbol B.

**Ejemplo 5.** Supongamos que el campo de búsqueda del Ejemplo 4 es un campo clave desordenado, y que construimos un árbol B en ese campo. Los nodos del árbol B están al 60% de su capacidad. Cada nodo, por término medio, tendrá $p * 0{,}69 = 23 * 0{,}69$ o aproximadamente 16 punteros y, por tanto, 15 valores de campo clave de búsqueda. El ***fan-out* medio** es $fo = 16$. Podemos empezar en la raíz y ver cuántos valores y punteros pueden existir, por término medio, en cada nivel subsiguiente:

| | | | |
|---|---|---|---|
| Raíz: | 1 nodo | 15 entradas | 16 punteros |
| Nivel 1: | 16 nodos | 240 entradas | 256 punteros |
| Nivel 2: | 256 nodos | 3.840 entradas | 4.096 punteros |
| Nivel 3: | 4.096 nodos | 61.440 entradas | |

En cada nivel calculamos el número de entradas multiplicando el número total de punteros del nivel anterior por 15, la cantidad media de entradas de cada nodo. Por tanto, para un tamaño de bloque, tamaño de puntero y tamaño del campo clave de búsqueda dados, un árbol B de dos niveles almacena $3.840 + 240 + 15 = 4.095$ entradas por término medio; un árbol B de tres entradas alberga 65.535 entradas por termino medio.

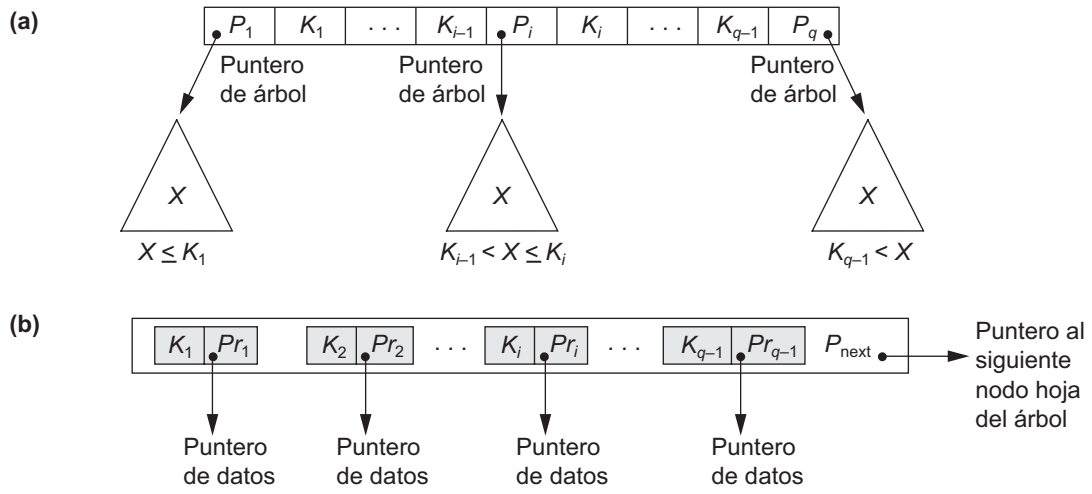
Los árboles B se utilizan a veces como organizaciones de fichero principales. En este caso, dentro de los nodos del árbol B se guardan los registros enteros, en lugar de entradas <clave de búsqueda, puntero de registro>. Esto funciona bien para ficheros con un *número relativamente pequeño de registros* y un *tamaño pequeño de registro*. En caso contrario, el *fan-out* y el número de niveles se hacen demasiado grandes para permitir un acceso eficaz.

En resumen, los árboles B proporcionan una estructura de acceso multinivel que es una estructura de árbol equilibrado en la que cada nodo está, como mínimo, medio lleno. Cada nodo de un árbol B de orden $p$ puede tener como máximo $p - 1$ valores de búsqueda.

## 14.3.2 Árboles B+

La mayoría de las implementaciones de un índice multinivel dinámico utilizan una variante de la estructura de datos en árbol B denominada **árbol B+**. En un árbol B, cada valor del campo de búsqueda aparece una vez en algún nivel del árbol, junto con un puntero de datos. En un árbol $\text{B}^+$ los punteros a datos se almacenan *sólo en los nodos hoja* del árbol, por lo cual, la estructura de los nodos hoja difiere de la de los nodos internos. Los nodos hoja tienen una entrada por *cada valor* del campo de indexación, junto con un puntero al registro (o al bloque que contiene ese registro) si el campo de búsqueda es un campo clave. En el caso de un campo de búsqueda que no es clave, el puntero apunta a un bloque que contiene punteros a los registros del fichero de datos, creándose un nivel extra de indirección.

**Figura 14.11.** Nodos de un árbol $B^+$. (a) Nodo interno de un árbol $B^+$ con $q - 1$ valores de búsqueda. (b) Nodo hoja de un árbol $B^+$ con $q - 1$ valores de búsqueda y $q - 1$ punteros de datos.




Los nodos hoja de un árbol $B^+$ normalmente están enlazados para ofrecer un acceso ordenado a los registros a través del campo de búsqueda. Estos nodos hoja son parecidos al primer nivel (nivel base) de un índice. Los nodos internos del árbol $B^+$ corresponden a los demás niveles de un índice multinivel. Algunos valores del campo de búsqueda de los nodo hoja se *repiten* en los nodos internos del árbol $B^+$ con el fin de guiar la búsqueda. La estructura de los *nodos internos* de un árbol $B^+$ de orden $p$ (véase la Figura 14.11[a]) es la siguiente:

1. Cada nodo interno tiene esta forma:

   $<P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q>$

   donde $q = p$ y cada $P_i$ es un **puntero de árbol**.
2. Dentro de cada nodo interno, $K_1 < K_2 < ... < K_{q-1}$.
3. Para todos los valores $X$ del campo de búsqueda en el subárbol al cual apunta $P_i$, tenemos que $K_{i-1} < X \leq K_i$ para $1 < i < q$; $X \leq K_i$ para $i = 1$; y $K_{i-1} < X$ para $i = q$ (véase la Figura 14.11[a]).[9]
4. Cada nodo interno tiene, a lo sumo, $p$ punteros de árbol.
5. Cada nodo interno, excepto el raíz, tiene al menos $\lceil (p/2) \rceil$ punteros de árbol. El nodo raíz tiene como mínimo dos punteros de árbol si es un nodo interno.
6. Un nodo interno con $q$ punteros, $q \leq p$, tiene $q - 1$ valores del campo de búsqueda.

La estructura de los *nodos hoja* del árbol $B^+$ de orden $p$ (véase la Figura 14.11[b]) es la siguiente:

1. Cada nodo hoja tiene la siguiente forma:

   $<<K_1, Pr_1>, <K_2, Pr_2>, ..., <K_{q-1}, Pr_{q-1}>, P_{sig}>$

   donde $q \leq p$, cada $Pr_i$ es un puntero de datos, y $P_{sig}$ apunta al siguiente *nodo hoja* del árbol $B^+$.
2. Dentro de cada nodo hoja, $K_1 \leq K_2 ..., K_{q-1}$, $q \leq p$.
3. Cada $Pr_i$ es un **puntero a datos** que apunta al registro cuyo valor del campo de búsqueda es $K_i$ o a un bloque del fichero que contiene el registro (o a un bloque de punteros de registro que apuntan a los registros cuyo valor del campo de búsqueda es $K_i$ si el campo de búsqueda no es una clave).

[9] Nuestra definición obedece a Knuth (1973). Un árbol $B^+$ puede definirse de forma diferente intercambiando los símbolos < y ≤ ($K_{i-1} \leq X < K_i$; $K_{q-1} \leq X$), pero los principios siguen siendo los mismos.

4. Cada nodo hoja tiene como mínimo $\lceil (p/2) \rceil$ valores.
5. Todos los nodos hoja están en el mismo nivel.

Los punteros en los nodos internos son *punteros de árbol* a bloques que son nodos del árbol, mientras que los punteros de los nodos hoja son *punteros de datos* a registros o bloques del fichero (excepto para el puntero $P_{sig}$, que es un puntero de árbol al siguiente nodo hoja). Empezando por el nodo hoja más a la izquierda, es posible atravesar los nodos hoja como una lista enlazada, utilizando los punteros $P_{sig}$. Esto ofrece un acceso ordenado a los registros de datos del campo de indexación. También podemos incluir un puntero $P_{anterior}$. Para un árbol $B^+$ en un campo no clave, se necesita un nivel de indirección adicional parecido al de la Figura 14.5, de tal manera que los punteros $P_r$ son punteros de bloque a bloques que contienen un conjunto de punteros de registro a los registros actuales del fichero de datos, como explicamos en la Opción 3 de la Sección 14.1.3.

Como las entradas de los *nodos internos* de un árbol $B^+$ incluyen valores de búsqueda y punteros de árbol sin punteros a los datos, es posible empaquetar más entradas en un nodo interno de un árbol $B^+$ que en un árbol B similar. Por tanto, si el tamaño de bloque (nodo) es el mismo, el orden $p$ será mayor para el árbol $B^+$ que para el árbol B, como se ilustra en el Ejemplo 6. Esto puede reducir el número de niveles del árbol $B^+$, mejorándose así el tiempo de búsqueda. Como las estructuras de los nodos internos y de los nodos hoja de los árboles $B^+$ son diferentes, su orden $p$ puede ser diferente. Utilizaremos $p$ para denotar el orden de los *nodos internos* y $p_{hoja}$ para denotar el orden de los *nodos hoja*, que definimos como el número máximo de punteros a datos en un nodo hoja.

**Ejemplo 6.** Para calcular el orden $p$ de un árbol $B^+$, vamos a suponer que el campo clave de búsqueda tiene una longitud de $V = 9$ bytes, que el tamaño del bloque es de $B = 512$ bytes, que un puntero de registro es de $Pr = 7$ bytes, y que un puntero de bloque es de $P = 6$ bytes, como en el Ejemplo 4. Un nodo interno del árbol $B^+$ puede tener hasta $p$ punteros de árbol y $p - 1$ valores del campo de búsqueda; éstos deben encajar en un solo bloque. Por tanto, tenemos:

$$(p * P) + ((p - 1) * V) \leq B$$
$$(p * 6) + ((p - 1) * 9) \leq 512$$
$$(15 * p) \leq 521$$

Podemos elegir $p$ para que sea el valor más grande que satisfaga la desigualdad anterior, que da $p = 34$. Se trata de un valor mayor que el obtenido para un árbol B (23), lo que ofrece un *fan-out* más grande y más entradas en cada nodo interno de un árbol $B^+$ que en el correspondiente árbol B. Los nodos hoja del árbol $B^+$ tendrán el mismo número de valores y punteros, excepto que los punteros son punteros a datos y un puntero "siguiente". Por tanto, el orden $p_{hoja}$ para los nodos hoja puede calcularse de este modo:

$$(p_{hoja} * (Pr + V)) + P \leq B$$
$$(p_{hoja} * (7 + 9)) + 6 \leq 512$$
$$(16 * p_{hoja}) \leq 506$$

Resulta que cada nodo hoja puede albergar hasta $p_{hoja} = 31$ combinaciones de puntero valor/datos, asumiendo que los punteros de datos son punteros de registro.

Como con el árbol B, necesitamos información adicional (para implementar los algoritmos de inserción y eliminación) en cada nodo. Esta información puede incluir el tipo de nodo (interno u hoja), el número actual de entradas $q$ en el nodo y punteros a los nodos padre y hermanos. Por tanto, antes de que hagamos los cálculos anteriores para $p$ y $p_{hoja}$, debemos reducir el tamaño del bloque en la cantidad de espacio necesario para toda esta información. El siguiente ejemplo ilustra cómo podemos calcular el número de entradas en un árbol $B^+$.

**Ejemplo 7.** Supongamos que construimos un árbol $B^+$ en el campo del Ejemplo 6. Para calcular el número aproximado de entradas del árbol $B^+$, asumimos que cada nodo está ocupado al 69%. Por término medio, cada nodo interno tendrá $34 * 0{,}69$ o aproximadamente 23 punteros, y por tanto 22 valores. Cada nodo hoja, por

término medio, albergará $0{,}69 * p_{\text{hoja}} = 0{,}69 * 31$ o aproximadamente 21 punteros a registros de datos. Un árbol $B^+$ tendrá por término medio la siguiente cantidad de entradas en cada nivel:

| | | | |
|---|---|---|---|
| Raíz: | 1 nodo | 22 entradas | 23 punteros |
| Nivel 1: | 23 nodos | 506 entradas | 529 punteros |
| Nivel 2: | 529 nodos | 11.638 entradas | 12.167 punteros |
| Nivel hoja: | 12.167 nodos | 255.507 punteros a registro de datos | |

Para un tamaño de bloque, tamaño de puntero y tamaño de campo de búsqueda dados, un árbol $B^+$ de tres niveles alberga hasta 255.507 punteros de registro, por término medio. Compare esto con las 65.535 entradas para el árbol B correspondiente del Ejemplo 5.

**Búsqueda, inserción y eliminación con árboles B+.** El Algoritmo 14.2 muestra el procedimiento utilizando el árbol $B^+$ como estructura de acceso para buscar un registro. El Algoritmo 14.3 ilustra el procedimiento para insertar un registro en un fichero con un árbol $B^+$ como estructura de acceso. Estos algoritmos asumen la existencia de un campo clave de búsqueda, y deben modificarse apropiadamente para el caso de un árbol $B^+$ en un campo no clave. Ilustramos la inserción y la eliminación con un ejemplo.

**Algoritmo 14.2. Búsqueda de un registro con un valor *K* del campo clave de búsqueda, utilizando un árbol B+.**

```
n ← bloque que contiene la raíz del árbol B+;
leer el bloque n;
while (n no sea un nodo hoja del árbol B+) do
  begin
  q ← número de punteros de árbol en el nodo n;
  if K ≤ n.K1 (*n.Ki se refiere al valor del campo de búsqueda i en el nodo n*)
     then n ← n.P1 (*n.Pi se refiere al puntero de árbol i en el nodo n*)
     else if K > n.Kq–1
        then n ← n.Pq
        else begin
           buscar el nodo n para una entrada i tal que n.Ki–1 < K ≤ n.Ki ;
           n ← n.Pi
           end;
  leer el bloque n
  end;
buscar en el bloque n la entrada (Ki,Pri) con K = Ki; (* buscar nodo hoja *)
if encontrado
   then leer el bloque del fichero de datos con la dirección Pri y recuperar el registro
   else el registro con el valor de campo de búsqueda K no está en el fichero de datos;
```

**Algoritmo 14.3. Inserción de un registro con un valor de campo clave de búsqueda *K* en un árbol B+ de orden *p*.**

```
n ← bloque que contiene la raíz del árbol B+;
leer el bloque n; establecer la pila S a vacío;
while (n no sea un nodo hoja del árbol B+) do
   begin
   push la dirección de n a la pila S;
      (*la pila S alberga los nodos padre necesarios en el caso de división*)
   q ← número de punteros de árbol en el nodo n;
```

```
    if K ≤ n.K1 (∗n.Ki se refiere al valor del campo de búsqueda i en el nodo n∗)
       then n ← n.P1 (∗n.Pi se refiere al puntero de árbol i en el nodo n∗)
       else if K > n.Kq–1
          then n ← n.Pq
          else begin
             buscar el nodo n para una entrada i tal que n.Ki–1 < K ≤ n.Ki;
             n ← n.Pi
             end;
        leer el bloque n
    end;
buscar en el bloque n la entrada (Ki,Pri) con K = Ki; (∗buscar el nodo hoja n∗)
if encontrado
    then el registro ya existe en el fichero; no se puede insertar
    else (∗insertar la entrada en el árbol B+ para apuntar al registro∗)
        begin
        crear la entrada (K, Pr) donde Pr apunta al registro nuevo;
        if el nodo hoja n no está lleno
           then insertar la entrada (K, Pr) en la posición correcta en el nodo hoja n
           else begin (∗el nodo hoja n está lleno con phoja punteros de registro; se divide∗)
              copiar n en temp (∗temp es un nodo hoja muy grande para almacenar la entrada extra∗);
              insertar la entrada (K, Pr) en la posición correcta de temp;
              (∗temp alberga ahora phoja + 1 entradas en forma de (Ki, Pri)∗)
              nuevo ← un nuevo nodo hoja vacío para el árbol; nuevo.Psig ← n.Psig;
              j ← ⌈(phoja+1)/2⌉;
              n ← primeras j entradas de temp (hasta la entrada (Kj, Prj)); n.Psig ← nuevo;
              nuevo ← entradas restantes en temp; K ← Kj;
              (∗ahora debemos mover (K, nuevo) e insertar en el nodo padre
              interno; sin embargo, si el padre está lleno, la división se puede propagar∗)
              finalizado ← falso;
              repeat
              if la pila S está vacía
                then (∗no hay nodo padre; se crea un nuevo nodo raíz para el árbol∗)
                   begin
                   root ← un nuevo nodo interno vacío para el árbol;
                   root ← <n, K, nuevo>; finalizado ← verdad;
                   end
                else begin
                   n ← pop pila S;
                   if nodo interno n no está lleno
                      then
                         begin (∗el nodo padre no está lleno; no hay división∗)
                         insertar (K, nuevo) en la posición correcta en el nodo interno n;
                         finalizado ← verdad
                         end
                      else begin (∗el nodo interno n está lleno con p punteros de árbol; hay división∗)
                         copiar n en temp (∗temp es un nodo interno bastante grande∗);
                         insertar (K, nuevo) en temp en la posición correcta;
                         (∗temp tiene ahora p +1 punteros de árbol∗)
```

```
                    nuevo ← un nuevo nodo interno vacío para el árbol;
                    j ← ⌈((p + 1)/2⌉;
                    n ← entradas hasta el puntero de árbol P_j en temp;
                    (∗n contiene <P_1, K_1, P_2, K_2,..., P_{j–1}, K_{j–1}, P_j >∗)
                    nuevo ← entradas del puntero de árbol P_{j+1} en temp;
                    (∗nuevo contiene < P_{j+1}, K_{j+1},..., K_{p–1}, P_p, K_p, P_{p+1} > ∗)
                    K ← Kj
                    (∗ahora debemos mover (K, nuevo) e insertar en el nodo interno
                    padre∗)
                    end
                end
            until finalizado
            end;
        end;
```

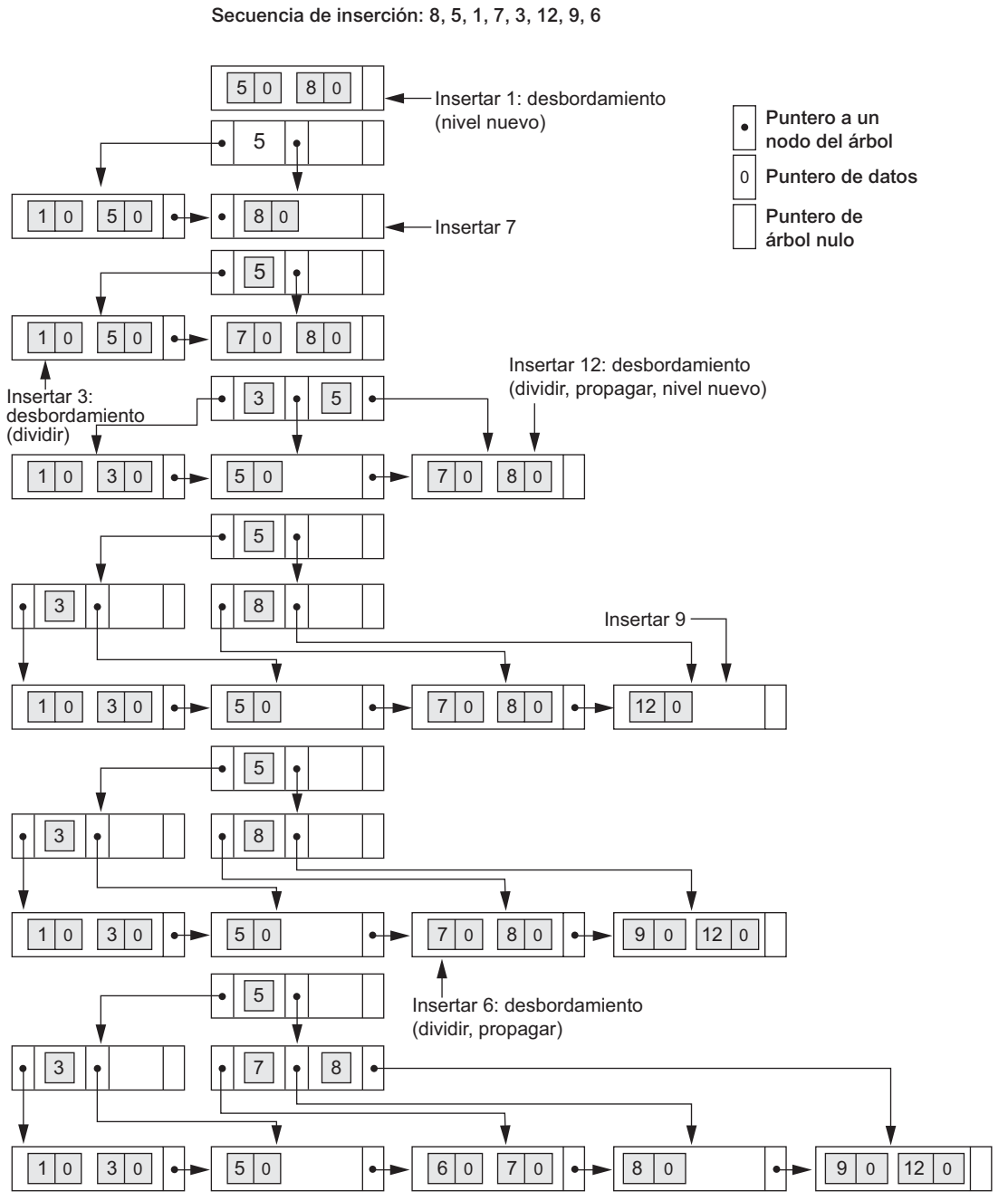
La Figura 14.12 ilustra la inserción de registros en un árbol $B^+$ de orden $p = 3$ y $p_{\text{hoja}} = 2$. En primer lugar, observamos que la raíz es el único nodo del árbol, por lo que también es un nodo hoja. En cuanto se crea más de un nivel, el árbol se divide en nodos internos y nodos hoja. Observe que *cada valor clave debe existir en el nivel hoja*, porque todos los punteros de datos están en el nivel hoja. Sin embargo, en los nodos internos sólo existen algunos valores para guiar la búsqueda. Además, cada valor que aparece en un nodo interno también aparece como *el valor más a la derecha* en el nivel hoja del subárbol apuntado por el puntero de árbol a la izquierda del valor.

Cuando un *nodo hoja* está lleno y se inserta en él una entrada nueva, el nodo se desborda y debe dividirse. Las primeras $j = \lceil((p_{\text{hoja}} + 1)/2)\rceil$ entradas del nodo original se mantienen, mientras que el resto se mueve a un nuevo nodo hoja. El valor de búsqueda en la posición $j$ se duplica en el nodo interno padre, y en éste se crea un puntero extra al nodo nuevo. Deben insertarse en el nodo padre en su secuencia correcta. Si el nodo interno padre está lleno, el valor nuevo también provocará su desbordamiento, por lo que deberá dividirse. Las entradas del nodo interno hasta $P_j$ (el puntero de árbol número $j$ después de insertar el valor y el puntero nuevos, donde $j = \lfloor((p + 1)/2)\rfloor$) se mantienen, mientras que el valor de búsqueda número $j$ se mueve al padre, no se duplica. Un nuevo nodo interno albergará las entradas desde $P_{j+1}$ hasta el final de las entradas del nodo (consulte el Algoritmo 14.3). Esta división puede propagarse hasta crear un nuevo nodo raíz y, por tanto, un nuevo nivel para el árbol $B^+$.

La Figura 14.13 ilustra la eliminación en un árbol $B^+$. Cuando se borra una entrada, siempre se elimina del nivel hoja. Si ocurre en un nodo interno, también debe eliminarse de allí. En este último caso, el valor a su izquierda en el nodo hoja debe sustituirse en el nodo interno porque este valor es ahora la entrada más a la derecha en el subárbol. La eliminación puede provocar un **desbordamiento por abajo**, al reducirse el número de entradas del nodo hoja por debajo del mínimo requerido. En este caso, intentamos encontrar un nodo hoja hermano (un nodo hoja situado inmediatamente a la izquierda o a la derecha del nodo desbordado por debajo) y redistribuir las entradas entre el nodo y su **hermano**, de modo que ambos estén rellenos, como mínimo, a la mitad; en caso contrario, el nodo se combina con sus hermanos y se reduce el número de nodos hoja. Un método común es intentar **redistribuir** las entradas con el hermano de la izquierda; si no es posible, se realiza un intento de redistribución con el hermano de la derecha. Si tampoco esto es posible, los tres nodos se combinan en dos nodos hoja. En tal caso, el desbordamiento por debajo se puede propagar a los nodos **internos** porque se necesitan un valor de búsqueda y un puntero de árbol menos. Esto puede propagarse y reducir los niveles del árbol.

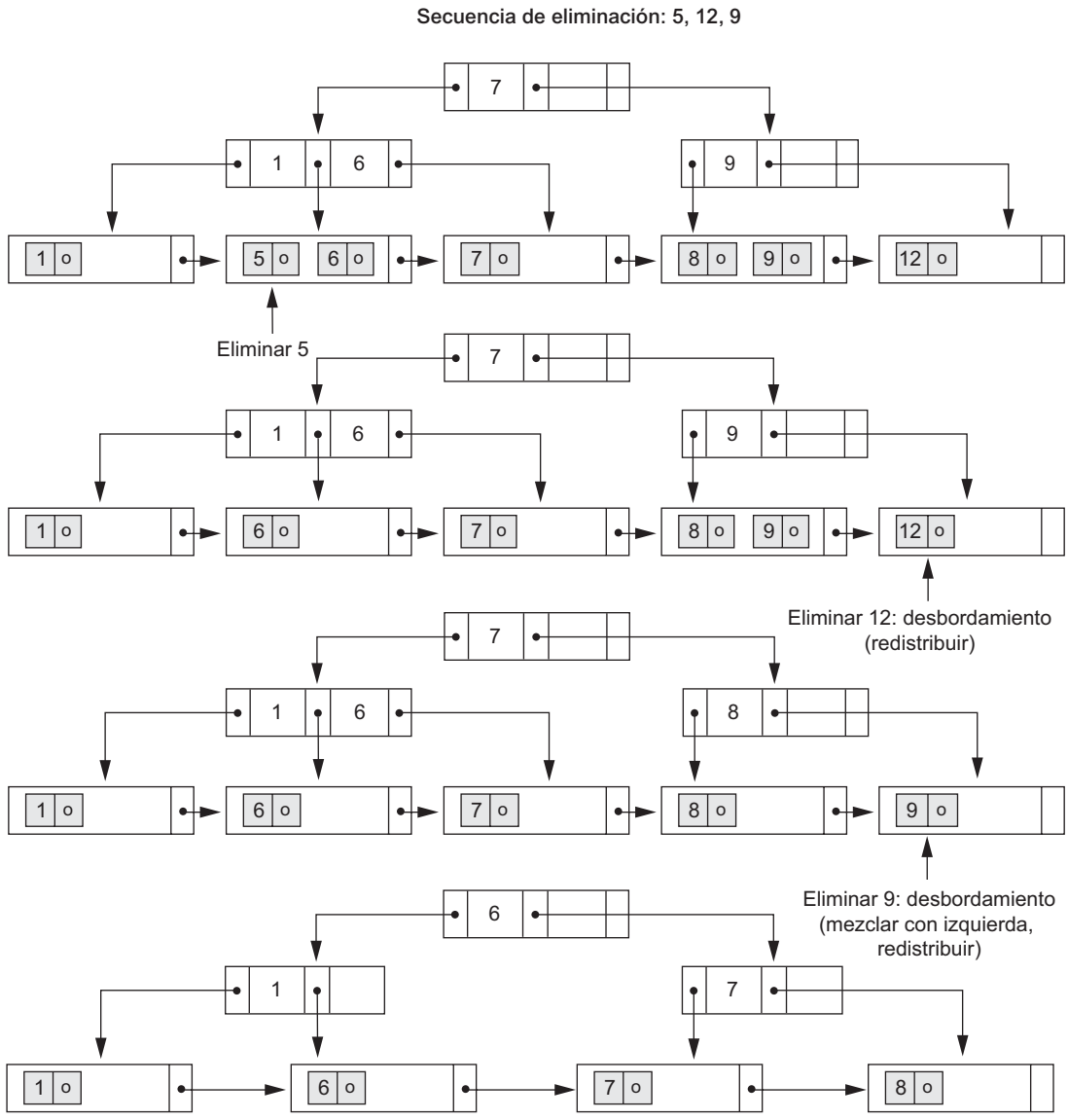
La implementación de los algoritmos de inserción y eliminación puede requerir punteros padre y hermano para cada nodo, o el uso de una pila como la del Algoritmo 14.3. Cada nodo también debe incluir el número de entradas que tiene y su tipo (hoja o interno). Otra alternativa es implementar la inserción y la eliminación como procedimientos recursivos.

**Figura 14.12.** Ejemplo de inserción en un árbol $B^+$ con $p = 3$ y $p_{hoja} = 2$.




**Variaciones de los árboles B y B[1].** Para concluir esta sección, mencionaremos brevemente algunas variantes de los árboles B y $B^+$. En algunos casos, la restricción 5 del árbol B (o del árbol $B^+$), que requiere que cada nodo esté lleno como mínimo hasta la mitad, puede modificarse para que cada nodo esté lleno a sus dos tercios. En este caso, el árbol B se denomina **árbol B∗**. En general, algunos sistemas permiten al usuario elegir un **factor de relleno** entre 0,5 y 1,0, donde esto último significa que los nodos del árbol B (índice) están completamente llenos. También es posible especificar dos factores de relleno para un árbol $B^+$: uno para el

**Figura 14.13.** Ejemplo de eliminación en un árbol $B^+$.




nivel hoja y otro para los nodos internos del árbol. Cuando se construye el índice al principio, los nodos se rellenan hasta aproximadamente los factores de relleno especificados. Recientemente, los investigadores han sugerido relajar el requisito de que un nodo esté lleno hasta la mitad, y en su lugar permiten que un nodo se quede completamente vacío antes de combinarse, simplificándose así el algoritmo de eliminación. Los estudios de simulación muestran que esto no desperdicia demasiado espacio adicional bajo inserciones y eliminaciones aleatoriamente distribuidas.

# 14.4 Índices en claves múltiples

Hasta ahora, en nuestra explicación hemos asumido que las claves principal y secundaria con las que se accede a los ficheros son atributos sencillos (campos). En muchas solicitudes de recuperación y actualización, son

varios los atributos implicados. Si se utiliza con frecuencia una cierta combinación de atributos, es recomendable configurar una estructura de acceso para ofrecer un acceso eficaz por medio de un valor clave que sea una combinación de esos atributos.

Por ejemplo, piense en un fichero EMPLEADO que contiene los atributos Dno (número de departamento), Edad, Calle, Ciudad, CodPostal, Sueldo y CodExperiencia, con Dni (documento nacional de identidad) como clave. Considere esta consulta: *Listar los empleados del departamento número 4 cuya edad sea 59*. Ni Dno ni Edad son atributos clave, lo que significa que un valor de búsqueda para cualquiera de ellos apuntará a varios registros. Podemos considerar las siguientes estrategia de búsqueda alternativas:

1. Asumiendo que Dno tiene un índice, pero no así Edad, acceder a los registros con Dno = 4 utilizando el índice y después seleccionar de entre ellos aquellos registros que satisfacen la condición Edad = 59.
2. Como alternativa, si Edad está indexado pero no así Dno, acceder a los registros con Edad = 59 utilizando el índice y después seleccionar de entre ellos aquellos registros que satisfacen la condición Dno = 4.
3. Si se han creado índices tanto en Dno como en Edad, se pueden utilizar los dos; cada uno ofrece un conjunto de registros o un conjunto de punteros (a bloques o registros). Una intersección de estos conjuntos de registros o punteros proporciona los registros que satisfacen las dos condiciones, los punteros de registro que satisfacen las dos condiciones, o los bloques en los que se encuentran los registros que satisfacen las dos condiciones.

Todas estas alternativas proporcionan al final el resultado correcto. No obstante, si el conjunto de registros que satisface cada condición por separado (Dno = 4 o Edad = 59) es grande, y sólo unos cuantos registros satisfacen la condición combinada, entonces ninguna de las anteriores es una técnica eficaz para la solicitud de búsqueda dada. Existen varias posibilidades que tratarían la combinación <Dno, Edad> o <Edad, Dno> como una clave de búsqueda compuesta por varios atributos. Perfilaremos brevemente estas técnicas a continuación. Nos referiremos a las claves que contienen varios atributos como **claves compuestas**.

### 14.4.1 Índice ordenado por varios atributos

Todo lo explicado hasta ahora todavía es aplicable si creamos un índice en un campo clave de búsqueda que es una combinación de <Dno, Edad>. En el ejemplo anterior, la clave de búsqueda es un par de valores, <4, 59>. En general, si creamos un índice sobre los atributos $<A_1, A_2, ..., A_n>$, los valores de la clave de búsqueda son tuplas con $n$ valores: $<v_1, v_2, ..., v_n>$.

Una ordenación lexicográfica de los valores de esta tupla establece un orden en esta clave de búsqueda compuesta. Para nuestro ejemplo, todas las claves de departamento para el departamento número 3 preceden a las del departamento número 4. Por tanto, <3, $n$> precede a <4, $m$> para cualquier valor de $m$ y $n$. El orden de clave ascendente para las claves con Dno = 4 sería <4, 18>, <4, 19>, <4, 20>, etcétera. La ordenación lexicográfica funciona de forma parecida para ordenar las cadenas de caracteres. Un índice en una clave compuesta por $n$ atributos funciona de forma parecida a cualquier índice de los explicados hasta ahora en este capítulo.

### 14.4.2 Dispersión partida

La dispersión partida (o subdividida) es una extensión de la dispersión externa estática (Sección 13.8.2) que permite el acceso sobre varias claves. Sólo resulta adecuada para comparaciones de igualdad; las consultas de rango no están soportadas. En la dispersión partida, para una clave compuesta por $n$ componentes, la función de dispersión está diseñada para producir un resultado con $n$ direcciones de dispersión separadas. La dirección de cubo es una concatenación de estas $n$ direcciones. Entonces es posible buscar por la clave de búsqueda compuesta requerida buscando los cubos adecuados que coinciden con las partes de la dirección en la que estamos interesados.

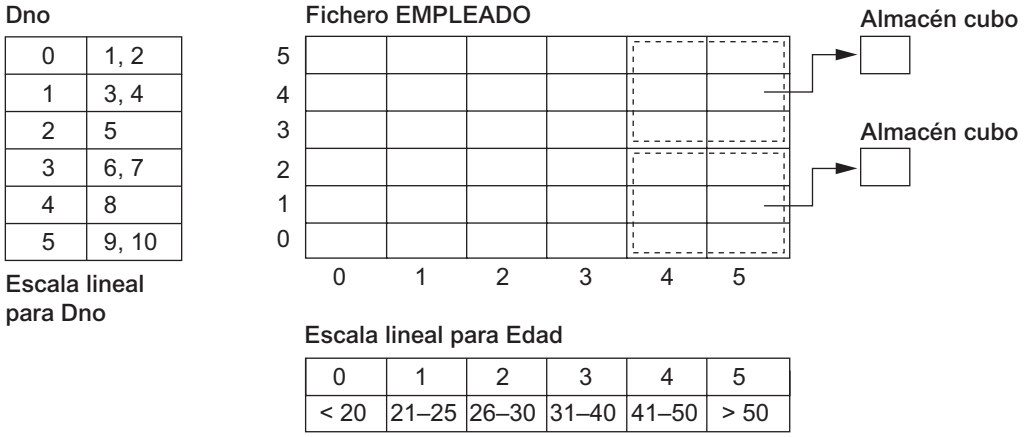
Por ejemplo, considere la clave de búsqueda compuesta <Dno, Edad>. Si Dno y Edad están dispersas en una dirección de 3 bits y 5 bits respectivamente, obtenemos una dirección de cubo de 8 bits. Suponga que Dno = 4 tiene una dirección de dispersión '100' y que Edad = 59 tiene una dirección de dispersión '10101'. Entonces, para buscar el valor de búsqueda combinado, Dno = 4 y Edad = 59, acudimos a la dirección de cubo 100 10101; para buscar todos los empleados con una Edad = 59, se buscarán todos los cubos (ocho de ellos) cuyas direcciones son '000 10101', '001 10101', ..., etcétera. Una ventaja de la dispersión partida es que puede extenderse fácilmente a cualquier número de atributos. Las direcciones de cubo pueden diseñarse para que los bits de orden superior de las direcciones correspondan a los atributos a los que se accede con más frecuencia. Además, no es necesario mantener estructuras de acceso separadas para los atributos individuales. El principal inconveniente de la dispersión partida es que no puede manipular consultas de rango sobre cualquiera de los atributos componente.

### 14.4.3 Ficheros rejilla

Otra alternativa es reorganizar el fichero EMPLEADO como un fichero rejilla. Si queremos acceder a un fichero con dos claves, como Dno y Edad en nuestro ejemplo, podemos construir un array rejilla con una escala (o dimensión) lineal por cada uno de los atributos de búsqueda. La Figura 14.14 muestra un array de rejilla para el fichero EMPLEADO con una escala lineal para Dno y otra para el atributo Edad. Las escalas se hacen en cierto modo para lograr una distribución uniforme de ese atributo. Así, en nuestro ejemplo, vemos que la escala lineal para Dno tiene Dno = 1, 2 combinados como un valor 0 en la escala, mientras que Dno = 5 corresponde al valor 2 en esa escala. De forma parecida, Edad se divide en su escala de 0 a 5 agrupando edades a fin de distribuir los empleados uniformemente por edad. El array rejilla mostrado para este fichero tiene un total de 36 celdas. Cada celda apunta a alguna dirección de cubo donde se almacenan los registros correspondientes a esa celda. La Figura 14.14 también muestra la asignación de celdas a cubos (sólo parcialmente).

Por tanto, nuestra solicitud de Dno = 4 y Edad = 59 se mapea dentro de la celda (1, 5) correspondiente al array rejilla. Los registros para esta combinación se encontrarán en el cubo correspondiente. Este método es particularmente útil para consultas de rango que se mapearían en un conjunto de celdas correspondiente a un grupo de valores a lo largo de las escalas lineales. Conceptualmente, el concepto de fichero rejilla puede aplicarse a cualquier número de claves de búsqueda. Para *n* claves de búsqueda, el array rejilla tendría *n* dimensiones. El array rejilla permite así una partición del fichero a lo largo de las dimensiones de los atributos clave de búsqueda y proporciona un acceso mediante combinaciones de valores a lo largo de esas dimensiones. Los ficheros rejilla funcionan bien al reducir el tiempo de acceso por clave múltiple. Sin embargo, representan un

**Figura 14.14.** Ejemplo de un array rejilla sobre los atributos Dno y Edad.



gasto de espacio debido a la estructura en rejilla. Además, con los ficheros dinámicos, se añade una reorganización frecuente del fichero al coste de mantenimiento.[10]

# 14.5 Otros tipos de índices

## 14.5.1 Uso de la dispersión y otras estructuras de datos como índices

También es posible crear estructuras de acceso parecidas a índices basándose en la *dispersión*. Las entradas de índice <*K*, *Pr*> (o <*K*, *P*>) pueden organizarse como un fichero de dispersión ampliable dinámicamente, utilizando una de las técnicas explicadas en la Sección 13.8.3; la búsqueda de una entrada utiliza el algoritmo de búsqueda dispersa sobre *K*. Una vez encontrada una entrada, se utiliza el puntero *Pr* (o *P*) para encontrar el registro correspondiente en el fichero de datos. También es posible utilizar otras estructuras de búsqueda como índices.

## 14.5.2 Índices lógicos frente a índices físicos

Hasta ahora, hemos asumido que las entradas de índice <*K*, *Pr*> (o <*K*, *P*>) siempre incluyen un puntero físico *Pr* (o *P*) que especifica la dirección física del registro en el disco, a modo de número de bloque y desplazamiento. Es lo que a veces se denomina índice físico, y tiene el inconveniente de que el puntero debe modificarse si el registro se mueve a otra ubicación dentro del disco. Por ejemplo, suponga que la organización principal de un fichero está basada en la dispersión lineal o extensible; entonces, cada vez que un cubo se divide, algunos registros son asignados a cubos nuevos y, por tanto, cambian sus direcciones físicas. Si había un índice secundario en el fichero, tendrían que buscarse y actualizarse los punteros a esos registros, que es una tarea compleja.

Para remediar esta situación, podemos utilizar una estructura denominada índice lógico, cuyas entradas tienen la forma <*K*, *Kp*>. Cada entrada tiene un valor *K* para el campo de indexación secundario coincidente con el valor *Kp* del campo utilizado para la organización primaria del fichero. Buscando en el índice secundario con el valor de *K*, un programa puede localizar el valor correspondiente de *Kp* y utilizarlo para acceder al registro a través de la organización primaria del fichero. Los índices lógicos introducen así un nivel adicional de indirección entre la estructura de acceso y los datos. Se utilizan cuando se espera que las direcciones físicas de los registros cambien con frecuencia. El coste de esta indirección es una búsqueda extra basada en la organización primaria del fichero.

## 14.5.3 Explicación

En muchos sistemas, un índice no es parte integral del fichero de datos, pero puede crearse y descartarse dinámicamente. Es por lo que a menudo se le denomina *estructura de acceso*. Siempre que esperemos acceder con frecuencia a un fichero basándonos en alguna condición de búsqueda que implique un campo en particular, podemos solicitar al DBMS que cree un índice con ese campo. Normalmente, se crea un índice secundario para evitar la ordenación física de los registros en el fichero de datos en disco.

La principal ventaja de los índices secundarios es que (al menos teóricamente) pueden crearse en combinación con *prácticamente cualquier organización primaria de registros*. Por tanto, un índice secundario podría utilizarse para complementar otros métodos de acceso principales, como la ordenación o la dispersión, o incluso podría utilizarse con ficheros mixtos. Para crear un índice secundario de árbol $B^+$ en algún campo de un

[10] Los algoritmos de inserción/eliminación para los ficheros rejilla pueden encontrarse en Nievergelt (1984).

fichero, debemos pasar por todos los registros del fichero para crear las entradas en el nivel hoja del árbol. Estas entradas se ordenan y rellenan entonces de acuerdo con el factor de relleno especificado; a la vez, se crean los otros niveles del índice. Es más costoso y mucho más duro crear índices primarios o principales e índices agrupados dinámicamente, porque los registros del fichero de datos deben ordenarse físicamente en el disco según el orden del campo de indexación. No obstante, algunos sistemas permiten a los usuarios crear dinámicamente esos índices en sus ficheros ordenando el fichero durante la creación del índice.

Es común utilizar un índice para implementar una *restricción de clave* en un atributo. Mientras se busca en el índice para insertar un nuevo registro, es fácil comprobar al mismo tiempo si otro registro del fichero (y, por tanto, del árbol del índice) tiene el mismo atributo clave que el registro nuevo. Si es así, la inserción puede rechazarse.

Un fichero que tiene un índice secundario en cada uno de sus campos se denomina a veces **fichero totalmente invertido**. Como todos los índices son secundarios, los registros nuevos se insertan al final del fichero; por consiguiente, el propio fichero de datos es un fichero desordenado (*heap*). Los índices normalmente se implementan como árboles $B^+$, por lo que se actualizan dinámicamente para reflejar la inserción y la eliminación de registros. Algunos DBMSs comerciales, como ADABAS de Software-AG, utilizan mucho este método.

En la Sección 14.2 nos referimos a la popular organización de fichero de IBM denominada ISAM. Otro método de IBM, **VSAM** (**Método de acceso de almacenamiento virtual**, *Virtual Storage Access Method*), es algo parecido a la estructura de acceso de árbol $B^+$.

# 14.6 Resumen

En este capítulo presentamos las organizaciones de ficheros que implican estructuras de acceso adicionales, denominadas índices, para mejorar la eficacia de la recuperación de los registros de un fichero de datos. Estas estructuras de acceso pueden utilizarse *en combinación con* las organizaciones de fichero primarias explicadas en el Capítulo 13, que se utilizan para organizar los registros del fichero en el disco.

Hemos visto tres tipos de índices ordenados de un solo nivel: primario o principal, agrupado y secundario. Cada índice se especifica en un campo del fichero. Los índices primario y agrupado se construyen sobre el campo de ordenación física de un fichero, mientras que los índices secundarios se especifican sobre los campos desordenados. El campo para un índice principal también debe ser una clave del fichero, mientras que es un campo no clave para un índice agrupado. Un índice de un solo nivel es un fichero ordenado y se explora utilizando una búsqueda binaria. Hemos visto la construcción de índices multinivel para mejorar la eficacia de la búsqueda en un índice.

A continuación, vimos cómo pueden implementarse índices multinivel como árboles B y $B^+$, que son estructuras dinámicas que permiten que un índice se expanda o se reduzca dinámicamente. Los nodos (bloques) de estas estructuras de índice se mantienen rellenos entre el 50% y el 100% de su capacidad gracias a los algoritmos de inserción y eliminación. Los nodos se estabilizan finalmente y por término medio cuando su ocupación ronda el 69%, lo que ofrece espacio para realizar inserciones sin tener que reorganizar el índice en la mayoría de los casos. Los árboles $B^+$ pueden mantener generalmente más entradas en sus nodos internos que los árboles B, por lo que pueden tener menos niveles o albergar menos entradas que el árbol B equivalente.

También hemos ofrecido una visión general de los métodos de acceso por clave múltiple, así como de la construcción de un índice basándonos en estructuras de datos dispersos. Después ofrecimos la introducción del concepto de índice lógico, y lo comparamos con los índices físicos que describimos anteriormente. Por último, explicamos cómo podemos utilizar combinaciones de estas organizaciones. Por ejemplo, los índices secundarios se utilizan con frecuencia con índices mixtos, así como con ficheros desordenados y ordenados. También es posible crear índices secundarios para los ficheros dispersos y los ficheros dispersos dinámicos.

## Preguntas de repaso

**14.1.** Defina los siguientes términos: campo de indexación, campo clave principal, campo agrupado, campo clave secundario, ancla de bloque, índice denso e índice no denso (escaso).

**14.2.** ¿Cuáles son las diferencias entre índice principal, secundario y agrupado? ¿Cómo afectan estas diferencias a la forma en que se implementan estos índices? ¿Cuáles son índices densos, y cuáles no?

**14.3.** ¿Por qué en un fichero podemos tener a lo sumo un índice principal o agrupado, pero varios índices secundarios?

**14.4.** ¿Cómo mejora la eficacia de búsqueda de un fichero de índice la indexación multinivel?

**14.5.** ¿Cuál es el orden p de un árbol B? Describa la estructura de los nodos de un árbol B.

**14.6.** ¿Cuál es el orden p de un árbol $B^+$? Describa la estructura de los nodos internos y hoja de un árbol $B^+$.

**14.7.** ¿Cómo difiere un árbol B de un árbol $B^+$? ¿Por qué se prefiere normalmente un árbol $B^+$ como estructura para acceder a un fichero de datos?

**14.8.** Explique las alternativas que existen para acceder a un fichero basándose en claves de búsqueda múltiples.

**14.9.** ¿Qué es la dispersión partida? ¿Cómo funciona? ¿Cuáles son sus limitaciones?

**14.10.** ¿Qué es un fichero rejilla? ¿Cuáles son sus ventajas y sus inconvenientes?

**14.11.** Muestre un ejemplo de construcción de un array rejilla sobre dos atributos de algún fichero.

**14.12.** ¿Qué es un fichero totalmente invertido? ¿Qué es un fichero secuencial indexado?

**14.13.** ¿Cómo puede utilizarse la dispersión para construir un índice? ¿Cuál es la diferencia entre un índice lógico y un índice físico?

## Ejercicios

**14.14.** Considere un disco con un tamaño de bloque de $B = 512$ bytes. Un puntero de bloque tiene una longitud de $P = 6$ bytes, y un puntero de registro tiene una longitud de $P_R = 7$ bytes. Un fichero tiene $r = 30.000$ registros EMPLEADO de *longitud fija*. Cada registro tiene los siguientes campos: Nombre (30 bytes), Dni (9 bytes), CodDpto (9 bytes), Direcc (40 bytes), Tlf (9 bytes), FechaNac (8 bytes), Sexo (1 byte), CodTrabajo (4 bytes), Sueldo (4 bytes, número real). Se utiliza un byte adicional como marcador de eliminación.

- **a.** Calcule el tamaño de registro $R$ en bytes.
- **b.** Calcule el factor de bloqueo *bfr* y el número de bloques de fichero $b$, asumiendo una organización no extendida.
- **c.** Suponga que el fichero está *ordenado* por el campo clave Dni y que queremos construir un *índice principal* sobre Dni. Calcule (i) el factor de bloqueo de índice $bfr_i$ (que también es el índice *fan-out fo*); (ii) el número de entradas de índice de primer nivel y el número de bloques de índice de primer nivel; (iii) el número de niveles necesarios si lo convertimos en un índice multinivel; (iv) el número total de bloques que el índice multinivel necesita; y (v) el número de accesos a bloques que se necesitan para buscar y recuperar un registro del fichero (dado el valor de su Dni) utilizando el índice principal.
- **d.** Suponga que el fichero *no está ordenado* por el campo clave Dni y queremos construir un *índice secundario* sobre Dni. Repita el ejercicio anterior (parte c) para el índice secundario y compárelo con el índice principal.
- **e.** Suponga que el fichero *no está ordenado* por el campo no clave CodDpto y queremos construir un *índice secundario* sobre CodDpto, utilizando la opción 3 de la Sección 14.1.3, con un nivel

extra de indirección que almacena punteros de registro. Asuma que hay 1.000 valores distintos de CodDpto y que los registros EMPLEADO están uniformemente distribuidos entre estos valores. Calcule (i) el factor de bloqueo de índice $bfr_i$ (que también es el índice *fan-out fo*); (ii) el número de bloques que el nivel de indirección que almacena los punteros de registro necesita; (iii) el número de entradas de índice de primer nivel y el número de bloques de índice de primer nivel; (iv) el número de niveles necesarios si lo convertimos en un índice multinivel; (v) el número total de bloques que el índice multinivel necesita y los bloques utilizados en el nivel extra de indirección; y (vi) el número aproximado de accesos a bloques necesarios para buscar y recuperar todos los registros del fichero que tienen un valor CodDpto determinado, utilizando el índice.

**f.** Suponga que el fichero está *ordenado* por el campo no clave CodDpto y queremos construir un *índice agrupado* sobre CodDpto que utiliza anclas de bloque (cada valor nuevo de CodDpto empieza al principio de un bloque nuevo). Asuma que hay 1.000 valores distintos de CodDpto y que los registros EMPLEADO están uniformemente distribuidos entre estos valores. Calcule (i) el factor de bloqueo de índice $bfr_i$ (que también es el índice *fan-out fo*); (ii) el número de entradas de índice de primer nivel y el número total de bloques de índice de primer nivel; (iii) el número de niveles necesarios si lo convertimos en un índice multinivel; (iv) el número total de bloques que el índice multinivel necesita; y (v) el número de accesos a bloques que se necesitan para buscar y recuperar todos los registros del fichero que tienen un valor concreto de CodDpto, utilizando el índice agrupado (asuma que varios bloques en un grupo son contiguos).

**g.** Suponga que el fichero *no está ordenado* por el campo clave Dni y que queremos construir una estructura de acceso en árbol $B^+$ (índice) sobre Dni. Calcule (i) los órdenes $p$ y $p_{\text{hoja}}$ del árbol $B^+$; (ii) el número de bloques a nivel hoja necesarios si los bloques están llenos aproximadamente al 69% (redondeado por arriba por conveniencia); (iii) el número de niveles necesarios si los nodos internos están llenos al 69% (redondeado por arriba por conveniencia); (iv) el número total de bloques que el árbol $B^+$ requiere; y (v) el número de accesos a bloques necesarios para buscar y recuperar un registro del fichero (dado su valor de Dni) utilizando el árbol $B^+$.

**h.** Repita la parte g, pero para un árbol B en lugar de para un árbol $B^+$. Compare los resultados obtenidos con ambos tipos de árbol.

**14.15.** Un fichero PIEZAS con NumPieza como campo clave incluye registros con los siguientes valores de NumPieza: 23, 65, 37, 60, 46, 92, 48, 71, 56, 59, 18, 21, 10, 74, 78, 15, 16, 20, 24, 28, 39, 43, 47, 50, 69, 75, 8, 49, 33, 38. Suponga que los valores del campo de búsqueda se insertan en el orden dado en un árbol $B^+$ de orden $p = 4$ y $p_{\text{hoja}} = 3$; muestre la expansión del árbol y cuál será su aspecto final.

**14.16.** Repita el Ejercicio 14.15, pero utilice un árbol B de orden $p = 4$ en lugar de un árbol $B^+$.

**14.17.** Suponga que se borran los siguientes valores del campo de búsqueda, en el orden dado, del árbol $B^+$ del Ejercicio 14.15; muestre cómo se reducirá el árbol, así como el árbol final. Los valores borrados son 65, 75, 43, 18, 20, 92, 59, 37.

**14.18.** Repita el Ejercicio 14.17, pero para el árbol B del Ejercicio 14.16.

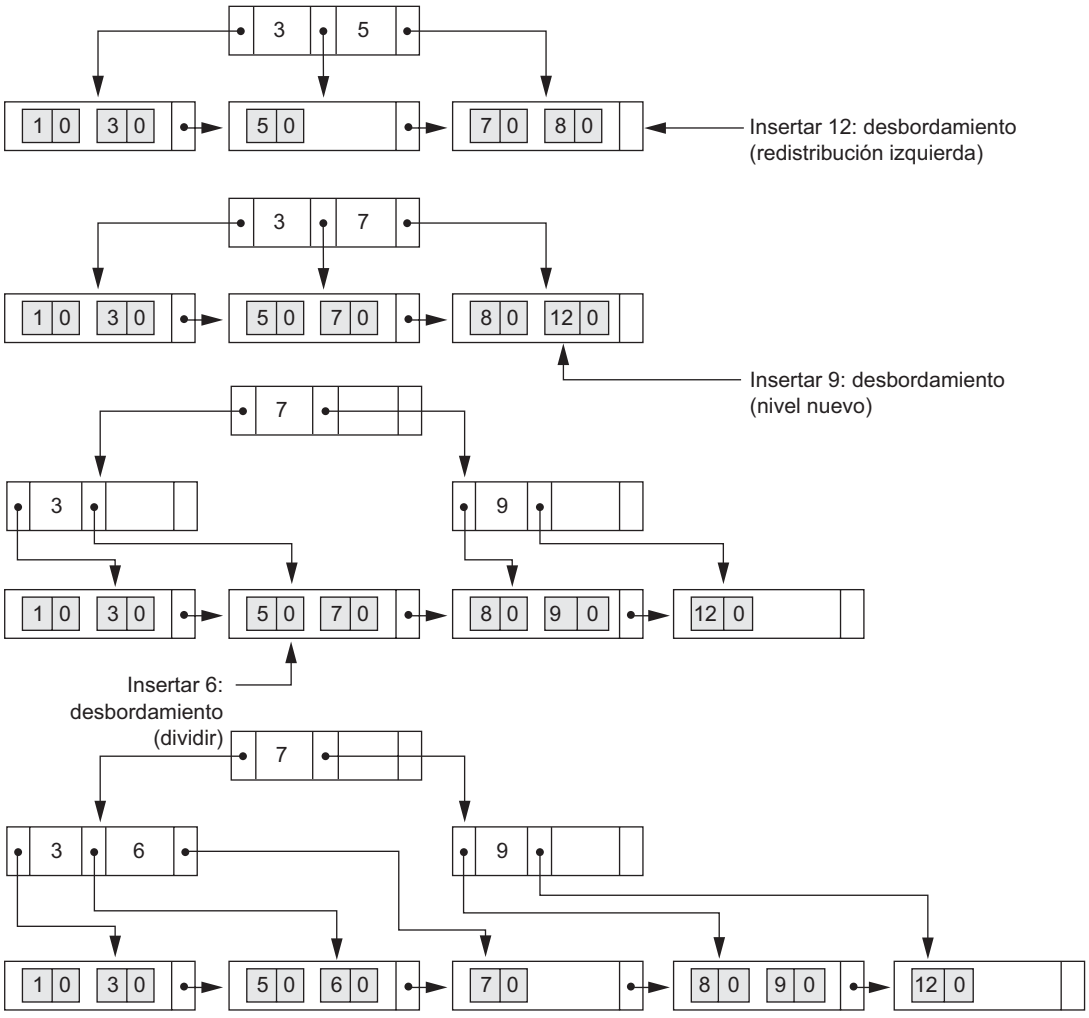
**14.19.** El Algoritmo 14.1 esboza el procedimiento de búsqueda en un índice principal multinivel no denso para recuperar un registro del fichero. Adapte el algoritmo para cada uno de los siguientes casos:

**a.** Un índice secundario multinivel sobre un campo no ordenado y no clave de un fichero. Asuma que se utiliza la opción 3 de la Sección 14.1.3, donde un nivel de indirección extra almacena punteros a los registros individuales con el correspondiente valor del campo índice.

**b.** Un índice secundario multinivel sobre un campo clave no ordenado de un fichero.

**c.** Un índice agrupado multinivel sobre un campo de ordenación no clave de un fichero.

**Figura 14.15.** Inserción en un árbol $B^+$ con redistribución izquierda.




**14.20.** Suponga que existen varios índices secundarios en campos no clave de un fichero, implementados con la opción 3 de la Sección 14.1.3; por ejemplo, podríamos tener índices secundarios en los campos CodDpto, CodTrabajo y Sueldo del fichero EMPLEADO del Ejercicio 14.14. Describa una forma eficaz de buscar y recuperar los registros que satisfagan una condición de selección compleja sobre estos campos, como, por ejemplo, (CodDpto = 5 AND CodTrabajo = 12 AND Sueldo = 50.000), utilizando los punteros de registro en el nivel de indirección.

**14.21.** Adapte para un árbol B los Algoritmos 14.2 y 14.3, que esbozan los procedimientos de búsqueda e inserción en un árbol $B^+$.

**14.22.** Es posible modificar el algoritmo de inserción de un árbol $B^+$ para retardar el caso de que tenga que producirse un nivel nuevo verificando una posible *redistribución* de valores a lo largo de los nodos hoja. La Figura 14.15 ilustra cómo se puede hacer esto para nuestro ejemplo de la Figura 14.12; en lugar de dividir el nodo hoja situado más a la izquierda cuando se inserta 12, hacemos una *redistribución a la izquierda* moviendo el 7 al nodo hoja de su izquierda (si hay espacio en este nodo). La Figura 14.15 muestra el aspecto que debe tener el árbol cuando se considera la redistri-

bución. También es posible considerar la *redistribución derecha*. Intente modificar el algoritmo de inserción para un árbol $B^+$ para tener en cuenta la redistribución.

**14.23.** Diseñe un algoritmo para la eliminación en un árbol $B^+$.

**14.24.** Repita el Ejercicio 14.23 para un árbol B.

## Bibliografía seleccionada

Bayer y McCreight (1972) ofrece una introducción a los árboles B y los algoritmos asociados. Comer (1979) proporciona un excelente estudio de los árboles B y su historia, así como de sus variantes. Knuth (1973) proporciona un análisis detallado de muchas técnicas de búsqueda, incluyendo los árboles B y algunas de sus variantes. Nievergelt (1974) explica el uso de los árboles de búsqueda binaria para la organización de los ficheros. Libros que hablan de las estructuras de los ficheros, como Wirth (1972), Claybrook (1983), Smith y Barnes (1987), Miller (1987) y Salzberg (1988), explican la indexación en detalle y en ellos pueden consultarse los algoritmos de búsqueda, inserción y eliminación para árboles B y $B^+$. Larson (1981) analiza los ficheros secuenciales indexados, y Held y Stonebraker (1978) compara los índices multinivel estáticos con los índices dinámicos de árboles B. Lehman y Yao (1981) y Srinivasan y Carey (1991) realizan un análisis más profundo del acceso concurrente a los árboles B. Los libros Wiederhold (1983), Smith y Barnes (1987) y Salzberg (1988), entre otros, explican muchas de las técnicas de búsqueda descritas en este capítulo. Los ficheros rejilla se introducen en Nievergelt (1984). La recuperación por coincidencia parcial, que hace uso de la dispersión partida, se explica en Burkhard (1976, 1979).

Las nuevas técnicas y aplicaciones de los índices y los árboles $B^+$ se explican en Lanka y Mays (1991), Zobel y otros (1992), y Faloutsos y Jagadish (1992). Mohan y Narang (1992) explican la creación de índices. El rendimiento de diversos algoritmos de árboles B y $B^+$ se evalúa en Baeza-Yates y Larson (1989) y Johnson y Shasha (1993). La administración de los búferes de cara a los índices se explica en Chan y otros (1992).

CAPÍTULO 15

# Algoritmos para procesamiento y optimización de consultas

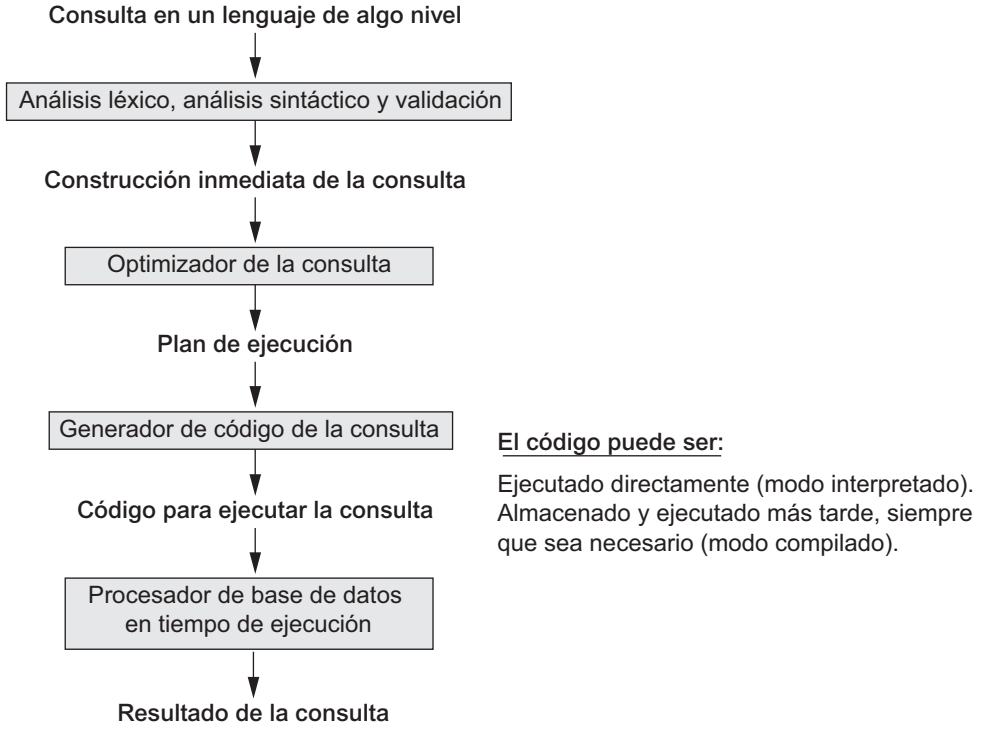
En este capítulo revisaremos las técnicas utilizadas por un DBMS para procesar, optimizar y ejecutar consultas de alto nivel. Una consulta expresada en un lenguaje de alto nivel como SQL debe, en una primera fase, ser explorada, analizada sintácticamente y validada. El **analizador léxico** identifica los elementos del lenguaje como, por ejemplo, las palabras reservadas de SQL, los nombres de atributos y los nombres de las relaciones, en el texto de la consulta, mientras que el **analizador sintáctico** comprueba la sintaxis de la consulta para determinar si ha sido formulada con arreglo a las reglas de sintaxis (las reglas de la gramática) del lenguaje de consultas. La consulta debe ser también **validada**, comprobando que todos los nombres de atributos y de relaciones son válidos y tienen significado semántico dentro del esquema de la base de datos en particular sobre la cual se realiza la consulta.[1] A continuación, se creará una representación interna de la consulta; normalmente mediante una estructura de datos en árbol denominada **árbol de consultas**. También es posible representar la consulta utilizando una estructura de datos en grafo denominada **grafo de consulta**. Seguidamente, el DBMS debe desarrollar una estrategia de ejecución para obtener el resultado de la consulta a partir de los ficheros de la base de datos. Lo habitual es que en una consulta se disponga de muchas estrategias distintas de ejecución; el proceso de elección de la estrategia más adecuada se conoce como **optimización de consultas**.

La Figura 15.1 muestra los distintos pasos a seguir en la ejecución de una consulta de alto nivel. El **optimizador de consultas** se encarga de generar un plan de ejecución y el **generador de código** se encarga de generar el código para ejecutar dicho plan. El **procesador de base de datos en tiempo de ejecución** tiene como cometido la ejecución del código, bien en modo compilado o bien en modo interpretado, para generar el resultado de la consulta. Si se produce un error en tiempo de ejecución el procesador de base de datos en tiempo de ejecución generará un mensaje de error.

El término optimización resulta, en realidad, inapropiado ya que en algunos casos el plan de ejecución elegido no resulta ser la estrategia más optima (se trata solo de una *estrategia razonablemente eficiente* para ejecutar la consulta). Por lo general, la búsqueda de la estrategia más optima consume demasiado tiempo excepto en el caso de las consultas más sencillas y es posible que se necesite información sobre cómo están construidos los ficheros e incluso sobre el contenido de los mismos (información que quizá no esté disponible por

[1] No discutiremos aquí las fases de análisis léxico y comprobación de sintaxis del procesamiento de una consulta, ya que estos temas se tratan en los textos relacionados con los compiladores.

**Figura 15.1.** Pasos habituales al procesar una consulta de alto nivel.




completo en el catálogo del DBMS). Por tanto, la *planificación de una estrategia de ejecución* puede llegar a requerir una descripción más detallada que en el caso de la *optimización de consultas*.

En el caso de los lenguajes de navegación por bases de datos de nivel más bajo (como, por ejemplo, DML en red o HDML jerárquico [consulte los Apéndices D y E]), el programador debe elegir la estrategia de ejecución de consultas a la hora de escribir un programa de bases de datos. Si un DBMS proporciona únicamente un lenguaje de navegación, las *necesidades u ocasiones de optimización* por parte del DBMS resultan ser bastante limitadas; en lugar de esto, al programador se le da la posibilidad de elegir la estrategia de ejecución *óptima*. Por otra parte, un lenguaje de consultas de alto nivel (como SQL para DBMS relacionales [RDBMS] u OQL [consulte el Capítulo 21] para DBMS orientadas a objetos [ODBMS]), es más expresivo por su propia naturaleza ya que especifica cuáles son los resultados esperados de la consulta en lugar de identificar los detalles sobre *cómo* se debería obtener el resultado. Según lo anterior, la optimización de consultas sí resulta necesaria en el caso de consultas especificadas en un lenguaje de alto nivel.

Nos concentraremos en describir la optimización de consultas en el contexto de un RDBMS, ya que muchas de las técnicas que describiremos han sido adaptadas para ODBMS.[2] Un DBMS relacional debe evaluar de forma sistemática las distintas estrategias de ejecución de consultas y elegir una estrategia razonablemente eficiente u óptima. Todos los DBMS disponen por lo general de varios algoritmos genéricos de acceso a bases de datos que implementan operaciones relacionales como SELECT o JOIN o combinaciones de ambas. Sólo las estrategias de ejecución que pueden ser implementadas por los algoritmos de acceso del DBMS y que se aplican a una consulta en particular y a un diseño de base de datos en particular son las que pueden ser tomadas en consideración por el módulo de optimización de consultas.

[2] Existen algunas técnicas y problemas de optimización de consultas que sólo pertenecen al ámbito de los ODBMS. Sin embargo, no los discutiremos aquí ya que sólo estamos ofreciendo una introducción a la optimización de consultas.

En la Sección 15.1 comenzaremos con una visión general sobre cómo se traducen habitualmente las consultas SQL a consultas de álgebra relacional para, posteriormente, ser optimizadas. A continuación, en las Secciones 15.2 a 15.6 veremos los algoritmos utilizados en la implementación de operaciones relacionales. Posteriormente, ofreceremos una visión general de las estrategias de optimización de consultas. Existen dos técnicas principales para la construcción de las optimizaciones de consultas. La primera técnica se basa en **reglas heurísticas** para la ordenación de las operaciones a realizar en una estrategia de ejecución de una consulta. Una regla heurística es una regla que funciona bien en la mayoría de los casos, aunque no está garantizado que funcione correctamente en todos los casos. Normalmente, las reglas reordenan las operaciones en un árbol de consultas. La segunda de las técnicas implica una **estimación sistemática** del coste de las diferentes estrategias de ejecución y la elección del plan de ejecución con la estimación de coste más baja. Normalmente, estas técnicas se combinan en un optimizador de consultas. Discutiremos la optimización heurística en la Sección 15.7 y la estimación de costes en la Sección 15.8. Posteriormente, en la Sección 15.9 haremos un breve repaso de los factores a tener en cuenta durante la optimización de consultas en el RDBMS comercial de Oracle. La Sección 15.10 presenta la idea de la optimización semántica de consultas, en la cual se utilizan restricciones conocidas para desarrollar estrategias eficaces de ejecución de consultas.

# 15.1 Traducción de consultas SQL al álgebra relacional

En la práctica, SQL es el lenguaje de consultas que se utiliza en la mayoría de los RDBMS comerciales. Una consulta SQL se traduce, en primer lugar, a una expresión extendida equivalente de álgebra relacional (representada como una estructura de datos de árbol de consultas) que es optimizada posteriormente. Por regla general, las consultas SQL se descomponen en **bloques de consulta** que forman las unidades básicas que se pueden traducir a los operadores algebraicos y después ser optimizadas. Un bloque de consulta contiene una única expresión SELECT-FROM-WHERE, y también cláusulas GROUP BY y HAVING en el caso de que éstas formen parte del bloque. Por tanto, las consultas anidadas dentro de una consulta se identifican como bloques de consulta independientes. Debido a que SQL incluye operadores de agregación (como MAX, MIN, SUM y COUNT), estos operadores deben ser también incluidos en el álgebra extendida, según vimos en la Sección 6.4.

Observe la siguiente consulta SQL sobre la tabla EMPLEADO de la Figura 5.5:

```
SELECT   Apellido1, Nombre
FROM     EMPLEADO
WHERE    Sueldo > ( SELECT   MAX (Sueldo)
                    FROM     EMPLEADO
                    WHERE    Dno=5 );
```

Esta consulta incluye una subconsulta anidada y, por tanto, debería ser descompuesta en dos bloques. El bloque interno es:

```
(SELECT   MAX (Sueldo)
 FROM     EMPLEADO
 WHERE    Dno=5 )
```

y el bloque externo es:

```
SELECT   Apellido1, Nombre
FROM     EMPLEADO
WHERE    Sueldo > c
```

donde c representa el resultado devuelto por el bloque interno. Podríamos traducir el bloque a la expresión extendida de álgebra relacional:

$$\mathcal{T}\text{MAX Sueldo}(\sigma_{\text{Dno}=5}(\text{EMPLEADO}))$$

y el bloque externo a la expresión:

$$\pi_{\text{Apellido1,Nombre}}(\sigma_{\text{Sueldo}>c}(\text{EMPLEADO}))$$

El *optimizador de consultas* seleccionaría a continuación un plan de ejecución para cada bloque. Deberíamos observar que en el ejemplo anterior el bloque interno necesita ser evaluado sólo una vez para obtener el sueldo máximo que será utilizado posteriormente (como constante c) por el bloque externo. En el Capítulo 8 habíamos llamado a esto una *consulta anidada no correlacionada* . Es mucho más difícil optimizar las *consultas anidadas correlacionadas* ya que son más complejas (consulte la Sección 8.5), debido a que una variable de tupla del bloque externo aparece en la cláusula WHERE del bloque interno.

# 15.2 Algoritmos para ordenación externa

La ordenación es uno de los algoritmos principales utilizados en el procesamiento de las consultas. Por ejemplo, siempre que en una consulta SQL se especifique una cláusula ORDER BY, el resultado de la consulta debe ser ordenado. La ordenación es también un componente clave en los algoritmos de ordenación y mezclado utilizados en las operaciones de JOIN y en algunas otras (como UNION y INTERSECTION), y en los algoritmos de eliminación de duplicados de la operación PROYECTO (cuando en una consulta SQL se especifica la opción DISTINCT en la cláusula SELECT). En esta sección veremos uno de estos algoritmos. Es posible evitar la ordenación si existe el índice adecuado que permita un acceso ordenado a los registros.

La **ordenación externa** tiene relación con los algoritmos de ordenación que son adecuados para los ficheros de tamaño grande con registros almacenados en disco que no caben en su totalidad en la memoria principal, como sucede con la mayoría de los ficheros de bases de datos.[3]

El **algoritmo de ordenación externa** más habitual es el que utiliza una **estrategia de ordenación-mezcla**, que comienza con la ordenación de pequeños subficheros (denominados **porciones**) del fichero principal y, a continuación, mezcla esos subficheros ordenados para crear subficheros ordenados más grandes que, a su vez, serán mezclados nuevamente. El algoritmo de ordenación-mezcla, al igual que los algoritmos de bases de datos, necesita de un *espacio temporal* en memoria principal donde se realiza la ordenación y la mezcla de las porciones. El algoritmo básico, descrito en la Figura 15.2, consta de dos fases: la fase de ordenación y la fase de mezclado.

Durante la **fase de ordenación**, las porciones del fichero que caben en el espacio temporal disponible se leen en memoria principal, se ordenan utilizando un algoritmo de ordenación interna y se vuelven a escribir en disco en forma de subficheros temporales ordenados (o porciones). El tamaño de una porción y el **número inicial de porciones ($n_R$)** viene marcado por el **número de bloques de fichero ($b$)** y el **espacio temporal disponible ($n_B$)**. Por ejemplo, si $n_B = 5$ bloques y el tamaño del fichero es $b = 1.024$ bloques, entonces $n_R = \lceil (b/n_B) \rceil$ o 205 porciones iniciales, cada una de ellas con 5 bloques (excepto la última que tendrá 4 bloques). Por tanto, tras la fase de ordenación se almacenarán 205 porciones ordenadas en disco como ficheros temporales.

En la **fase de mezclado**, las porciones ordenadas se mezclan en una o más **pasadas**. El **grado de mezclado ($d_M$)** es el número de porciones que pueden ser mezcladas en cada pasada. Se necesita un bloque de espacio temporal en cada pasada para almacenar un bloque de cada una de las porciones que están siendo mezcladas y se necesita otro bloque que contenga un bloque del resultado de la mezcla. Según esto, $d_M$ es el valor más pequeño entre $(n_B - 1)$ y $n_R$, y el número de pasadas es $\lceil (\log_{dM}(n_R)) \rceil$. En nuestro ejemplo, $d_M = 4$ (mezcla en cuatro pasadas); por tanto, las 205 porciones ordenadas iniciales se mezclarían y se quedarían en 52 al final de la primera pasada. Éstas serían mezcladas nuevamente quedando 13, después 4, y después 1 porción, lo

[3] *Los algoritmos de ordenación interna* son adecuados para la ordenación de estructuras de datos que caben en su totalidad en memoria.

**Figura 15.2.** Descripción del algoritmo de ordenación y mezcla en la ordenación externa.

```
set     i ← 1;
        j ← b;                  {tamaño en bloques del fichero}
        k ← n_B;                {tamaño en bloques del espacio temporal}
        m ← ⌈(j/k)⌉;

{Fase de ordenación}
while (i ≤ m)
do {
        leer los siguientes k bloques del fichero en el búfer o, si quedan menos de k bloques,
            leer los bloques restantes;
        ordenar los registros en el búfer y escribirlos como subfichero temporal;
        i ← i + 1;
}

{Fase de mezcla: mezclar los subficheros hasta que sólo quede 1}
set     i ← 1;
        p ← ⌈log_{k–1m}⌉;       {p es el número de pasadas en la fase de mezcla}
        j ←m;
while (i ≤ p)
do {
        n ← 1;
        q ← ⌈(j/(k–1)⌉;         {número de subficheros a escribir en esta pasada}
        while (n ≤ q)
        do {
            leer los siguientes k–1 subficheros o los subficheros restantes (de pasadas previas),
                    un bloque cada vez;
            mezclar y escribir como nuevo subfichero un bloque cada vez;
            n ← n + 1;
        }
        j ← q;
        i ← + 1;
}
```

que significa que se necesitan *cuatro pasadas*. El valor mínimo de 2 para $d_M$ corresponde al caso peor de la ejecución del algoritmo, que es:

$$(2 * b) + (2 * (b * (\log_2 b))).$$

El primer término representa el número de accesos a bloques durante la fase de ordenación, ya que a cada bloque del fichero se accede dos veces: una vez para ser leído a memoria y otra vez para escribir nuevamente los registros en disco una vez ordenados. El segundo término representa el número de accesos a bloques durante la fase de mezclado, suponiendo el peor caso en el que $d_M$ vale 2. En general, el logaritmo se toma en base $d_M$ y la expresión que indica el número de bloques accedidos queda de este modo:

$$(2 * b) + (2 * (b * (\log_{dM} n_R))).$$

# 15.3 Algoritmos para las operaciones SELECT y JOIN

## 15.3.1 Implementación de la operación SELECT

Existen numerosas opciones a la hora de ejecutar una operación SELECT; algunas dependen de las rutas de acceso específicas al fichero y quizá sólo se apliquen a determinados tipos de condiciones de selección. En esta sección veremos algunos de los algoritmos para la implementación de SELECT. Utilizaremos las siguientes operaciones, que aparecen en la base de datos relacional de la Figura 5.5, para ilustrar nuestra discusión.

OP1: $\sigma_{\text{Dni} = \text{'123456789'}}$ (EMPLEADO)

OP2: $\sigma_{\text{NumeroDpto} > 5}$ (DEPARTAMENTO)

OP3: $\sigma_{\text{Dno} = 5}$ (EMPLEADO)

OP4: $\sigma_{\text{Dno} = 5 \text{ AND Sueldo} > 30000 \text{ AND Sexo} = \text{'F'}}$ (EMPLEADO)

OP5: $\sigma_{\text{DniEmpleado}=\text{'123456789' AND NumProy} = 10}$(TRABAJA_EN)n

**Métodos de búsqueda en una selección simple.** Es posible utilizar varios algoritmos de búsqueda para realizar la selección de registros en un fichero. Estos algoritmos se conocen, a veces, como **exploraciones de fichero**, ya que exploran los registros del fichero en el que se realiza la búsqueda y encuentran los registros que satisfacen una condición de selección.[4] Si el algoritmo de búsqueda implica la utilización de un índice, esta búsqueda mediante un índice se denomina **exploración indexada**. Los siguientes métodos de búsqueda (S1 a S6) son ejemplos de algunos de los algoritmos de búsqueda que se pueden utilizar para implementar una operación de selección.

- **S1—Búsqueda lineal (fuerza bruta).** Extraer todos los registros del fichero y comprobar si sus valores de atributos satisfacen la condición de selección.
- **S2—Búsqueda binaria.** Si la condición de selección implica una comparación de igualdad sobre un atributo clave con el que está ordenado el fichero se puede utilizar la búsqueda binaria, que es más eficaz que la búsqueda lineal. Un ejemplo es OP1 si Dni es el atributo de ordenación para el fichero EMPLEADO.[5]
- **S3—Utilización de un índice primario (o clave *hash* o clave de dispersión).** Si la condición de selección implica una comparación de igualdad sobre un **atributo clave** con un índice primario (o clave *hash*) (por ejemplo, Dni = '123456789' en OP1), se utilizará el índice primario (o clave *hash*) para encontrar el registro. Observe que con esta condición se extrae un único registro (como mucho).
- **S4—Utilización de un índice primario para encontrar varios registros.** Si la condición de comparación es >, >=, <, o <= sobre un campo clave con un índice primario (por ejemplo, NúmeroDpto > 5 en OP2), se utilizará el índice para encontrar el registro que satisfaga la correspondiente condición de igualdad (NúmeroDpto = 5); a continuación, se extraerán los registros posteriores en el fichero ordenado. Si la condición es NúmeroDpto < 5, se extraerán todos los registros precedentes.
- **S5—Utilización de un índice agrupado para encontrar varios registros.** Si la condición de selección implica una comparación de igualdad sobre un **atributo que no es clave** mediante un índice

[4] A veces, a una operación de selección se la denomina **filtro**, ya que filtra los registros del fichero que *no* satisfacen la condición de selección.

[5] Por lo general, no se utiliza la búsqueda binaria en las búsquedas realizadas en bases de datos, ya que no se utilizan los ficheros ordenados a menos que dispongan del correspondiente índice primario.

agrupado (por ejemplo, Dno = 5 en OP3), se utilizará el índice para extraer todos los registros que cumplan la condición.

- **S6—Utilización de un índice secundario (árbol B+) sobre una comparación de igualdad.** Este método de búsqueda se puede utilizar para extraer un único registro si el campo indexado es una **clave** (tiene valores únicos) o para extraer varios registros si el campo indexado **no es clave**. Esto también se puede utilizar en las comparaciones del tipo >, >=, <, o <=.

En la Sección 15.8, veremos cómo desarrollar fórmulas que calculen el coste de acceso de estos métodos de búsqueda en términos de número de accesos a bloques y de tiempo de acceso. El método S1 se aplica a cualquier fichero, pero los demás métodos dependen de si se dispone de la ruta de acceso adecuada al atributo utilizado en la condición de selección. Los métodos S4 y S6 pueden utilizarse para extraer registros en un *rango* determinado (por ejemplo, 30000 <= Sueldo <= 35000). Las consultas que implican condiciones de este tipo se denominan **consultas de rango**.

**Métodos de búsqueda en selecciones complejas.** Cuando la condición de una operación SELECT es una **condición conjuntiva** (es decir, si está formada por varias condiciones simples conectadas mediante la conjunción lógica AND como, por ejemplo, el caso OP4 anterior) el DBMS puede utilizar los siguientes métodos adicionales para implementar la operación:

- **S7—Selección conjuntiva utilizando un índice individual.** Si un atributo implicado en cualquier **condición simple y única** de la condición conjuntiva tiene una ruta de acceso que permite el uso de uno de los métodos S2 a S6, se utilizará esa condición para extraer los registros y, posteriormente, comprobar si cada registro extraído *satisface las condiciones simples restantes* de la condición conjuntiva.
- **S8—Selección conjuntiva utilizando un índice compuesto.** Si dos o más atributos están implicados en condiciones de igualdad de la condición conjuntiva y existe un índice compuesto (o estructura *hash*) sobre los campos combinados (por ejemplo, si se ha creado un índice sobre la clave compuesta [DniEmpleado, NumProy] del fichero TRABAJA_EN en OP5), podemos utilizar el índice directamente.
- **S9—Selección conjuntiva mediante intersección de punteros a registros.**[6] Si se encuentran disponibles índices secundarios (u otras rutas de acceso) sobre más de uno de los campos implicados en condiciones simples de la condición conjuntiva, y si los índices incluyen punteros a registros (en lugar de punteros a bloques) entonces se puede utilizar cada uno de los índices para extraer el **conjunto de punteros a registros** que satisfacen la condición individual. La **intersección** de estos conjuntos de punteros a registros da como resultado los punteros a registros que satisfacen la condición conjuntiva y que serán utilizados posteriormente para extraer esos registros de forma directa. Si sólo tienen índices secundarios algunas de las condiciones, cada uno de los registros extraídos será revisado posteriormente para determinar si cumple las condiciones restantes.[7]

Siempre que una condición única especifique la selección (como en el caso de OP1, OP2 u OP3), podemos limitarnos a comprobar si existe una ruta de acceso sobre el atributo implicado en esa condición. Si existe una ruta de acceso, se utilizará el método que corresponde a dicha ruta; en caso contrario, se puede utilizar el modelo de búsqueda de fuerza bruta lineal del método S1. Es necesaria la optimización de consultas en una operación SELECT en la mayoría de los casos de condiciones de selección conjuntivas en las que *más de uno* de los atributos implicados en la condición dispone de una ruta de acceso. El optimizador deberá elegir la ruta

[6] Un puntero a registro identifica de manera única a un registro y proporciona la dirección del registro en el disco; por tanto, también se le denomina **identificador de registro**.

[7] La técnica puede ofrecer muchas variantes (por ejemplo, si los índices son *índices lógicos* que almacenan valores de clave primaria en lugar de punteros a registros).

de acceso que *extraiga el menor número de registros* del modo más eficaz mediante la estimación de los diferentes costes (consulte la Sección 15.8) y la elección del método con el menor coste estimado.

Cuando el optimizador tiene que elegir entre las distintas condiciones simples dentro de una condición de selección conjuntiva, lo que hace normalmente es tener en cuenta la selectividad de cada una de las condiciones. La **selectividad** se define como la relación entre el número de registros (tuplas) que satisfacen la condición y el número total de registros (tuplas) del fichero y, de acuerdo con esto, es un número entre cero y 1 (selectividad cero significa que ningún registro satisface la condición y 1 significa que todos los registros satisfacen la condición). Aunque es posible que no estén disponibles selectividades exactas para todas las condiciones, a menudo se guardan en el catálogo del DBMS unas **estimaciones de selectividades** para que sean utilizadas por el optimizador. Por ejemplo, para una condición de igualdad sobre un atributo clave de la relación $r(R)$, $s = 1/|r(R)|$, donde $|r(R)|$ es el número de tuplas de la relación $r(R)$. Para una condición de igualdad sobre un atributo con $i$ *valores distintos*, $s$ se puede estimar a partir de $(|r(R)|/i)/|r(R)|$ o $1/i$, suponiendo que los registros se encuentran distribuidos de manera uniforme entre los distintos valores.[8] Con esta suposición, un total de $|r(R)|/i$ registros cumplirán una condición de igualdad sobre este atributo. Por lo general, el número estimado de registros que satisfacen una condición de selección con selectividad $s$ es $|r(R)| * s$. Cuanto menor sea este valor estimado, más recomendable será tener en cuenta esa condición para extraer los registros.

Una **condición disyuntiva** (en la cual las condiciones simples están conectadas mediante la operación lógica OR en lugar de estarlo mediante AND) es mucho más difícil de procesar y optimizar que una condición de selección conjuntiva. Tomemos como ejemplo OP4':

OP4': $\sigma_{\text{Dno=5 OR Sueldo>30000 OR Sexo='M'}}$ (EMPLEADO)

Con una condición de este tipo no se puede realizar demasiada optimización, ya que los registros que satisfacen la condición disyuntiva son la *unión* de los registros que satisfacen las condiciones individuales. Por tanto, si *una* cualquiera de las condiciones no dispone de ruta de acceso, nos veremos obligados a utilizar el modelo de búsqueda de fuerza bruta lineal. Sólo si existe una ruta de acceso para *todas* las condiciones podremos optimizar la selección obteniendo los registros (o los identificadores de registro) que satisfacen cada una de las condiciones para después utilizar la operación de unión para eliminar duplicados.

Un DBMS suele disponer de muchos de los métodos discutidos anteriormente y, por lo general, muchos otros métodos adicionales. El optimizador de consultas elegirá el más adecuado para la ejecución de cada operación SELECT en una consulta. Esta optimización utiliza fórmulas para estimar los costes de cada uno de los métodos de acceso disponibles, como veremos en la Sección 15.8. El optimizador elegirá el método de acceso que tenga el menor coste estimado.

### 15.3.2 Implementación de la operación JOIN

La operación JOIN es una de las operaciones que más tiempo consumen durante el procesamiento de una consulta. Muchas de las operaciones de concatenación que se pueden encontrar en las consultas son de los tipos EQUIJOIN y NATURAL JOIN, así que aquí sólo tendremos en cuenta estos dos. Durante el resto de este capítulo, el término **concatenación** se refiere a una EQUIJOIN (o NATURAL JOIN). Existen muchas maneras de implementar una operación de concatenación que involucre a dos ficheros (**concatenación de dos vías**). Las concatenaciones que involucran a más de dos ficheros se denominan **concatenaciones multivía**. El número de vías posibles para ejecutar concatenaciones multivía crece con mucha rapidez. En esta sección sólo revisaremos técnicas para la implementación de concatenaciones del tipo dos-vías. Para ilustrar nuestra discusión utilizaremos, una vez más, el esquema relacional de la Figura 5.5, en concreto, utilizaremos las relaciones

[8] En los optimizadores más sofisticados es posible almacenar en el catálogo los histogramas que representan la distribución de los registros entre los diferentes valores de los atributos.

EMPLEADO, DEPARTAMENTO y PROYECTO. Los algoritmos que estamos tratando se utilizan en operaciones de concatenación de la forma

$$R \bowtie_{A=B} S$$

donde $A$ y $B$ son atributos compatibles en dominio de $R$ y $S$, respectivamente. Los métodos que estamos discutiendo pueden ser extendidos a formas más generales de concatenación. Mostraremos cuatro de las técnicas más habituales de ejecutar una concatenación de este tipo utilizando las siguientes operaciones de ejemplo:

OP6: EMPLEADO $\bowtie_{\text{Dno}=\text{NúmeroDpto}}$ DEPARTAMENTO

OP7: DEPARTAMENTO $\bowtie_{\text{DniDirector}=\text{Dni}}$ EMPLEADO

### Métodos para la implementación de concatenaciones.

- **J1—Concatenación de bucle anidado (fuerza bruta).** Para cada registro $t$ de $R$ (bucle externo), obtiene todos los registros $s$ de $S$ (bucle interno) y comprueba si los dos registros satisfacen la condición de concatenación $t[A] = s[B]$.[9]
- **J2—Concatenación de bucle simple (utilizando una estructura de acceso para obtener los registros que cumplen la condición).** Si existe un índice (o clave *hash*) para uno de los atributos de la concatenación (por ejemplo, $B$ de $S$), obtiene todos los registros $t$ de $R$, uno a la vez (bucle simple), y a continuación utiliza la estructura de acceso para obtener directamente todos los registros $s$ de $S$ que cumplen $s[B] = t[A]$.
- **J3—Concatenación de ordenación-mezcla.** Si los registros de $R$ y $S$ se encuentran *físicamente clasificados* (ordenados) por el valor de los atributos de concatenación $A$ y $B$ respectivamente, podemos implementar la concatenación del modo más eficiente posible. Ambos ficheros serán explorados concurrentemente siguiendo el orden de los atributos de concatenación y emparejando los registros que tienen los mismos valores para $A$ y $B$. Si los ficheros no se encuentran ordenados, podrían ser ordenados previamente mediante una ordenación externa (consulte la Sección 15.2). Según este método, las parejas de los bloques de los ficheros se copian en los búferes de memoria en orden y los registros de cada fichero se exploran sólo una vez cada uno para comprobar su emparejamiento con el otro fichero (a menos que $A$ y $B$ no sean atributos clave, en cuyo caso el método necesita ser modificado ligeramente). En la Figura 15.3(a) se muestra un esquema del algoritmo de concatenación de clasificación-mezcla. Utilizamos $R(i)$ para referirnos al *i-ésimo* registro de $R$. Se puede utilizar una variación de la concatenación de clasificación-mezcla cuando existen los índices secundarios en ambos atributos de concatenación. Los índices proporcionan la posibilidad de acceder (explorar) a los registros siguiendo el orden de los atributos de concatenación, pero ya que los registros se encuentran dispersos por todos los bloques del fichero este método puede resultar bastante ineficaz, ya que todos los accesos a registros pueden implicar el acceso a distintos bloques en disco.
- **J4—Concatenación de dispersión (*hash*).** Los registros de los ficheros $R$ y $S$ se encuentran clasificados en el mismo fichero de dispersión, utilizando la misma función de dispersión sobre los atributos $A$ de $R$ y $B$ de $S$ como claves de dispersión. En primer lugar, una única pasada sobre el fichero con menor número de registros (por ejemplo, $R$ ) reparte sus registros en los bloques del fichero de dispersión; esto se denomina **fase de particionamiento**, ya que los registros de $R$ se particionan sobre los bloques de dispersión. En la segunda fase, denominada **fase de prueba**, se hace una única pasada por el otro fichero ($S$) tomando cada uno de los registros, *probando* en el bloque adecuado y emparejándolos con los registros correspondientes de $R$ en ese bloque. En esta descripción simplificada de la concatenación

[9] En los ficheros en disco, es obvio que los bucles se ejecutarán sobre bloques en disco y, por tanto, esta técnica también recibe el nombre de *concatenación anidada en bloque*.

de dispersión se supone que el fichero de menor tamaño *cabe en su totalidad en los bloques de memoria* tras la primera fase. Veremos más adelante algunas variaciones de la concatenación de dispersión en las que no se presupone esta afirmación.

**Figura 15.3.** Implementación de JOIN, PROJECT, UNION, INTERSECTION y SET DIFFERENCE utilizando clasificación-mezcla, donde *R* tiene *n* tuplas y *S* tiene *m* tuplas. (a) Implementación de la operación $T \leftarrow R \bowtie_{A=B} S$. (b) Implementación de la operación $T \leftarrow \pi_{<\text{lista de atributos}>}(R)$.

**(a)**
```
clasificación de las tuplas de R sobre el atributo A;   (* suponiendo que R tiene n tuplas (registros) *)
clasificación de las tuplas de S sobre el atributo B;   (* suponiendo que S tiene m tuplas (registros) *)
set i ← 1, j ← 1;
while (i ≤ n) and (j ≤ m)
do {   if R(i)[A] > S(j)[B]
           then set j ← j + 1
       elseif R(i)[A] < S(j)[B]
           then     set i ← i + 1
       else { (* R(i)[A] = S(j)[B], así que generamos una tupla emparejada *)
              generar la tupla combinada <R(i), S(j)> en T;

              (* generar otras tuplas que cuadren con R(i), si existen *)
              set l ← j + 1;
              while (l ≤ m) and (R(i)[A] = S(l)[B])
              do {   generar la tupla combinada <R(i), S(l)> en T;
              set l ← l + 1
              }

              (* generar otras tuplas que cuadren con S(j), si existen *)
              set k ¬ i + 1;
              while (k ≤ n) and (R(k)[A] = S(j)[B])
              do {   generar la tupla combinada <R(k), S(j)> en T;
                     set k ← k + 1
              }
              set i ← k, j ← l
   }
}
```

**(b)**
```
crear una tupla t[<lista de atributos>] en T' para cada tupla t de R;
        (* T' contiene los resultados de la proyección antes de la eliminación de duplicados *)
if <lista de atributos> incluye una clave de R
        then T ← T'
else {  ordenar las tuplas de T';
        set i ← 1, j ← 2;
        while i ≤ n
        do {   generar la tupla T'[i] en T;
               while T'[i] = T'[j] and j ≤ n do j ← j + 1; (* eliminar duplicados *)
               i ← j; j ← i + 1
        }
}
(* T contiene el resultado de la proyección tras la eliminación de duplicados *)
```

**Figura 15.3.** (*continuación*) Implementación de JOIN, PROJECT, UNION, INTERSECTION y SET DIFFERENCE utilizando clasificación-mezcla, donde $R$ tiene $n$ tuplas y $S$ tiene $m$ tuplas. (c) Implementación de la operación $T \leftarrow R \cup S$. (d) Implementación de la operación $T \leftarrow R \cap S$. (e) Implementación de la operación $T \leftarrow R - S$.

**(c)**
```
ordenar las tuplas de R y S utilizando los mismos atributos de ordenación únicos;
set i ← 1, j ← 1;
while (i ≤ n) and ( j ≤ m)
do { if R(i) > S(j)
        then {  generar S(j) en T;
                set j ← j + 1
        }
   elseif R(i) < S(j)
        then {  generar R(i) en T;
                set i ← i + 1
        }
   else set j ← j + 1 (* R(i)=S(j), por lo que nos saltamos una de las tuplas duplicadas *)
}
if (i ≤ n) then agregar tuplas R(i) a R(n) en T;
if ( j ≤ m) then agregar tuplas S(j) a S(m) en T;
```

**(d)**
```
ordenar las tuplas de R y S utilizando los mismos atributos de ordenación únicos;
set i ← 1, j ← 1;
while (i ≤ n) and (j ≤ m)
do {   if R(i) > S(j)
        then set j ← j + 1
   elseif R(i) < S(j)
        then set i ← i + 1
   else {  output R(j) to T; (* R(i)=S(j), así que generamos la tupla *)
        set i ← i + 1, j ← j + 1
   }
}
```

**(e)**
```
ordenar las tuplas de R y S utilizando los mismos atributos de ordenación únicos;
set i ← 1, j ← 1;
while (i ≤ n) and (j ≤ m)
do {   if R(i) > S(j)
            then set j ← j + 1
       elseif R(i) < S(j)
            then { generar R(i) en T; (* R(i) no se empareja con S(j), así que generamos R(i) *)
            set i ← i + 1
            }
       else set i ← i + 1, j ← j + 1
}
if (i ≤ n) then agregar tuplas R(i) a R(n) en T;
```

---

En la práctica, las técnicas J1 a J4 se implementan accediendo a *bloques completos en disco* de un fichero, en lugar de a registros individuales. Se puede ajustar el número de bloques leídos del fichero en función del espacio disponible para búferes en memoria.

**Efectos del espacio disponible para búferes y del factor de selección de concatenaciones sobre la ejecución de concatenaciones.** El espacio disponible para búferes tiene un efecto importante sobre los diferentes algoritmos de concatenación. En primer lugar, tomemos el modelo de bucle anidado (J1). Echando un vistazo de nuevo a la operación OP6 anterior, supongamos que el número de búferes disponibles en memoria principal para implementar la concatenación es $n_B = 7$ bloques (búferes). Como ejemplo, supongamos que el fichero DEPARTAMENTO está formado por $r_D = 50$ registros almacenados en $b_D = 10$ bloques de disco y que el fichero EMPLEADO está formado por $r_E = 6.000$ registros almacenados en $b_E = 2.000$ bloques de disco. Resulta ventajoso leer a memoria tantos bloques como sea posible de una vez del fichero cuyos registros sean los utilizados en el bucle externo (es decir, $n_B - 2$ bloques). Tras esto, el algoritmo podrá leer un bloque cada vez en el fichero usado en el bucle interno y utilizar sus registros para **probar** (es decir, buscar) en los bloques del bucle externo en memoria los registros que puedan ser emparejados. Esto reduce el número total de accesos a bloques. Se necesita un bloque de búfer adicional para albergar los registros resultantes una vez que han sido unidos y el contenido de este bloque de búfer será agregado al **fichero resultante** (el fichero en disco que contendrá el resultado de la concatenación) cuando se llene. Posteriormente, este bloque de búfer será reutilizado para albergar registros adicionales del resultado.

En la concatenación del bucle anidado, existe una diferencia dependiendo del fichero elegido para el bucle externo y del elegido para el bucle interno. Si EMPLEADO se utiliza para el bucle externo, cada uno de los bloques de EMPLEADO se leerá una sola vez y el fichero DEPARTAMENTO en su totalidad (cada uno de sus bloques) será leído una vez *cada vez* que leamos $(n_B - 2)$ bloques del fichero EMPLEADO. Obtendremos lo siguiente:

Número total de bloques accedidos en el fichero externo $= b_E$

Número de veces $(n_B - 2)$ que se cargan los bloques del fichero externo $= \lceil b_E/(n_B - 2) \rceil$

Número total de bloques accedidos en el fichero interno $= b_D * \lceil b_E/(n_B - 2) \rceil$

Por tanto, obtendremos el siguiente número total de accesos a bloque:

$$b_E + (\lceil b_E/(n_B - 2) \rceil * b_D) = 2000 + (\lceil (2.000/5) \rceil * 10) = 6.000 \text{ accesos a bloque}$$

Por otra parte, si utilizamos los registros de DEPARTAMENTO en el bucle externo, por simetría obtendríamos el siguiente número total de accesos a bloque:

$$b_D + (\lceil b_D/(n_B - 2) \rceil * b_E) = 10 + (\lceil (10/5) \rceil * 2.000) = 4.010 \text{ accesos a bloque}$$

El algoritmo de concatenación utiliza un búfer para albergar los registros del fichero resultante. Una vez que se llena el búfer, se escribe a disco y se reutiliza.[10]

Si el fichero resultante de la operación de concatenación tiene $b_{RES}$ bloques de disco, cada uno de los bloques se escribe sólo una vez; por tanto, se deberían añadir $b_{RES}$ accesos a bloque adicionales a las fórmulas anteriores para calcular el coste total de la operación de concatenación. Lo mismo se aplica en las fórmulas desarrolladas posteriormente para los otros algoritmos de concatenación. Según se muestra en este ejemplo, resulta ventajoso utilizar el fichero *con menor número de bloques* en el bucle externo de la concatenación de bucle anidado.

Otro factor que afecta a la ejecución de una concatenación, en particular con el método de bucle simple J2, es el porcentaje de registros de un fichero que serán concatenados con los registros del otro fichero. A esto se le denomina factor de **selección de concatenación**[11] de un fichero con respecto a la condición de *equijoin* con otro fichero. Este factor depende de la condición de *equijoin* en particular entre los dos ficheros. Para ilustrar esto, tomemos la operación OP7, en la que se une cada registro de DEPARTAMENTO con el registro de EMPLEADO correspondiente al responsable de ese departamento. En este caso, se pretende que cada registro

[10] Si reservamos dos búferes para el fichero resultante se puede utilizar doble búfer para acelerar el algoritmo (consulte la Sección 13.3).

[11] Esto es diferente de la selectividad de concatenación, que veremos en la sección 15.8.

de DEPARTAMENTO (en nuestro ejemplo hay 50 registros de este tipo) se una con un *único* registro de EMPLEADO, aunque muchos de ellos (los 5.950 que no son responsables de ningún departamento) quedarán fuera de la operación de concatenación.

Supongamos que existen unos índices secundarios en ambos atributos Dni de EMPLEADO y DniDirector de DEPARTAMENTO, con número de niveles de índice $x_{\text{Dni}} = 4$ y $x_{\text{DniDirector}} = 2$, respectivamente. Tenemos dos opciones para la implementación del método J2. La primera obtiene cada registro de EMPLEADO y después utiliza el índice sobre DniDirector de DEPARTAMENTO para encontrar un registro de DEPARTAMENTO que sea emparejable. En este caso, no se encontrará ningún registro emparejable para los empleados que no son responsables de un departamento. El número de accesos a bloque en este caso es, aproximadamente,

$$b_E + (r_E * (x_{\text{DniDirector}} + 1)) = 2.000 + (6.000 * 3) = 20.000 \text{ accesos a bloque}$$

La segunda opción obtiene cada registro de DEPARTAMENTO y después utiliza el índice sobre Dni de EMPLEADO para encontrar un registro de EMPLEADO que sea emparejable como responsable. En este caso, para todos los registros de DEPARTAMENTO existirá un registro de EMPLEADO que sea emparejable. El número de accesos a bloque en este caso es, aproximadamente,

$$b_D + (r_D * (x_{\text{Ssn}} + 1)) = 10 + (50 * 5) = 260 \text{ accesos a bloque}$$

La segunda opción es más eficiente porque el factor de selección de concatenación de DEPARTAMENTO con respecto a la condición de concatenación Dni = DniDirector es 1, mientras que el factor de selección de concatenación de EMPLEADO con respecto a la misma condición de concatenación es (50/6.000), o 0,008. En el método J2 se debería usar en el bucle (externo) de concatenación o bien el fichero de menor tamaño o bien el fichero que tiene todos sus registros emparejables (es decir, el fichero con el factor de selección de concatenación más alto). También se puede crear un índice específico para ejecutar la operación de concatenación si es que todavía no existe ninguno.

La concatenación de ordenación-mezcla J3 es muy eficiente si ambos ficheros se encuentran ordenados por su atributo de concatenación. Sólo se realiza una única pasada por cada fichero. Por tanto, el número de bloques accedidos es igual a la suma del número de bloques en ambos ficheros. Para este método, tanto OP6 como OP7 necesitarían $b_E + b_D = 2.000 + 10 = 2.010$ accesos a bloque. Sin embargo, es necesario que ambos ficheros se encuentren ordenados por los atributos de concatenación; si uno de ellos o ambos no lo están, pueden ser ordenados expresamente para ejecutar la operación de concatenación. Si calculamos el coste de ordenar un fichero externo como $(b \log_2 b)$ accesos a bloque y si ambos ficheros necesitan ser ordenados, el coste total de una concatenación de ordenación-mezcla se podría calcular como $(b_E + b_D + b_E \log_2 b_E + b_D \log_2 b_D)$.[12]

**Concatenación de dispersión particionada y concatenación de dispersión híbrida.** El método de concatenación de dispersión J4 también es bastante eficiente. En este caso, sólo se realiza una única pasada en cada fichero, independientemente de que los ficheros estén o no ordenados. Si la tabla de dispersión del más pequeño de los dos ficheros se puede guardar en su totalidad en la memoria principal después de haber realizado la operación de dispersión (particionamiento) sobre su atributo de concatenación, la implementación es sencilla. Si, en algún caso, algunas partes del fichero de dispersión deben ser almacenadas en disco, el método será algo más complejo, aunque se han propuesto algunas variaciones al método para mejorar su eficiencia. Veremos dos técnicas: concatenación de dispersión particionada y una variación denominada concatenación de dispersión híbrida, que ha resultado ser muy eficiente.

En el **algoritmo de concatenación de dispersión particionada**, cada uno de los ficheros se divide, en primer lugar, en $M$ particiones utilizando una función de particionamiento de dispersión sobre los atributos de la concatenación. A continuación, uniremos cada pareja de particiones. Por ejemplo, supongamos que estamos realizando una concatenación en las relaciones $R$ y $S$ sobre los atributos de concatenación $R.A$ y $S.B$:

[12] Podemos usar las fórmulas más precisas de la Sección 15.2 si conocemos el número de búferes disponibles para la ordenación.

$$R \bowtie_{A=B} S$$

Durante la **fase de particionamiento**, $R$ se divide en las $M$ particiones $R_1, R_2, ..., R_M$ y $S$ en las $M$ particiones $S_1, S_2, ..., S_M$. La propiedad de cada par de particiones relacionadas $R_i, S_i$ es que los registros de $R_i$ *sólo necesitan hacer la concatenación* con los registros de $S_i$, y viceversa. Esta propiedad se confirma utilizando la misma función de dispersión sobre los atributos de concatenación para particionar ambos ficheros (el atributo $A$ para $R$ y el atributo $B$ para $S$). El mínimo número de búferes necesarios para la fase de partición es $M + 1$. En cada uno de los ficheros $R$ y $S$ se particionará por separado. Para cada una de las particiones, se ubica en memoria un único búfer (de tamaño un bloque de disco) para almacenar los registros sobre los que se realiza la dispersión en esa partición. En el momento en el que el búfer de memoria para esa partición se llene, su contenido se agregará a un **subfichero de disco** que almacena esa partición. La fase de particionamiento se realiza en *dos pasadas*. Tras la primera pasada, el primer fichero $R$ se divide en los subficheros $R_1, R_2, ..., R_M$, donde todos los ficheros que han sido sometidos a la dispersión se encuentran en la misma partición. Tras la segunda pasada, el segundo fichero $S$ se particiona del mismo modo.

En la segunda fase, denominada **fase de concatenación o de prueba**, se necesitan $M$ *iteraciones*. Durante la iteración $j$, se unen las dos particiones $R_i$ y $S_i$. El número mínimo de búferes necesarios para la iteración $j$ es el número de bloques de la partición de menor tamaño, por ejemplo $R_i$, más dos búferes adicionales. Si utilizamos una concatenación de bucle anidado durante la iteración $i$, los registros de la más pequeña de las particiones $R_i$ se copian en búferes de memoria; a continuación, se leen todos los bloques de la otra partición $S_i$ (uno cada vez) y se utiliza cada registro para **probar** (es decir, buscar) en la partición $R_i$ en busca de los registros implicados en la operación. Estos registros se someten a la operación de concatenación y se escriben en el fichero resultante. Para mejorar la eficiencia de la prueba en memoria, es habitual utilizar una *tabla de dispersión en memoria* para almacenar los registros de la partición $R_i$ utilizando una función de dispersión *diferente* de la función de dispersión de particionamiento.[13]

Podemos estimar el coste de este particionamiento como $3 * (b_R + b_S) + b_{RES}$ en nuestro ejemplo, ya que cada registro se lee una vez y se escribe nuevamente en disco una vez durante la fase de particionamiento. Durante la fase de concatenación (prueba), se lee cada registro una segunda vez para realizar la concatenación. La *dificultad principal* de este algoritmo es asegurar que la función de dispersión de particionamiento es **uniforme** (es decir, los tamaños de particionamiento son casi iguales entre sí). Si la función de particionamiento **no es uniforme**, entonces quizá algunas particiones sean demasiado grandes para caber en el espacio de memoria disponible en la segunda fase de la concatenación.

Observe que si el espacio de búferes en memoria disponible $n_B > (b_R + 2)$, siendo $b_R$ el número de bloques del *más pequeño* de los dos ficheros sobre los que se hace la concatenación, por ejemplo $R$, entonces no hay motivo para realizar el particionamiento, ya que, en ese caso, la operación de concatenación se puede realizar por completo en memoria utilizando alguna variación de la concatenación de bucle anidado basada en dispersión y prueba. Como ejemplo, supongamos que estamos ejecutando la operación de concatenación OP6, repetida a continuación:

$$\text{OP6: EMPLEADO} \bowtie_{\text{Dno=NúmeroDpto}} \text{DEPARTAMENTO}$$

En este ejemplo, el fichero de menor tamaño es DEPARTAMENTO; por tanto, si el número de búferes de memoria disponibles $n_B > (b_D + 2)$, es posible leer a memoria el fichero DEPARTAMENTO en su totalidad y organizarlo en una tabla de dispersión sobre el atributo de concatenación. A continuación, se lee cada bloque EMPLEADO sobre un bloque y se realiza una operación de dispersión sobre el atributo de concatenación en cada registro EMPLEADO del búfer que, a su vez, será utilizado para probar en el correspondiente bloque en memoria de la tabla de dispersión DEPARTAMENTO. Si se encuentra un registro que cumpla la condición, se realiza una operación de concatenación sobre los registros y el registro, o registros, resultantes se escribirán

[13] Si la función de dispersión usada en el particionamiento se utiliza de nuevo, todos los registros de la partición serán repartidos de nuevo en el mismo bloque de memoria.

en el búfer de resultados y, posiblemente, en el fichero de resultados en disco. El coste en términos de accesos a bloques es, por tanto, ($b_D + b_E$), más $b_{RES}$ (el coste de escribir el fichero de resultados).

El **algoritmo de concatenación de dispersión híbrida** es una variación de la concatenación de dispersión particionada, donde la fase de *concatenación* en *una de las particiones* está incluida en la fase de *particionamiento*. Para ilustrar esto, supongamos que el tamaño de un búfer de memoria es un bloque de disco, que están disponibles $n_B$ búferes de este tipo y que la función de dispersión utilizada es $h(K) = K \bmod M$ para que se creen $M$ particiones, donde $M < n_B$. Como ejemplo, supongamos que estamos ejecutando la operación de concatenación OP6. En la primera pasada de la fase de particionamiento, cuando el algoritmo de concatenación de dispersión híbrida se encuentra particionando el más pequeño de los dos ficheros (DEPARTAMENTO en OP6), el algoritmo divide el espacio de búfer entre las M particiones, de modo que todos los bloques de la *primera partición* de DEPARTAMENTO residen en su totalidad en la memoria principal. Para cada una de las otras particiones, sólo se reserva un único búfer (con un tamaño de un bloque de disco) en memoria; el resto de la partición se escribe a disco como en el caso de la concatenación de dispersión en una partición normal. Por tanto, al final de la *primera pasada de la fase de particionamiento*, la primera partición de DEPARTAMENTO reside por completo en la memoria principal, mientras que cada una de las otras particiones de DEPARTAMENTO reside en un subfichero de disco.

En la segunda pasada de la fase de particionamiento, son los registros del segundo fichero que está siendo unido (el fichero de mayor tamaño, EMPLEADO en OP6) los que se están particionando. Si un registro se asigna a la *primera partición*, se une con el registro de DEPARTAMENTO con el que se deba realizar el emparejamiento y los registros unidos se escriben en el búfer resultante (y posiblemente en disco). Si un registro de EMPLEADO se asigna a una partición distinta de la primera, se particiona de forma normal. Según esto, al final de la segunda pasada de la fase de particionamiento todos los registros que fueron asignados a la primera partición han sido unidos. En este momento habrá $M - 1$ pares de particiones en disco. Por tanto, durante la segunda **unión** o fase de **prueba**, se necesitan $M - 1$ *iteraciones* en lugar de $M$. El objetivo es unir el mayor número posible de registros durante la fase de particionamiento para ahorrar el coste de volver a almacenar esos registros y volver a leerlos una segunda vez durante la fase de unión.

## 15.4 Algoritmos para las operaciones de proyección y de conjunto

Una operación PROYECCIÓN $\pi_{<\text{lista de atributos}>}(R)$ es fácil de implementar si <lista de atributos> incluye una clave de la relación $R$, ya que en este caso el resultado de la operación tendrá el mismo número de tuplas que $R$, pero con sólo los valores de los atributos de <lista de atributos> en cada tupla. Si <lista de atributos> no incluye una clave de $R$, *las tuplas duplicadas deben ser eliminadas*. Estro se hace habitualmente ordenando el resultado de la operación y eliminando después las tuplas duplicadas, que aparecen consecutivamente tras la ordenación. Un esquema del algoritmo aparece en la Figura 15.3(b). También se puede usar la dispersión para eliminar duplicados: a medida que cada registro es asignado a un bloque en memoria del fichero de dispersión, se comprueba con los ya existentes en ese bloque; si se trata de un duplicado, no será insertado. Resulta útil recordar aquí que en las consultas SQL la acción predeterminada es no eliminar los duplicados del resultado de la consulta; sólo si se incluye la palabra clave DISTINCT se eliminarán los duplicados de dicho resultado.

Las operaciones de conjunto (UNION, INTERSECTION, SET DIFFERENCE y CARTESIAN PRODUCT) son a veces difíciles de implementar. En particular, la operación CARTESIAN PRODUCT $R \times S$ es muy costosa porque su resultado incluye un registro para cada combinación de registros de $R$ y $S$. Además, los atributos del resultado incluyen todos los atributos de $R$ y $S$. Si $R$ tiene $n$ registros y $j$ atributos y $S$ tiene $m$ registros y $k$ atributos, la relación resultante tendrá $n * m$ registros y $j + k$ atributos. Por tanto, es importante evitar la operación CARTESIAN PRODUCT y sustituir otras operaciones equivalentes durante la optimización de consultas (consulte la Sección 15.7).

Las otras tres operaciones de conjunto (UNION, INTERSECTION y SET DIFFERENCE[14]) se aplican sólo a relaciones compatibles en unión, las cuales tienen el mismo número de atributos y los mismos dominios de atributo. La forma habitual de implementar estas operaciones es utilizar variaciones de la técnica de ordenación-mezcla: las dos relaciones se ordenan por los mismos atributos y, después de la ordenación, una única exploración por cada relación es suficiente para obtener el resultado. Por ejemplo, podemos implementar la operación UNION, $R \cup S$, explorando y mezclando de forma concurrente ambos ficheros y, cuando se encuentre la misma tupla en ambas relaciones, sólo se guardará una de ellas en la mezcla resultante. En el caso de la operación INTERSECTION, $R \cap S$, guardaremos en la mezcla resultante sólo esas tuplas que aparecen en *ambas relaciones*. En las Figuras 15.3(c) a (e) aparece un esquema de la implementación de estas operaciones mediante ordenación y mezcla. Algunos de los detalles no aparecen en estos algoritmos.

También se puede utilizar el método de dispersión para implementar UNION, INTERSECTION y SET DIFFERENCE. Una de las tablas se particiona y la otra se usa para probar en la partición adecuada. Por ejemplo, para implementar $R \cup S$, en primer lugar se realiza la dispersión (se particiona) en los registros de $R$; a continuación, se realiza la dispersión (se prueba) en los registros de $S$, pero no se insertan los registros duplicados en los bloques de memoria. Para implementar $R \cap S$, en primer lugar se particionan los registros de $R$ en el fichero de dispersión. Posteriormente, mientras se realiza la dispersión en cada uno de los registros de $S$, se realiza la prueba para comprobar si se encuentra un registro idéntico de $R$ en el bloque de memoria y, si es así, se agrega el registro al fichero resultante. Para implementar $R - S$, en primer lugar se particionan los registros de $R$ en los bloques en memoria del fichero de dispersión. Mientras se realiza la dispersión (se prueba) de cada registro de $S$, si se encuentra un registro idéntico en el bloque de memoria, se elimina ese registro del bloque.

# 15.5 Implementación de las operaciones de agregación y de OUTER JOIN

## 15.5.1 Implementación de las operaciones de agregación

Los operadores de agregación (MIN, MAX, COUNT, AVERAGE, SUM), cuando se aplican a una tabla completa, se pueden calcular mediante una exploración de tabla o utilizando el índice apropiado, si está disponible. Por ejemplo, veamos la siguiente consulta SQL:

```
SELECT MAX    (Sueldo)
FROM          EMPLEADO;
```

Si existe un índice (ascendente) sobre Sueldo para la relación EMPLEADO, entonces el optimizador puede decidir si utilizar el índice para buscar el valor más grande siguiendo el puntero *más a la derecha* en cada nodo de índices desde la raíz hasta la hoja más a la derecha. Ese nodo incluiría el valor más grande de Sueldo en su *última* entrada. En la mayoría de los casos, esto sería más eficiente que una exploración por toda la tabla EMPLEADO, ya que no se necesita obtener ningún registro. El agregado MIN se puede tratar de un modo parecido, exceptuando que sería el puntero *más a la izquierda* el que se seguiría desde la raíz hasta la hoja más a la izquierda. Ese nodo incluiría el valor más pequeño de Sueldo en su *primera* entrada.

El índice también se puede utilizar para los agregados COUNT, AVERAGE y SUM, pero sólo si se trata de un **índice denso**, es decir, si hay una entrada de índice para cada registro del fichero principal. En este caso, el cálculo asociado se aplicaría a los valores del índice. En el caso de un **índice no denso**, se debe usar el número real de registros asociados con cada entrada de índice para que el cálculo sea correcto (excepto para COUNT DISTINCT, donde el número de valores distintos puede obtenerse del propio índice).

[14] SET DIFFERENCE se llama EXCEPT en SQL.

Cuando se utiliza una cláusula GROUP BY en una consulta, el operador agregado se debe aplicar por separado a cada grupo de tuplas. Por tanto, en primer lugar se debe particionar la tabla en subgrupos de tuplas, teniendo cada partición (grupo) el mismo valor en los atributos de agrupación. Es este caso, el cálculo es más complejo. Observemos la siguiente consulta:

```
SELECT    Dno, AVG(Sueldo)
FROM      EMPLEADO
GROUP BY  Dno;
```

La técnica habitual para las consultas de este tipo es utilizar en primer lugar o la **ordenación** o la **dispersión** sobre los atributos de agrupación para particionar el fichero en los grupos adecuados. Posteriormente, el algoritmo calcula la función de agregación en las tuplas de cada grupo, que tienen el mismo valor en los atributos de agrupación. En la consulta de ejemplo, el conjunto de tuplas para cada número de departamento se agruparía en una partición y se calcularía el salario medio para cada grupo.

Observe que si existe un **índice agrupado** (consulte el Capítulo 14) sobre los atributos de agrupación, los registros *ya se encuentran particionados* (agrupados) en los subconjuntos apropiados. En este caso, sólo es necesario aplicar el cálculo a cada grupo.

## 15.5.2 Implementación de la concatenación externa

En la Sección 6.4 se mostró la operación de concatenación externa, con sus tres variantes: concatenación externa izquierda, concatenación externa derecha y concatenación externa completa. También vimos en el Capítulo 8 cómo se pueden especificar estas operaciones en SQL. A continuación se muestra un ejemplo de concatenación externa izquierda en SQL:

```
SELECT    Apellido1, Nombre, NombreDpto
FROM      (EMPLEADO LEFT OUTER JOIN DEPARTAMENTO ON Dno=NúmeroDpto);
```

El resultado de esta consulta es una tabla de nombres de empleado y de sus departamentos asociados. Es similar al resultado de una concatenación normal (interna), con la excepción de que si una tupla de EMPLEADO (una tupla de la relación de la *izquierda*) *no tiene un departamento asociado*, el nombre del empleado seguirá apareciendo en la tabla resultante, aunque el nombre del departamento sería NULL en esas tuplas del resultado de la consulta.

La concatenación externa se puede calcular modificando uno de los algoritmos de concatenación como la concatenación de bucle anidado o la concatenación de bucle simple. Por ejemplo, para calcular una concatenación externa *izquierda*, utilizamos la relación de la izquierda en el bucle externo o bucle simple, ya que cada tupla de la relación de la izquierda debe aparecer en el resultado. Si existen tuplas emparejables en la otra relación, se generarán las tuplas unidas y se guardarán en el resultado. Sin embargo, si no se encuentra ninguna tupla emparejable, la tupla se sigue incluyendo en el resultado, aunque se rellena con valores NULL. Los algoritmos de ordenación-mezcla y de dispersión-concatenación también pueden ser ampliados para el cálculo de las concatenaciones externas.

Como alternativa, la concatenación externa también se puede calcular ejecutando una combinación de operadores de álgebra relacional. Por ejemplo, la operación de concatenación externa izquierda que se mostró anteriormente es equivalente a la siguiente secuencia de operaciones relacionales:

1. Calcular la concatenación (interna) de las tablas EMPLEADO y DEPARTAMENTO.

   $\text{TEMP1} \leftarrow \pi_{\text{Apellido1, Nombre, NombreDpto}} (\text{EMPLEADO} \bowtie_{\text{Dno=NúmeroDpto}} \text{DEPARTAMENTO})$

2. Encontrar las tuplas de EMPLEADO que no aparecen en el resultado de la concatenación (interna).

   $\text{TEMP2} \leftarrow \pi_{\text{Apellido1, Nombre}} (\text{EMPLEADO}) - \pi_{\text{Apellido1, Nombre}} (\text{TEMP1})$

3. Rellenar cada una de las tuplas de TEMP2 con un campo NombreDpto igual a NULL.

   TEMP2 ← TEMP2 × NULL

4. Aplicar la operación UNIÓN a TEMP1, TEMP2 para producir el resultado de la concatenación externa izquierda.

   RESULT ← TEMP1 ∪ TEMP2

El coste de la concatenación externa según el cálculo anterior podría ser la suma de los costes de los pasos asociados (concatenación interna, proyecciones, diferencia de conjuntos y unión). Sin embargo, observe que el paso 3 se puede ejecutar a medida que se construye la relación temporal en el paso 2, es decir, podemos limitarnos a rellenar cada tupla con un valor NULL. Además, en el paso 4, sabemos que los dos operandos de la unión son disjuntos (no existen tuplas en común), así que no es necesaria la eliminación de duplicados.

## 15.6 Combinación de operaciones mediante flujos

Una consulta especificada en SQL será traducida, por lo general, a una expresión en álgebra relacional que será una *secuencia de operaciones relacionales*. Si ejecutamos una única operación cada vez, debemos generar ficheros temporales en disco para guardar los resultados de estas operaciones temporales, apareciendo una sobrecarga de computación excesiva. La generación y el almacenamiento de grandes ficheros temporales en disco consume mucho tiempo y puede ser innecesaria en muchos casos, ya que estos ficheros serán utilizados de inmediato como entrada a la siguiente operación. Para reducir el número de ficheros temporales, es habitual la generación de código de ejecución de consultas que se corresponde con los algoritmos para combinar operaciones en una consulta. Por ejemplo, en lugar de implementarla por separado, una concatenación se puede combinar con dos operaciones SELECT sobre los ficheros de entrada y una última operación PROJECT sobre el fichero resultante; todo esto se implementa mediante un algoritmo con dos ficheros de entada y un único fichero de salida. En lugar de crear cuatro ficheros temporales, aplicamos el algoritmo directamente y obtendremos un solo fichero resultante. En la Sección 17.7.2, veremos cómo la optimización heurística del álgebra relacional puede agrupar operaciones para su ejecución, esto se denomina **procesamiento en secuencia o basado en flujos**.

Es habitual crear el código de ejecución de consultas de forma dinámica para implementar múltiples operaciones. El código generado para producir la consulta combina varios algoritmos que se corresponden con las operaciones individuales. A medida que se generan las tuplas resultantes de una operación, son suministradas como entrada a las operaciones siguientes. Por ejemplo, si a dos operaciones de selección en las relaciones base les sigue una operación de concatenación, las tuplas resultantes de cada selección se suministran como entrada al algoritmo de concatenación en un **flujo** o **secuencia** a medida que son generadas.

## 15.7 Utilización de la heurística en la optimización de consultas

En esta sección veremos técnicas de optimización que aplican reglas heurísticas para modificar la representación interna de una consulta (que se encuentra normalmente bajo la forma de un árbol de consultas o de una estructura de datos de consultas en grafo) para mejorar su ejecución. El analizador de una consulta de alto nivel genera en primer lugar una *representación interna inicial*, que es optimizada posteriormente según unas reglas heurísticas. Tras esto, se genera un plan de ejecución de consultas para ejecutar grupos de operaciones basadas en las rutas de acceso disponibles para los ficheros implicados en la consulta.

Una de las principales **reglas heurísticas** es aplicar las operaciones SELECT y PROJECT antes de aplicar JOIN u otras operaciones binarias, ya que el tamaño del fichero resultante de una operación binaria (como JOIN) es, por lo general, una función multiplicativa de los tamaños de los ficheros de entrada. Las operaciones SELECT y PROJECT reducen el tamaño de un fichero y, por tanto, deberían ser aplicadas *antes* de JOIN u otra operación binaria.

En la Sección 15.7.1 volveremos a hacer hincapié en las notaciones de árboles de consultas y de grafos de consultas que ya vimos anteriormente en el contexto del álgebra relacional y del cálculo en las Secciones 6.3.5 y 6.6.5, respectivamente. Éstas pueden ser utilizadas como base para las estructuras de datos que se usan para la representación interna de las consultas. Un árbol de consultas se utiliza para representar una expresión en álgebra relacional o en álgebra relacional extendida, mientras que un grafo de consultas se utiliza para representar una expresión de cálculo relacional. Posteriormente, en la Sección 15.7.2, mostraremos cómo se aplican las reglas de optimización heurísticas para convertir un árbol de consultas en un **árbol de consultas equivalente**, que representa una expresión de álgebra relacional distinta que se ejecuta de modo más eficiente proporcionando el mismo resultado que la original. También veremos la equivalencia entre distintas expresiones de álgebra relacional. Por último, en la Sección 15.7.3 veremos la generación de planes de ejecución de consultas.

## 15.7.1 Notación para los árboles de consultas y los grafos de consultas

Un **árbol de consultas** es una estructura de datos en árbol que equivale a una expresión de álgebra relacional. Representa las relaciones de entrada a una consulta como los *nodos hoja* del árbol y las operaciones del álgebra relacional como nodos internos. Una ejecución del árbol de consultas consiste en ejecutar una operación en un nodo interno siempre que estén disponibles sus operandos y después reemplazar ese nodo interno por la relación que resulta de la ejecución de la operación. La ejecución finaliza cuando se ejecuta el nodo raíz y se genera la relación resultante de la consulta.

En la Figura 15.4a se muestra un árbol de consultas (el mismo que aparece en la Figura 6.9) para la consulta C2 de los Capítulos 5 a 8: para cada proyecto localizado en 'Gijón', obtener el número de proyecto, el número de departamento controlador y el apellido del responsable del departamento, su dirección y su fecha de nacimiento. Esta consulta se especifica en el esquema relacional de la Figura 5.5 y equivale a la siguiente expresión en álgebra relacional:

$$\pi_{\text{NumProyecto, NumDptoProyecto, Apellido1, Direccion, FechaNac}}\ (((\sigma_{\text{UbicaciónProyecto}='\text{Gijón}'}(\text{PROYECTO}))\ \bowtie_{\text{NumDptoProyecto}=\text{NúmeroDpto}}(\text{DEPARTAMENTO}))\ \bowtie_{\text{DniDirector}=\text{Dni}}(\text{EMPLEADO}))$$

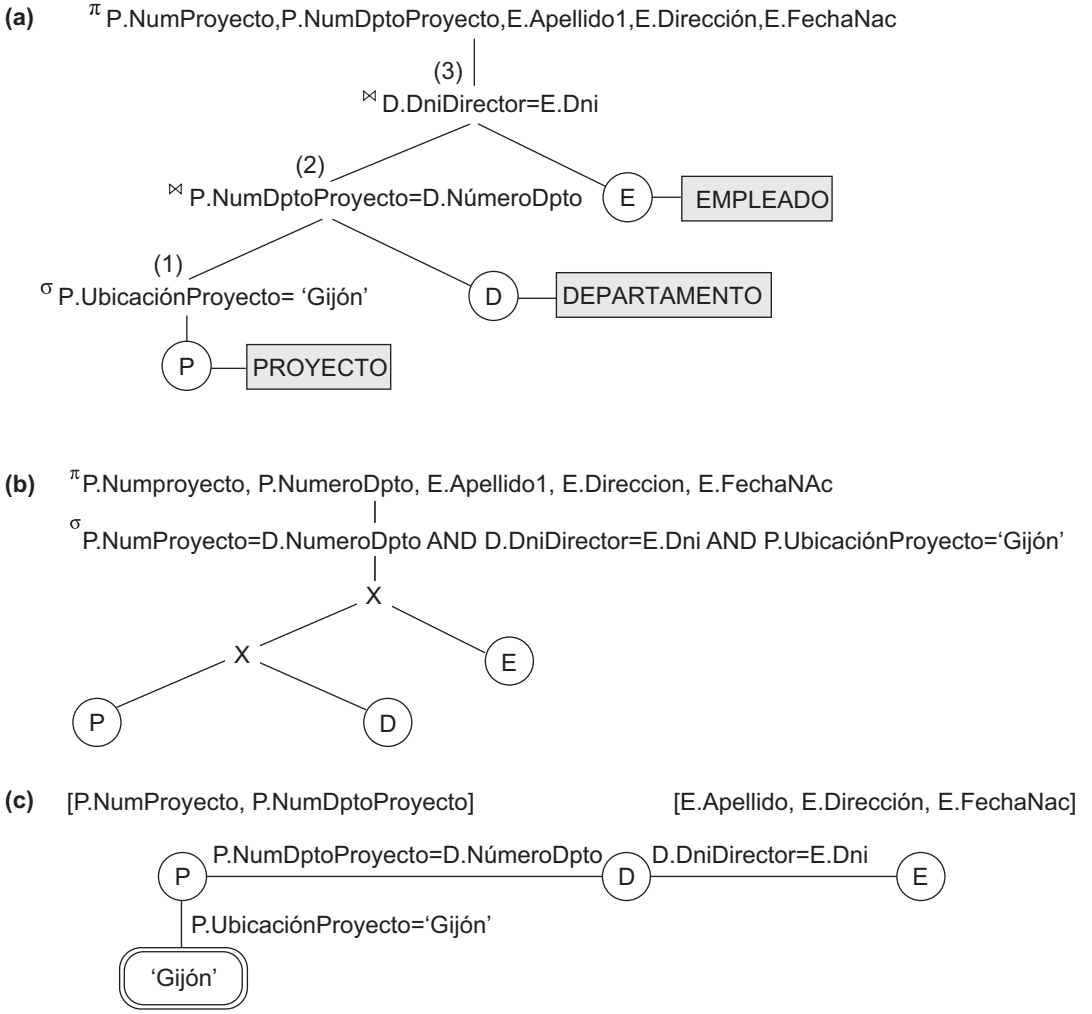
Esto equivale a la siguiente consulta en SQL:

```
C2:  SELECT   P.NumProyecto, P.NumDptoProyecto, E.Apellido1, E.Dirección, E.FechaNac
     FROM     PROYECTO AS P, DEPARTAMENTO AS D, EMPLEADO AS E
     WHERE    P.NumDptoProyecto=D.NúmeroDpto AND D.DniDirector=E.Dni AND
              P.UbicaciónProyecto= 'Gijón';
```

En la Figura 15.4a las tres relaciones PROYECTO, DEPARTAMENTO y EMPLEADO se representan como los nodos hoja P, D y E, mientras que las operaciones de álgebra relacional de la expresión se representan mediante nodos internos del árbol. Cuando se ejecuta esta consulta, el nodo marcado como (1) en la Figura 15.4a debe comenzar su ejecución antes que el nodo (2), ya que algunas de las tuplas resultantes de la operación (1) deben estar disponibles antes de que comience la ejecución de la operación (2). Del mismo modo, el nodo (2) debe comenzar su ejecución y generar los resultados antes de que el nodo (3) pueda comenzar su ejecución, y así sucesivamente.

Como puede observar, el árbol de consultas representa un orden determinado en las operaciones para la ejecución de una consulta. Una forma más neutral de representar una consulta es la notación en **grafo de**

**Figura 15.4.** Dos árboles de consultas para la consulta C2. (a) Árbol de consultas correspondiente a la expresión de álgebra relacional para C2. (b) Árbol de consultas inicial (canónico) para la consulta SQL C2. (c) Grafo de consultas para C2.




**consultas**. La Figura 15.4c (la misma que aparece en la Figura 6.13) muestra el grafo de consultas correspondiente a la consulta C2. Las relaciones de la consulta se representan mediante **nodos de relación**, que aparecen como círculos sencillos. Los valores constantes, que son por lo general las condiciones de selección de la consulta, se representan mediante **nodos de constantes**, que aparecen como círculos dobles u óvalos. Las condiciones de selección y de concatenación se representan mediante los **bordes** del grafo, según se muestra en la Figura 15.4c. Por último, los atributos a obtener de cada relación se muestran dentro de corchetes sobre cada relación.

La representación del grafo de consultas no indica un orden para ejecutar en primer lugar ciertas operaciones. Sólo existe un único grafo para cada consulta.[15] Aunque algunas técnicas de optimización se basaban en grafos de consultas, hoy en día se prefieren, por lo general, los árboles de consultas, ya que en la práctica el

[15] Según esto, un árbol de consultas equivale a una expresión de *cálculo relacional* según se muestra en la Sección 6.6.5.

optimizador de consultas necesita conocer el orden de las operaciones para la ejecución de la consulta, lo cual no es posible en los grafos de consultas.

### 15.7.2 Optimización heurística de los árboles de consultas

Por lo general, muchas expresiones en álgebra relacional (y, por tanto, muchos árboles de consultas distintos) son equivalentes; esto es, equivalen a la misma consulta.[16]

El analizador de consultas generará, normalmente, un **árbol de consultas inicial** estándar que equivalga a una consulta SQL, sin hacer ninguna optimización. Por ejemplo, el árbol inicial para una consulta SELECT-PROJECT-JOIN como C2 se muestra en la Figura 15.4(b). La operación CARTESIAN PRODUCT de las relaciones que aparece en la cláusula FROM es lo primero que se aplica; después se realizarán las condiciones de selección y concatenación de la cláusula WHERE, seguidas de la proyección sobre los atributos de la cláusula SELECT. Un árbol de consultas canónico de este tipo representa una expresión de álgebra relacional que es *muy ineficaz si se ejecuta directamente*, debido a las operaciones CARTESIAN PRODUCT ($\times$). Por ejemplo, si las relaciones PROYECTO, DEPARTAMENTO y EMPLEADO tuviesen tamaños de registro de 100, 50 y 150 bytes y contuviesen 100, 20 y 5.000 tuplas respectivamente, el resultado de la operación CARTESIAN PRODUCT contendría 10 millones de tuplas con un tamaño de registro igual a 300 bytes cada una. Sin embargo, el árbol de consultas de la Figura 15.4(b) se encuentra en una forma estándar simple que puede ser creada fácilmente. Deberá ser el optimizador heurístico de consultas el que transforme este árbol de consultas inicial en un **árbol de consultas final** de ejecución eficiente.

El optimizador debe incluir reglas de equivalencia entre las expresiones de álgebra relacional que puedan ser aplicadas al árbol inicial. Las reglas de optimización heurística de consultas utilizarán estas expresiones de equivalencia para transformar el árbol inicial en el árbol de consultas final optimizado. En primer lugar, veremos de manera informal cómo se transforma un árbol de consultas mediante el uso de la heurística. Posteriormente, veremos reglas generales de transformación y cómo se pueden utilizar en un optimizador algebraico heurístico.

**Ejemplo de transformación de una consulta.** Tomemos la siguiente consulta C de la base de datos de la Figura 5.5. *Encontrar los apellidos de los empleados nacidos después de 1957 que trabajan en un proyecto llamado 'Aquarius'*. Esta consulta se puede especificar en SQL como:

```
C:  SELECT   Apellido1
    FROM     EMPLEADO,TRABAJA_EN, PROYECTO
    WHERE    NombreProyecto='Aquarius' AND NumProyecto=Pno AND DniEmpleado=Dni
             AND FechaNac > '1957-12-31';
```

El árbol de consultas inicial para C se muestra en la Figura 15.5(a). La ejecución de este árbol directamente crea un fichero de gran tamaño que contiene el resultado de la operación CARTESIAN PRODUCT del total de los ficheros EMPLEADO, TRABAJA_EN y PROYECTO. Sin embargo, esta consulta necesita un solo registro de la relación PROYECTO (el que corresponde al proyecto 'Aquarius') y sólo los registros de EMPLEADO de aquellos cuya fecha de nacimiento es posterior a '31-12-1957'. La Figura 15.5(b) muestra un árbol de consultas mejorado que aplica en primer lugar las operaciones SELECT para reducir el número de tuplas que aparecen en la operación CARTESIAN PRODUCT.

Una mejora posterior se podría realizar conmutando las posiciones de las relaciones EMPLEADO y PROYECTO en el árbol, según se muestra en la Figura 15.5(c). Haciendo esto, aprovechamos que sabemos que NumProyecto es un atributo clave de la relación de proyectos y, por tanto, la operación SELECT sobre la relación PROYECTO devolverá sólo un único registro. Podemos mejorar aún más el árbol de consultas si susti-

[16] La misma consulta se puede declarar de distintas formas en un lenguaje de alto nivel como SQL (consulte el Capítulo 8).

**Figura 15.5.** Pasos para convertir un árbol de consultas durante la optimización heurística.
(a) Árbol de consultas inicial (canónico) para la consulta SQL C.
(b) Desplazar las operaciones SELECT hacia abajo por el árbol de consultas.
(c) Ejecutar en primer lugar las operaciones SELECT más restrictivas.

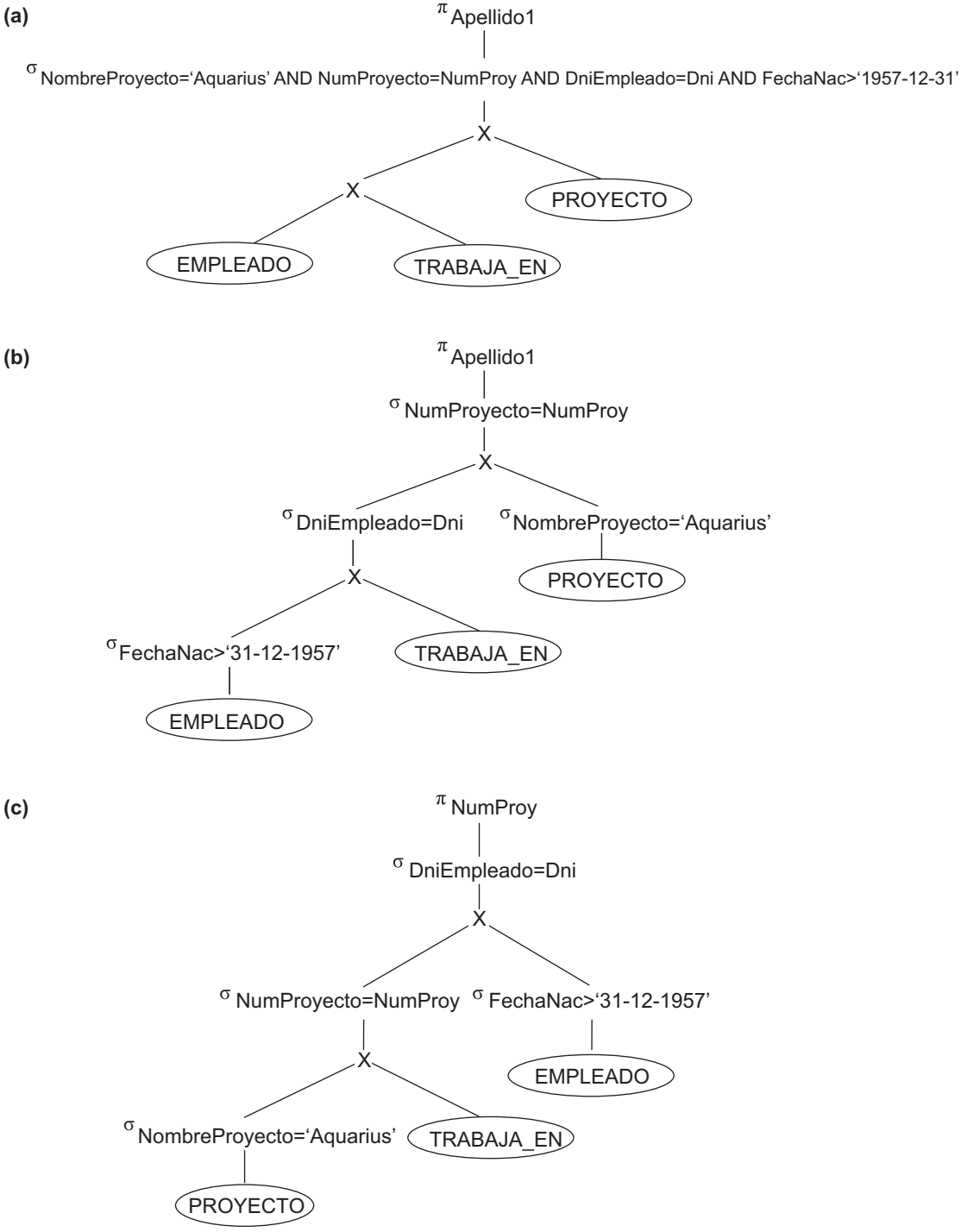
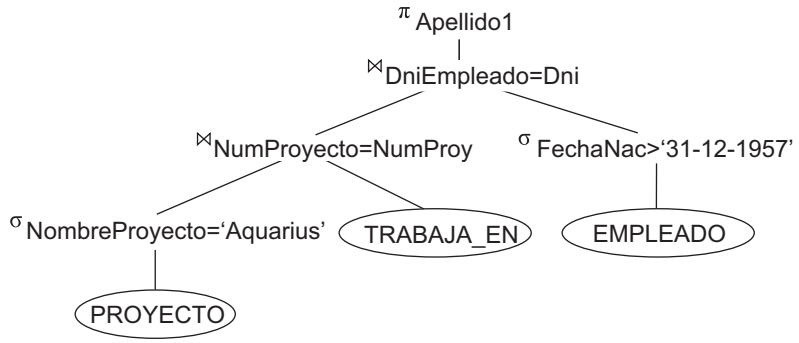
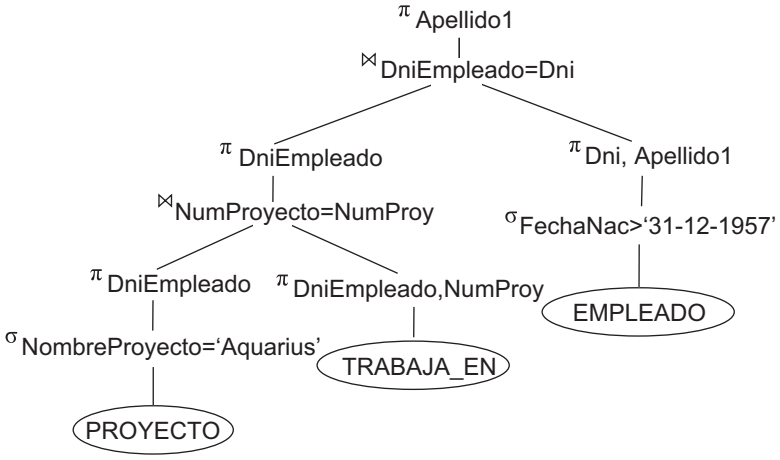


**Figura 15.5.** (*Continuación*) Pasos para convertir un árbol de consultas durante la optimización heurística.
(d) Reemplazar CARTESIAN PRODUCT y SELECT con operaciones JOIN.
(e) Desplazar las operaciones PROJECT hacia abajo por el árbol de consultas.

**(d)**




**(e)**




tuimos cualquier operación del tipo CARTESIAN PRODUCT que esté seguida de una condición de concatenación por una operación JOIN, según aparece en la Figura 15.5(d). Otra mejora sería guardar en las relaciones intermedias sólo los atributos que se necesitan en las operaciones posteriores, incluyendo operaciones PROJECT ($\pi$) tan pronto como sea posible en el árbol de consultas, según se muestra en la Figura 15.5(e). Esto reduce el número de atributos (columnas) de las relaciones intermedias, a diferencia de las operaciones SELECT en las cuales se reduce el número de tuplas (registros).

Según se demostró en el ejemplo anterior, un árbol de consultas se puede transformar paso a paso en un árbol de consultas diferente cuya ejecución sea más eficiente. Sin embargo, nos debemos asegurar de que los pasos de transformación siempre conducen a un árbol de consultas equivalente.

**Reglas generales de transformación para las operaciones de álgebra relacional**. Existen muchas reglas para transformar las operaciones de álgebra relacional en expresiones equivalentes. Aquí nos inte-

resa el significado de las operaciones y de las relaciones resultantes. Según esto, si dos relaciones tienen el mismo conjunto de atributos en *diferente orden* pero representan la misma información, las consideraremos equivalentes. En la Sección 5.1.2 dimos una definición alternativa de *relación* que hace que el orden de los atributos no sea importante; utilizaremos esta definición en este punto. Mostraremos, sin demostrarlas, algunas reglas de transformación que son útiles en la optimización de consultas:

1. **Cascada de σ.** Una condición de selección conjuntiva puede ser dividida en una cascada (es decir, una secuencia) de operaciones σ individuales:

   $\sigma_{c1 \text{ AND } c2 \text{ AND } \ldots \text{ AND } cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R))\ldots))$

2. **Commutatividad de σ.** La operación σ es conmutativa:

   $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$

3. **Cascada de π**. En una cascada (secuencia) de operaciones π, todas, excepto la última, pueden ser ignoradas:

   $\pi_{\text{Lista1}}(\pi_{\text{Lista2}}(\ldots(\pi_{\text{Listan}}(R))\ldots)) \equiv \pi_{\text{Lista1}}(R)$

4. **Conmutación de σ por π.** Si la condición $c$ de selección incluye sólo los atributos $A_1, \ldots, A_n$ en la lista de proyección, las dos operaciones pueden ser conmutadas:

   $\pi_{A1, A2, \ldots, An}(\sigma_c(R)) \equiv \sigma_c(\pi_{A1, A2, \ldots, An}(R))$

5. **Conmutatividad de ⋈ (y ×).** La operación ⋈ es conmutativa, al igual que la operación ×:

   $R \bowtie_c S \equiv S \bowtie_c R$

   $R \times S \equiv S \times R$

   Observe que aunque puede que el orden de los atributos no sea el mismo en las relaciones resultantes de las dos concatenaciones (o de los dos productos cartesianos), el *significado* es el mismo, ya que el orden de los atributos no es importante según la definición alternativa de relación.

6. **Conmutación de σ con ⋈ (o ×).** Si todos los atributos de la condición de selección $c$ involucran sólo a los atributos de una de las relaciones que están siendo unidas (por ejemplo, $R$), las dos operaciones pueden ser conmutadas entre sí de este modo:

   $\sigma_c(R \bowtie S) \equiv (\sigma_c(R)) \bowtie S$

   Como alternativa, si la condición $c$ de selección se puede escribir como ($c_1$ AND $c_2$), donde la condición $c_1$ incluye sólo los atributos de $R$ y la condición $c2$ incluye sólo los atributos de $S$, las operaciones pueden ser conmutadas entre sí de este modo:

   $\sigma_c(R \bowtie S) \equiv (\sigma_{c1}(R)) \bowtie (\sigma_{c2}(S))$

   Las mismas reglas se aplican si se reemplaza ⋈ por una operación ×.

7. **Conmutación de π con ⋈ (o ×).** Supongamos que la lista de proyección es $L = \{A_1, \ldots, A_n, B_1, \ldots, B_m\}$, donde $A_1, \ldots, A_n$ son atributos de $R$ y $B_1, \ldots, B_m$ son atributos de $S$. Si la condición de concatenación $c$ incluye sólo los atributos de $L$, las dos operaciones pueden ser conmutadas entre sí de este modo:

   $\pi_L(R \bowtie_c S) \equiv (\pi_{A1, \ldots, An}(R)) \bowtie_c (\pi_{B1, \ldots, Bm}(S))$

   Si la condición de concatenación $c$ contiene atributos adicionales que no están en $L$, éstos deben ser agregados a la lista de proyección, y se necesita una última operación π. Por ejemplo, si los atributos $A_{n+1}, \ldots, A_{n+k}$ de $R$ y $B_{m+1}, \ldots, B_{m+p}$ de $S$ están incluidos en la condición de concatenación $c$ pero no lo están en la lista de proyección $L$, las operaciones pueden ser conmutadas entre sí de este modo:

   $\pi_L(R \bowtie_c S) \equiv \pi_L((\pi_{A1, \ldots, An, An+1, \ldots, An+k}(R)) \bowtie_c (\pi_{B1, \ldots, Bm, Bm+1, \ldots, Bm+p}(S)))$

   Para ×, no existe la condición $c$, por tanto la primera regla de transformación siempre se aplica reemplazando $\bowtie_c$ por ×.

8. **Conmutatividad de las operaciones de conjunto.** Las operaciones de conjunto ∪ y ∩ son conmutativas pero – no lo es.
9. **Asociatividad de ⋈, ×, ∪ y ∩.** Estas cuatro operaciones son asociativas de forma individual; es decir, si θ representa a cualquiera de estas cuatro operaciones (a lo largo de toda la expresión), tendremos:
   $(R\ \theta\ S)\ \theta\ T \equiv R\ \theta\ (S\ \theta\ T)$
10. **Conmutación de σ con las operaciones de conjunto.** La operación σ se puede conmutar con ∪, ∩ y −. Si θ representa a cualquiera de estas tres (a lo largo de toda la expresión), tendremos:
    $\sigma_c\ (R\ \theta\ S) \equiv (\sigma_c\ (R))\ \theta\ (\sigma_c\ (S))$
11. **La operación π se puede conmutar con ∪.**
    $\pi_L\ (R \cup S) \equiv (\pi_L\ (R)) \cup (\pi_L\ (S))$
12. **Conversión de una secuencia (σ, ×) en ⋈.** Si la condición *c* de una operación σ que sigue a una operación × corresponde a una operación de concatenación, convierte la secuencia (σ, ×) en una ⋈ de este modo:
    $(\sigma_c\ (R \times S)) \equiv (R \bowtie_c S)$

Existe otro tipo de transformaciones. Por ejemplo, una condición *c* de selección o de concatenación se puede convertir en una condición equivalente utilizando las siguientes reglas (leyes de DeMorgan):

**NOT** ($c_1$ **AND** $c_2$) ≡ (**NOT** $c_1$) **OR** (**NOT** $c_2$)
**NOT** ($c_1$ **OR** $c_2$) ≡ (**NOT** $c_1$) **AND** (**NOT** $c_2$)

No repetiremos aquí las transformaciones adicionales que se vieron en los Capítulos 5 y 6. A continuación veremos cómo se pueden utilizar las transformaciones en la optimización heurística.

**Descripción de un algoritmo de optimización algebraica heurística.** En este momento podemos describir los pasos de un algoritmo que utiliza algunas de las reglas descritas anteriormente para transformar un árbol de consultas inicial en un árbol optimizado que se ejecute de forma más eficiente (en la mayoría de los casos). El algoritmo nos conducirá a transformaciones similares a las vistas en nuestro ejemplo de la Figura 15.5. Los pasos del algoritmo son los que se muestran a continuación:

1. Utilizando la regla 1, descomponer cualquier operación SELECT con condiciones conjuntivas en una cascada de operaciones SELECT. Esto permite un mayor grado de libertad para mover las operaciones SELECT hacia abajo por las distintas ramas del árbol.
2. Utilizando las reglas 2, 4, 6 y 10 relativas a la conmutatividad de SELECT con otras operaciones, desplazar cada operación SELECT hacia abajo por el árbol tan lejos como lo permitan los atributos incluidos en la condición de selección.
3. Utilizando las reglas 5 y 9 relativas a la conmutatividad y asociación de las operaciones binarias, reordenar las relaciones de los nodos hoja utilizando los siguientes criterios. En primer lugar, posicionar las relaciones de los nodos hoja con las operaciones SELECT más restrictivas de forma que sean ejecutadas en primer lugar en la representación del árbol de consultas. La definición de SELECT *más restrictiva* se refiere a las que generan una relación bien con el menor número de tuplas o bien con el menor tamaño absoluto.[17] Otra posibilidad es definir la SELECT más restrictiva como aquella que tiene una selectividad más pequeña; esto resulta más práctico ya que a veces los cálculos de selectividades se encuentran dentro del catálogo del DBMS. En segundo lugar, asegurarse de que la ordenación de los nodos hoja no produce ninguna operación CARTESIAN PRODUCT; por ejemplo, si las dos relacio-

[17] Se puede usar cualquier definición, ya que estas reglas son heurísticas.

nes con la SELECT más restrictiva no tienen una condición de concatenación directa entre ellas, puede ser conveniente cambiar el orden de los nodos hoja para evitar productos cartesianos.[18]

4. Utilizando la regla 12, combinar una operación CARTESIAN PRODUCT con la siguiente operación SELECT del árbol para formar una operación JOIN, en el caso de que la condición represente una condición de concatenación.
5. Utilizando las reglas 3, 4, 7 y 11 relativas a la secuenciación de PROJECT y a la conmutación de PROJECT con otras operaciones, descomponer y desplazar las listas de atributos de proyección hacia abajo por el árbol lo más lejos posible mediante la creación de nuevas operaciones PROJECT según sea necesario. Tras cada operación PROJECT sólo se deberían guardar aquellos atributos necesarios en el resultado de la consulta y en las operaciones siguientes.
6. Identificar los subárboles que representan grupos de operaciones que se pueden ejecutar mediante un único algoritmo.

En nuestro ejemplo, la Figura 15.5(b) muestra el árbol de la Figura 15.5(a) después de haber aplicado los pasos 1 y 2 del algoritmo; la Figura 15.5(c) muestra el árbol tras el paso 3; la Figura 15.5(d) tras el paso 4; y la Figura 15.5(e) tras el paso 5. En el paso 6 podemos agrupar las operaciones en el árbol cuya raíz es la operación $\pi_{\text{DniEmpleado}}$, ya que el primer agrupamiento significa que este subárbol se ejecuta en primer lugar.

**Resumen de las reglas heurísticas para la optimización algebraica.** Resumiremos ahora las reglas heurísticas básicas para la optimización algebraica. La principal regla heurística es aplicar en primer lugar las operaciones que reducen el tamaño de los resultados intermedios. Eso significa ejecutar tan pronto como sea posible las operaciones SELECT para reducir el número de tuplas y las operaciones PROJECT para reducir el número de atributos. Esto se lleva a cabo desplazando las operaciones SELECT y PROJECT hacia abajo en el árbol lo más lejos posible. Además, las operaciones SELECT y JOIN más restrictivas (es decir, con las relaciones resultantes con el menor número de tuplas o con el tamaño absoluto menor) deberían ser ejecutadas antes que otras operaciones similares. Esto se hace reordenando los nodos hoja del árbol entre sí evitando los productos cartesianos y ajustando el resto del árbol adecuadamente.

### 15.7.3 Conversión de árboles de consulta en planes de ejecución de consultas

Un plan de ejecución para una expresión de álgebra relacional representada en forma de árbol de consultas incluye información acerca de los métodos de acceso disponibles para cada relación y acerca de los algoritmos a utilizar en el cálculo de los operadores relacionales representados en el árbol. Como ejemplo sencillo, tomemos la consulta C1 del Capítulo 5, cuya expresión en álgebra relacional es:

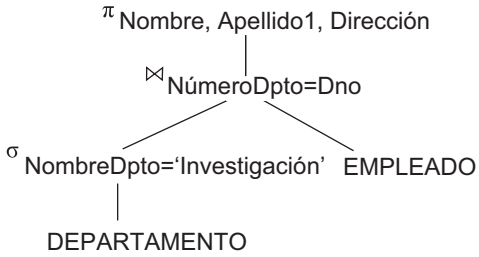
$$\pi_{\text{Nombre, Apellido1, Dirección}}(\sigma_{\text{NombreDpto}='\text{Investigación}'}(\text{DEPARTAMENTO}) \bowtie_{\text{NúmeroDpto}=\text{Dno}} \text{EMPLEADO})$$

El árbol de consultas se muestra en la Figura 15.6. Para convertirlo en un plan de ejecución, el optimizador podría elegir una búsqueda en índice para la operación SELECT (suponiendo que exista alguno), una exploración en la tabla como método de acceso en EMPLEADO, un algoritmo de concatenación de bucle anidado para la concatenación y una exploración del resultado de JOIN para la operación PROJECT. Además, el modelo elegido para ejecutar la consulta podría especificar una evaluación en secuencia o una evaluación materializada.

Mediante la **evaluación materializada**, el resultado de una operación se almacena como relación temporal (es decir, el resultado queda *materializado físicamente*). Por ejemplo, se puede calcular la operación JOIN y el resultado en su totalidad ser almacenado en forma de relación temporal, que será leída posteriormente como

[18] Observe que una operación CARTESIAN PRODUCT es aceptable en algunos casos; por ejemplo, si las relaciones sólo tienen una única tupla porque cada una de ellas se sometieron a una condición de selección sobre un campo clave.

**Figura 15.6.** Un árbol de consultas para la consulta C1.




entrada al algoritmo que calcula la operación PROJECT, que sería la que generase la tabla resultante de la consulta. Por otra parte, en el caso de la **evaluación en secuencia**, a medida que se generan las tuplas resultantes de una operación, serán enviadas directamente a la siguiente operación en la secuencia de la consulta. Por ejemplo, a medida que la operación SELECT genera las tuplas seleccionadas en DEPARTAMENTO, éstas son puestas en un búfer; a continuación, la operación JOIN extraerá las tuplas del búfer y las tuplas resultantes de la operación JOIN serán enviadas en secuencia al algoritmo de la operación de proyección. La ventaja de la secuenciación es el ahorro de costes al no tener que escribir los resultados intermedios en disco y no tener que leerlos nuevamente en la siguiente operación.

## 15.8 Utilización de la selectividad y la estimación de costes en la optimización de consultas

Un optimizador de consultas no debería basarse exclusivamente en reglas heurísticas; también debería calcular y comparar los costes de ejecución de una consulta utilizando diferentes estrategias de ejecución y debería elegir la estrategia con el *menor coste estimado*. Para que funcione este modelo se necesitan estimaciones de coste precisas para que la comparación entre las diferentes estrategias se realice con ecuanimidad y de forma real. Además, debemos limitar el número de estrategias a tener en cuenta ya que, en caso contrario, se perdería demasiado tiempo haciendo estimaciones de coste para un número tan elevado de estrategias de ejecución posibles. De acuerdo con esto, este modelo es más adecuado en **consultas compiladas** en las cuales la optimización se realiza en tiempo de compilación, y el código para la estrategia de ejecución resultante se almacena y se ejecuta directamente en tiempo de ejecución. En el caso de las **consultas interpretadas** en las cuales todo el proceso, mostrado en la Figura 15.1, se realiza en tiempo de ejecución, una optimización a gran escala podría ralentizar el tiempo de respuesta. La optimización más elaborada está indicada en el caso de las consultas compiladas, mientras que una optimización parcial que requiera menos tiempo funciona mejor en el caso de las consultas interpretadas.

A este modelo se le denomina **optimización de consultas basada en costes**.[19] Utiliza técnicas de optimización tradicionales que buscan en el *espacio de soluciones* de un problema la solución que minimice una función objetivo (función de coste). Las funciones de coste que se utilizan en la optimización de consultas son funciones estimadas, no funciones exactas; por tanto, la optimización podría elegir una estrategia de ejecución de consultas que no fuese la óptima. En la Sección 15.8.1 veremos los componentes del coste de ejecución de una consulta. En la Sección 15.8.2 el tipo de información necesaria en las funciones de coste. Esta información se guarda en el catálogo del DBMS. En la Sección 15.8.3 pondremos ejemplos de funciones de coste para la operación SELECT y en la Sección 15.8.4 veremos funciones de coste para operaciones JOIN de dos vías. La Sección 15.8.5 está dedicada a las concatenaciones multivía y la Sección 15.8.6 muestra un ejemplo.

[19] Este modelo se utilizó por primera vez en el SYSTEM R, el DBMS experimental desarrollado en IBM (Selinger y otros, 1979).

### 15.8.1 Componentes de coste de ejecución de una consulta

El coste de ejecución de una consulta incluye los siguientes componentes:

1. **Coste de acceso al almacenamiento secundario.** Es el coste de la búsqueda, lectura y escritura de bloques de datos que residen en almacenamiento secundario, principalmente en disco. El coste de la búsqueda de registros en un fichero depende del tipo de las estructuras de acceso a dicho fichero, como, por ejemplo, la ordenación, la dispersión y los índices primarios y secundarios. Aparte de esto, factores como la ubicación contigua de los bloques del fichero en el mismo cilindro del disco o diseminados por el disco afectan al coste de acceso.
2. **Coste de almacenamiento.** Es el coste de almacenamiento de los ficheros intermedios generados por una estrategia de ejecución de la consulta.
3. **Coste computacional.** Es el coste de la ejecución de operaciones en memoria sobre los búferes de datos durante la ejecución de la consulta. Este tipo de operaciones incluye la búsqueda y la ordenación de los registros, la mezcla de registros durante una concatenación y la ejecución de cálculos sobre valores de los campos.
4. **Coste de uso de memoria.** Es el coste relativo al número de búferes de memoria que se necesitan durante la ejecución de la consulta.
5. **Coste de comunicaciones.** Es el coste del envío de la consulta y de sus resultados desde el sitio donde se ubica la base de datos hasta el sitio o el terminal donde se originó la consulta.

En el caso de bases de datos de gran tamaño, el objetivo principal es minimizar el coste de acceso al almacenamiento secundario. Las funciones de coste sencillas ignoran otro tipo de factores y comparan las diferentes estrategias de ejecución en función del número de transferencias de bloque entre el disco y la memoria principal. En el caso de bases de datos de menor tamaño, en las que se puede almacenar en memoria la mayoría de los datos de los ficheros implicados en la consulta, el objetivo principal es minimizar el coste computacional. En el caso de bases de datos distribuidas en las que se encuentran implicadas varias ubicaciones físicas (consulte el Capítulo 25), también se debe minimizar el coste de las comunicaciones. Resulta difícil incluir todos los componentes de coste en una función (ponderada) de coste debido a la dificultad de asignar los pesos adecuados a los componentes de coste. Ésta es la razón por la que algunas funciones de coste tienen en cuenta un único factor: el acceso a disco. En la siguiente sección veremos la información que se necesita para formular las funciones de coste.

### 15.8.2 Información del catálogo utilizada en las funciones de coste

Para calcular los costes de las distintas estrategias de ejecución debemos tener en cuenta la información que se necesita en las funciones de coste. Esta información se puede almacenar en el catálogo del DBMS, donde es accedida por el optimizador de consultas. En primer lugar, debemos conocer el tamaño de cada fichero. En el caso de un fichero cuyos registros son todos del mismo tipo, necesitamos **el número de registros (tuplas) (*r*)**, el **tamaño** (medio) **de registro (*R*)**, y el **número de bloques (*b*)** (o sus valores aproximados). Es posible que también necesitemos el **factor de bloqueo (*bfr*)**. También debemos tener en cuenta el *método de acceso primario* y los *atributos de acceso primario* de cada fichero. Los registros del fichero pueden estar desordenados, ordenados por un atributo con o sin un índice primario o agrupado, o mezclados según un atributo clave. Se guarda información de todos los índices secundarios y de los atributos de indexación. Tenemos que conocer el **número de niveles (*x*)** de cada índice multinivel (primario, secundario o agrupado) para las funciones de coste que calculan el número de accesos a bloque que se producen durante la ejecución de la consulta. En algunas funciones de coste se necesita el **número de bloques de índices de primer nivel ($b_{I1}$)**.

Otro parámetro importante es el **número de valores distintos (*d*)** de un atributo y su **selectividad (*sl*)**, que es la porción de registros que satisfacen una condición de igualdad sobre el atributo. Esto permite el cálculo de

la **cardinalidad de selección ($s = sl * r$)** de un atributo, que es el número *medio* de registros que cumplirán una condición de igualdad sobre ese atributo. En el caso de un *atributo clave, d = r; sl* = 1/*r* y *s* = 1. En el caso de un *atributo no clave*, si suponemos que los *d* valores distintos se encuentran distribuidos uniformemente por los registros, el cálculo será $sl = (1/d)$ y, por tanto, $s = (r/d)$.[20]

Cierta información, como el número de niveles de índice, es fácil de mantener ya que no varía con frecuencia. Sin embargo, otro tipo de información puede cambiar con frecuencia; por ejemplo, el número de registros *r* de un fichero cambia cada vez que se inserta o se elimina un registro. El optimizador de consultas necesitará estos valores con cierta exactitud, para utilizarlos en el cálculo del coste de las distintas estrategias. En las dos secciones siguientes revisaremos cómo se utilizan algunos de estos parámetros en las funciones de coste para un optimizador de consultas basado en costes.

### 15.8.3 Ejemplos de funciones de coste en SELECT

Veremos ahora las funciones de coste de los algoritmos de selección S1 a S8 de la Sección 15.3.1 en términos de *número de transferencias de bloque* entre la memoria y el disco. Estas funciones de coste son operaciones de cálculo que ignoran el tiempo de computación, el coste de almacenamiento y otros factores. El coste del método Si se podría referenciar como $C_{Si}$ accesos a bloque.

- **S1—Modelo de búsqueda lineal (fuerza bruta).** Buscaremos por todos los registros del fichero para extraer todos los registros que satisfacen la condición de selección; según esto, $C_{S1a} = b$. En el caso de una condición de igualdad sobre una clave, por término medio sólo se buscará en la mitad de los bloques del fichero antes de encontrar el registro, por tanto $C_{S1b} = (b/2)$ si se localiza el registro y $C_{S1b} = b$ si ningún registro satisface la condición.
- **S2—Búsqueda binaria.** Esta búsqueda accede aproximadamente a $C_{S2} = \log_2 b + \lceil (s/bfr) \rceil - 1$ bloques del fichero. Esto se reduce a $\log_2 b$ si la condición de igualdad se realiza sobre un atributo clave único, ya que $s = 1$ en este caso.
- **S3—Uso de un índice primario (S3*a*) o de una clave de dispersión (S3*b*) para obtener un único registro.** Con un índice primario, se obtiene un bloque más que el número de niveles de índice; por tanto, $C_{S3a} = x + 1$. En el caso de una clave de dispersión, la función de coste es aproximadamente $C_{S3b} = 1$ si se trata de dispersión estática o lineal, y 2 en el caso de dispersión extensible (consulte el Capítulo 13).
- **S4—Uso de un índice de ordenación para obtener varios registros.** Si la condición de comparación es >, >=, <, o <= sobre un campo clave con un índice de ordenación, apenas cumplirán la condición la mitad de los registros. Esto nos da una función de coste $C_{S4} = x + (b/2)$. Es un cálculo poco preciso y, aunque puede que por lo general sea correcto, podría llegar a ser bastante inexacto en determinados casos.
- **S5—Uso de un índice agrupado para obtener varios registros.** Dada una condición de igualdad, *s* registros cumplirán la condición, siendo *s* la cardinalidad de selección del atributo de indexación. Esto significa que $\lceil (s/bfr) \rceil$ bloques de fichero serán accedidos, resultando que $C_{S5} = x + \lceil (s/bfr) \rceil$.
- **S6—Uso de un índice secundario (árbol B+).** En una condición de *igualdad*, *s* registros cumplirán la condición, siendo *s* la cardinalidad de selección del atributo de indexación. Sin embargo, debido a que no se trata de un índice agrupado, cada uno de los registros puede estar localizado en un bloque distinto, por tanto el cálculo de coste (en el caso peor) es $C_{S6a} = x + s$. Esto se reduce a $x + 1$ para un atributo de indexación clave. Si la condición de comparación es >, >=, <, o <= y se supone que la mitad de los registros cumplen la condición, entonces (a grandes rasgos) se accede a la mitad de los

[20] Algunos optimizadores más precisos almacenan histogramas de la distribución de registros sobre los valores de los datos para un atributo.

bloques de índice de primer nivel y a la otra mitad se accede a través del índice. El cálculo de coste para este caso es, aproximadamente, $C_{S6b} = x + (b_{I1}/2) + (r/2)$. El factor $r/2$ se puede refinar si se dispone de mejores cálculos de selectividad.

- **S7—Selección conjuntiva.** Podemos utilizar S1 o uno de los métodos S2 a S6 vistos anteriormente. En el segundo caso, usamos una condición para obtener los registros y, posteriormente, comprobar en el búfer de memoria si cada registro obtenido cumple las condiciones de la conjunción restantes.
- **S8—Selección conjuntiva utilizando un índice compuesto.** Igual que en S3*a*, S5, o S6*a*, dependiendo del tipo de índice.

**Ejemplo de utilización de las funciones de coste.** En un optimizador de consultas, es habitual disponer de varias estrategias posibles de ejecución de una consulta y calcular los costes de diferentes estrategias. Podemos utilizar una técnica de optimización como la programación dinámica para encontrar el cálculo de coste óptimo (el menor) de forma eficiente sin tener que considerar todas las posibles estrategias de ejecución. Aquí no revisaremos los algoritmos de optimización, sino que usaremos un ejemplo sencillo para describir cómo se pueden utilizar los cálculos de coste. Supongamos que el fichero EMPLEADO de la Figura 5.5 tiene $r_E = 10.000$ registros almacenados en $b_E = 2.000$ bloques de disco con factor de bloqueo $bfr_E = 5$ registros/bloque y con las siguientes rutas de acceso:

1. Un índice agrupado sobre Sueldo, con niveles $x_{\text{Sueldo}} = 3$ y una cardinalidad media de selección $s_{\text{Sueldo}} = 20$.
2. Un índice secundario sobre el atributo clave Dni, con $x_{\text{Dni}} = 4$ ($s_{\text{Dni}} = 1$).
3. Un índice secundario sobre el atributo no clave Dno, con $x_{\text{Dno}} = 2$ y bloques de índices de primer nivel $b_{I1\text{Dno}} = 4$. Existen $d_{\text{Dno}} = 125$ valores distintos para Dno; por tanto, la cardinalidad de selección de Dno es $s_{\text{Dno}} = (r_E/d_{\text{Dno}}) = 80$.
4. Un índice secundario sobre Sexo, con $x_{\text{Sexo}} = 1$. Existen $d_{\text{Sexo}} = 2$ valores para el atributo Sexo; por tanto, la cardinalidad de selección media es $s_{\text{Sexo}} = (r_E/d_{\text{Sexo}}) = 5.000$.

Describiremos el uso de las funciones de coste con los siguientes ejemplos:

OP1: $\sigma_{\text{Dni='123456789'}}$(EMPLEADO)
OP2: $\sigma_{\text{Dno>5}}$(EMPLEADO)
OP3: $\sigma_{\text{Dno=5}}$(EMPLEADO)
OP4: $\sigma_{\text{Dno=5 AND SUELDO>30000 AND Sexo='M'}}$(EMPLEADO)

El coste de la opción S1 de fuerza bruta (búsqueda lineal) se calculará como $C_{S1a} = b_E = 2.000$ (para una selección sobre un atributo no clave) o $C_{S1b} = (b_E/2) = 1000$ (coste medio para una selección sobre un atributo clave). Para OP1 podemos utilizar o el método S1 o el método S6*a*; el cálculo de coste para S6a es $C_{S6a} = x_{\text{Dni}} + 1 = 4 + 1 = 5$, y será el elegido en lugar del método S1, cuyo coste medio es $C_{S1b} = 1.000$. Para OP2 podemos utilizar o bien el método S1 (con coste estimado $C_{S1a} = 2.000$) o el método S6b (con coste estimado $C_{S6b} = x_{\text{Dno}} + (bI_{1\text{Dno}}/2) + (r_E/2) = 2 + (4/2) + (10.000/2) = 5.004$); por tanto, elegiremos el modelo de fuerza bruta para OP2. Para OP3 podemos utilizar el método S1 (con coste estimado $CS1a = 2.000$) o el método S6*a* (con coste estimado $C_{S6a} = x_{\text{Dno}} + s_{\text{Dno}} = 2 + 80 = 82$), así que elegiremos el método S6*a*.

Por último, veamos OP4, con una condición de selección conjuntiva. Necesitamos calcular el coste de utilizar cualquiera de los tres componentes de la condición de selección para obtener los registros, más el modelo de fuerza bruta. Este último tiene un coste estimado de $C_{S1a} = 2000$. Si tomamos en primer lugar la condición (Dno = 5) tendremos una estimación de coste $C_{S6a} = 82$. Si tomamos en primer lugar la condición (Sueldo > 30.000) tendremos un coste estimado de $C_{S4} = x_{\text{Sueldo}} + (b_E/2) = 3 + (2.000/2) = 1.003$. Si tomamos en primer lugar la condición (Sexo = 'M') tendremos un coste estimado de $C_{S6a} = x_{\text{Sex}} + s_{\text{Sexo}} = 1 + 5.000 = 5.001$. El optimizador elegiría entonces el método S6*a* con el índice secundario en Dno, ya que es el que tiene la menor estimación de coste. La condición (Dno = 5) se utiliza para obtener los registros y la parte

restante de la condición conjuntiva (Sueldo > 30.000 AND Sexo = 'M') se comprueba para cada registro seleccionado una vez que es cargado en memoria.

### 15.8.4 Ejemplos de funciones de coste en JOIN

Para desarrollar funciones de coste razonablemente precisas en las operaciones JOIN necesitamos tener una estimación del tamaño (número de tuplas) del fichero resultante tras la operación JOIN. Normalmente, esto se calcula como la relación entre el tamaño (número de tuplas) del fichero resultante de la concatenación y el tamaño del fichero PRODUCTO CARTESIANO, si ambos se aplican a los mismos ficheros de entrada, y se denomina **selectividad de concatenación (*js*)**. Si llamamos $|R|$ al número de tuplas de una relación $R$, tendremos:

$$js = |(R \bowtie_c S)| \,/\, |(R \times S)| = |(R \bowtie_c S)| \,/\, (|R| * |S|)$$

Si no existe condición $c$ para la concatenación, entonces $js = 1$ y la concatenación es igual al PRODUCTO CARTESIANO. Si ninguna de las tuplas de las relaciones cumple la condición de concatenación, entonces $js = 0$. En general, $0 \leqslant js \leqslant 1$. En el caso de una concatenación donde la condición $c$ es una comparación de igualdad $R.A = S.B$, se presentan los dos casos especiales que se muestran a continuación:

1. Si $A$ es clave de $R$, entonces $|(R \bowtie_c S)| \leqslant |S|$; por tanto, $js \leqslant (1/|R|)$.
2. Si $B$ es clave de $S$, entonces $|(R \bowtie_c S)| \leqslant |R|$; por tanto, $js \leqslant (1/|S|)$.

Disponer de una selectividad de concatenación estimada para las condiciones de concatenación que se presentan con más asiduidad permite al optimizador de consultas calcular el tamaño del fichero resultante tras la operación de concatenación, dados los tamaños de los dos ficheros de entrada, utilizando la fórmula $|(R \bowtie_c S)| = js * |R| * |S|$. Ahora podemos mostrar algunos ejemplos de funciones de coste *aproximado* para el cálculo del coste de algunos de los algoritmos de concatenación que se muestran en la Sección 15.3.2. Las operaciones de concatenación tienen la forma:

$$R \bowtie_{A=B} S$$

donde $A$ y $B$ son atributos compatibles en dominio de $R$ y $S$, respectivamente. Supongamos que $R$ tiene $b_R$ bloques y que $S$ tiene $b_S$ bloques:

- **J1—Concatenación de bucle anidado.** Supongamos que utilizamos $R$ en el bucle externo; entonces tendríamos la siguiente función de coste para calcular el número de accesos a bloque para este método suponiendo que tenemos *tres búferes en memoria*. Supongamos también que el factor de bloqueo del fichero resultante es $bfr_{RS}$ y que la selectividad de la concatenación es conocida:

  $$C_{J1} = b_R + (b_R * b_S) + ((js * |R| * |S|)/bfr_{RS})$$

  La última parte de la fórmula es el coste de la escritura del fichero resultante en disco. Esta fórmula de coste puede ser modificada para tener en cuenta diferentes cantidades de búferes en memoria, según se muestra en la Sección 15.3.2.

- **J2—Concatenación de bucle simple (utilizando una estructura de acceso para obtener los registros que cumplen la condición).** Si existe un índice sobre el atributo $B$ de $S$ con niveles de índice $xB$, podemos obtener cada uno de los registros $s$ de $R$ y, posteriormente, utilizar el índice para obtener todos los registros emparejables $t$ de $S$ que satisfacen $t[B] = s[A]$. El coste depende del tipo de índice. Para un índice secundario donde $s_B$ es la cardinalidad de selección para el atributo de concatenación $B$ de $S$,[21] obtendremos:

[21] *La cardinalidad de selección* se definió como el número medio de registros que satisfacen una condición de igualdad sobre un atributo, que es el número medio de registros que tienen el mismo valor para ese atributo y que, por tanto, serán unidos en un único registro en el otro fichero.

$C_{J2a} = b_R + (|R| * (x_B + s_B)) + ((js * |R| * |S|)/bfr_{RS})$

Para un índice agrupado donde $s_B$ es la cardinalidad de selección de $B$, obtendremos:

$C_{J2b} = b_R + (|R| * (x_B + (sB/bfr_B))) + ((js * |R| * |S|)/bfr_{RS})$

Para un índice primario, obtendremos:

$C_{J2c} = b_R + (|R| * (x_B + 1)) + ((js * |R| * |S|)/bfr_{RS})$

Si existe una clave de dispersión en uno de los dos atributos de concatenación (por ejemplo, $B$ de $S$), obtendremos:

$C_{J2d} = b_R + (|R| * h) + ((js * |R| * |S|)/bfr_{RS})$

donde $h \geq 1$ es la cantidad media de accesos a bloque para obtener un registro, dado el valor de su clave de dispersión.

- **J3—Concatenación de ordenación-mezcla.** Si los ficheros ya se encuentran ordenados por los atributos de concatenación, la función de coste para este método es:

  $C_{J3a} = b_R + {}_{bS} + ((js * |R| * |S|)/bfr_{RS})$

  Si tuviéramos que ordenar los ficheros, sería necesario añadir el coste de la ordenación. Podríamos utilizar las fórmulas de la Sección 15.2 para calcular el coste de la ordenación.

**Ejemplo de utilización de las funciones de coste.** Supongamos que tenemos el fichero EMPLEADO que se describe en el ejemplo de la sección anterior y supongamos que el fichero DEPARTAMENTO de la Figura 5.5 está formado por $r_D = 125$ registros almacenados en $b_D = 13$ bloques de disco. Observe las operaciones de concatenación:

OP6: EMPLEADO $\bowtie_{\text{Dno=NúmeroDpto}}$ DEPARTAMENTO

OP7: DEPARTAMENTO $\bowtie_{\text{DniDirector=Dni}}$ EMPLEADO

Supongamos que tenemos un índice primario sobre NumeroDpto de DEPARTAMENTO de nivel $x_{\text{NúmeroDpto}} = 1$ y un índice secundario sobre DniDirector de DEPARTAMENTO con cardinalidad de selección $s_{\text{DniDirector}} = 1$ y $x_{\text{DniDirector}} = 2$ niveles. Si suponemos que la selectividad de concatenación para OP6 es $js_{\text{OP6}} = (1/|\text{DEPARTAMENTO}|) = 1/125$, ya que NúmeroDpto es clave de DEPARTAMENTO. Supongamos también que el factor de bloqueo para el fichero de concatenación resultante es $bfr_{\text{ED}} = 4$ registros por bloque. Podemos calcular el coste del peor caso para la operación JOIN OP6 utilizando los métodos aplicables J1 y J2 según se muestra a continuación:

1. Usando el método J1 con EMPLEADO como bucle externo:

$C_{J1} = b_E + (b_E * b_D) + ((js_{\text{OP6}} * r_E * r_D)/bfr_{ED})$
$= 2.000 + (2.000 * 13) + (((1/125) * 10.000 * 125)/4) = 30.500$

2. Usando el método J1 con DEPARTAMENTO como bucle externo:

$C_{J1} = b_D + (b_E * b_D) + ((js_{\text{OP6}} * r_E * r_D)/bfr_{ED})$
$= 13 + (13 * 2.000) + (((1/125) * 10.000 * 125/4) = 28.513$

3. Usando el método J2 con EMPLEADO como bucle externo:

$C_{J2c} = b_E + (r_E * (x_{\text{NumeroDpto}} + 1)) + ((js_{\text{OP6}} * r_E * r_D)/bfr_{ED}$
$= 2.000 + (10.000 * 2) + (((1/125) * 10.000 * 125/4) = 24.500$

4. Usando el método J2 con DEPARTAMENTO como bucle externo:

$$C_{J2a} = b_D + (r_D * (x_{Dno} + s_{Dno})) + ((js_{OP6} * r_E * r_D)/bfr_{ED})$$
$$= 13 + (125 * (2 + 80)) + (((1/125) * 10.000 * 125/4) = 12.763$$

El caso 4 tiene la menor estimación de coste y será el elegido. Observe que si estuviesen disponibles 15 búferes (o más) para la ejecución de la concatenación en lugar de estar disponibles sólo 3, 13 de ellos podrían ser usados para almacenar en memoria la relación DEPARTAMENTO en su totalidad, uno de ellos podría ser usado como búfer para el resultado, y el coste para el Caso 2 se podría reducir drásticamente a sólo $b_E + b_D + ((js_{OP6} * r_E * r_D)/bfr_{ED})$ ó 4.513, según vimos en la Sección 15.3.2. Como ejercicio, el lector debería realizar un análisis similar para OP7.

### 15.8.5 Consultas de relación múltiples y ordenación de JOIN

Las reglas algebraicas de transformación de la Sección 15.7.2 incluyen una regla conmutativa y una regla asociativa para la operación JOIN. Mediante estas reglas es posible generar muchas reglas de concatenación equivalentes. Como resultado de esto, el número de árboles de consulta alternativos crece muy rápidamente a medida que se incrementa el número de concatenaciones en una consulta. En términos generales, una consulta que una $n$ relaciones tendrá $n - 1$ operaciones de concatenación y, por tanto, puede tener un gran número de ordenaciones de concatenación. El cálculo del coste de todos los posibles árboles de concatenación de una consulta que tenga un gran número de concatenaciones requerirá una sustanciosa cantidad de tiempo por parte del optimizador de consultas. Por tanto, es necesario realizar una pequeña poda en los posibles árboles de consulta. Normalmente, los optimizadores de consultas limitan la estructura de un árbol de consultas de concatenación a la de la parte izquierda (o derecha) de los árboles. Un **árbol izquierdo** es un árbol binario donde el hijo de la derecha de cada nodo que no es hoja es siempre una relación base. El optimizador elegirá el árbol izquierdo en particular que tenga la menor estimación de coste. En la Figura 15.7 se muestran dos ejemplos de árboles izquierdos. (Observe que los árboles de la Figura 15.5 son también árboles izquierdos.)

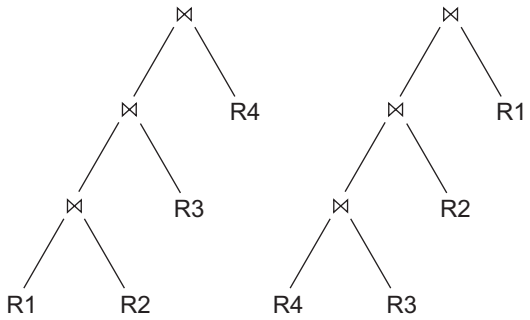
En el caso de los árboles izquierdos, se considera que el hijo de la derecha es la relación interna a la hora de ejecutar una concatenación de bucle anidado. Una ventaja de los árboles izquierdos (o de los árboles derechos) es que son susceptibles de ser secuenciados, según vimos en la Sección 15.6. Por ejemplo, observemos el primer árbol izquierdo de la Figura 15.7 y supongamos que el algoritmo de concatenación es el método de bucle simple; en este caso, se utiliza una página de disco con tuplas de la relación externa para probar la relación interna en búsqueda de tuplas emparejables. A medida que se genera un bloque de tuplas resultante de la concatenación de R1 y R2, podría ser utilizada para probar en R3. De igual modo, a medida que se genera una página de tuplas resultante de esta concatenación, podría ser utilizada para probar en R4. Otra ventaja de los árboles izquierdos (o derechos) es que usar una relación base como una de las entradas de cada unión permite al optimizador utilizar cualquier ruta de acceso a esa relación que resulte útil para la ejecución de la concatenación.

Si se utiliza la materialización en lugar de la secuenciación (véase la Sección 15.6), los resultados de la concatenación pueden ser materializados y almacenados como relaciones temporales. La idea clave desde el punto de vista del optimizador en relación con la ordenación de las concatenaciones es encontrar una ordenación que reduzca el tamaño de los resultados temporales, ya que éstos (secuenciados o materializados) son utilizados por los operadores siguientes y afectan, por tanto, al coste de ejecución de esos operadores.

### 15.8.6 Ejemplo para ilustrar la optimización de consultas basada en costes

Tomaremos la consulta C2 y su árbol de consultas que aparece en la Figura 15.4(a) para ilustrar la optimización de consultas basada en costes:

**Figura 15.7.** Dos árboles de consultas (JOIN) izquierdos.




```
C2:  SELECT   NumProyecto, NumDptoProyecto, Apellido1, Dirección, FechaNac
     FROM     PROYECTO, DEPARTAMENTO, EMPLEADO
     WHERE    NumDptoProyecto=NúmeroDpto AND DniDirector=Dni AND
              UbicaciónProyecto='Gijón';
```

Suponga que disponemos de la información estadística acerca de las relaciones que aparecen en la Figura 15.8. Los valores estadísticos VALOR_BAJO y VALOR_ALTO han sido normalizados por claridad. Se supone que el árbol de la Figura 15.4(a) representa el resultado del proceso de optimización algebraica heurística y el punto de comienzo para la optimización basada en costes (en este ejemplo, suponemos que el optimizador heurístico no desplaza las operaciones de proyección hacia abajo en el árbol).

La primera optimización basada en costes a tener en cuenta es la ordenación de concatenación. Según lo mencionado anteriormente, suponemos que el optimizador tiene en cuenta sólo los árboles izquierdos; por tanto, las posibles ordenaciones de concatenación (sin producto cartesiano) son:

1. PROYECTO ⋈ DEPARTAMENTO ⋈ EMPLEADO
2. DEPARTAMENTO ⋈ PROYECTO ⋈ EMPLEADO
3. DEPARTAMENTO ⋈ EMPLEADO ⋈ PROYECTO
4. EMPLEADO ⋈ DEPARTAMENTO ⋈ PROYECTO

Supongamos que la operación de selección ya ha sido aplicada a la relación PROYECTO. Si tenemos un modelo materializado, entonces se creará una nueva relación temporal tras cada operación de concatenación. Para examinar el coste de la ordenación de la concatenación (1), la primera concatenación se realiza entre PROYECTO y DEPARTAMENTO. Se debe determinar el método de concatenación y los métodos de acceso a las relaciones de entrada. Ya que DEPARTAMENTO no tiene ningún índice de acuerdo con la Figura 15.8, el único método de acceso disponible es la exploración de tabla (es decir, una búsqueda lineal). Se ejecutará la operación de selección sobre la relación PROYECTO antes que la concatenación, así que existen dos opciones: exploración de tabla (búsqueda lineal) o la utilización de su índice PROY_PUBICAC, así que el optimizador debe comparar sus costes estimados. La información estadística sobre el índice PROY_PUBICAC (véase la Figura 15.8) muestra el número de niveles de índice $x = 2$ (raíz más niveles hoja). El índice no es único (ya que UbicaciónProyecto no es clave de PROYECTO), así que el optimizador supone que la distribución de datos no es uniforme y calcula el número de punteros a registro para que cada valor de UbicaciónProyecto sea 10. Esto se calcula a partir de las tablas de la Figura 15.8 multiplicando Selectividad $*$ Num_filas, donde Selectividad se calcula como 1/Num_distintos. Por tanto, el coste de utilizar el índice y acceder a los registros se calcula en 12 accesos a bloque (2 para el índice y 10 para los bloques de datos). El coste de una exploración de tabla se calcula en 100 accesos a bloque, así que el acceso por índice es más eficiente según lo esperado.

**Figura 15.8.** Información estadística de ejemplo para las relaciones de C2. (a) Información de columna. (b) Información de tabla. (c) Información de índice.

**(a)**

| Nombre_tabla | Nombre_columna | Num_distintos | Valor_bajo | Valor_alto |
|---|---|---|---|---|
| PROYECTO | UbicaciónProyecto | 200 | 1 | 200 |
| PROYECTO | NumProyecto | 2.000 | 1 | 2.000 |
| PROYECTO | NumDptoProyecto | 50 | 1 | 50 |
| DEPARTAMENTO | NúmeroDpto | 50 | 1 | 50 |
| DEPARTAMENTO | DniDirector | 50 | 1 | 50 |
| EMPLEADO | Dni | 10.000 | 1 | 10.000 |
| EMPLEADO | Dno | 50 | 1 | 50 |
| EMPLEADO | Sueldo | 500 | 1 | 500 |

**(b)**

| Nombre_tabla | Num_filas | Bloques |
|---|---|---|
| PROYECTO | 200 | 100 |
| DEPARTAMENTO | 50 | 5 |
| EMPLEADO | 10.000 | 2.000 |

**(c)**

| Nombre_índice | Exclusividad | NivelB* | Bloques_hoja | Claves_distintas |
|---|---|---|---|---|
| PROY_PUBICAC | NONUNIQUE | 1 | 4 | 200 |
| EMP_DNI | UNIQUE | 1 | 50 | 10.000 |
| EMP_SUELDO | NONUNIQUE | 1 | 50 | 500 |

* NivelB es el número de niveles sin el nivel hoja.

En el modelo materializado, se crea un fichero temporal TEMP1 de tamaño 1 bloque para albergar el resultado de la operación de selección. El tamaño del fichero se calcula determinando el factor de bloqueo mediante la fórmula Num_filas/Bloques, lo cual da como resultado 2.000/100 o 20 filas por bloque. Por tanto, los 10 registros seleccionados en la relación PROYECTO cabrán en un único bloque. Ahora podemos calcular el coste estimado de la primera concatenación. Tendremos en cuenta sólo el método de concatenación de bucle anidado, donde la relación externa es el fichero temporal TEMP1 y la relación interna es DEPARTAMENTO. Ya que el fichero TEMP1 cabe en su totalidad en el espacio de búfer disponible, necesitamos leer cada uno de los cinco bloques de la tabla DEPARTAMENTO sólo una vez; por tanto, el coste de la concatenación es seis accesos a bloque más el coste de la escritura del fichero temporal resultante, TEMP2. El optimizador tendría que determinar el tamaño de TEMP2. Ya que el atributo de concatenación NúmeroDpto es la clave de DEPARTAMENTO, cualquier valor NumDptoProyecto de TEMP1 se unirá como máximo a un registro de DEPARTAMENTO, por lo que el número de filas de TEMP2 será igual al número de filas de TEMP1, que es 10. El optimizador calcularía el tamaño de registro de TEMP2 y el número de bloques necesarios para almacenar estas 10 filas. Para resumir, supongamos que el factor de bloqueo de TEMP2 es cinco filas por bloque, así que se necesitan dos bloques para almacenar TEMP2.

Por último, hay que calcular el coste de la última concatenación. Podemos utilizar una concatenación de bucle simple en TEMP2 ya que en este caso se puede utilizar el índice EMP_DNI (véase la Figura 15.8) para probar y localizar los registros emparejables de EMPLEADO. Por tanto, el método de concatenación implica la lectura de cada bloque de TEMP2 y la búsqueda de cada uno de los cinco valores DniDirector utilizando el índi-

ce EMP_DNI. Cada búsqueda de índice requeriría un acceso a la raíz, un acceso a las hojas y un acceso a bloques de datos ($x+1$, donde el número de niveles $x$ es 2). Por tanto, 10 búsquedas requieren 30 accesos a bloque. Si sumamos los dos accesos a bloque para TEMP2 nos da un total de 32 accesos a bloque para esta concatenación.

Para la proyección final, supongamos que se usa la secuenciación para generar el resultado final, lo que no requiere accesos a bloque adicionales; por tanto, el coste total para la ordenación de la concatenación (1) se calcula como la suma de los costes anteriores. El optimizador calcularía siguiendo el mismo método los costes para las otras tres ordenaciones de concatenación y elegiría la del menor coste estimado. Dejamos esto como ejercicio para el lector.

## 15.9 Revisión de la optimización de consultas en Oracle

El DBMS[22] de Oracle proporciona dos modelos distintos para la optimización de consultas: basado en reglas y basado en costes. Con el modelo basado en reglas, el optimizador selecciona los planes de ejecución basándose en operaciones clasificadas heurísticamente. Oracle mantiene una tabla de 15 rutas de acceso clasificadas, en las que la clasificación más baja implica un modelo más eficiente. Las rutas de acceso varían desde el acceso a la tabla por ID de fila (el más eficiente) (el ID de fila especifica la dirección física del registro que incluye el fichero de datos, el bloque de datos y el desplazamiento de la fila dentro del bloque) hasta la exploración completa en la tabla (la más ineficiente), en la que todas las filas de la tabla son exploradas haciendo lecturas de varios bloques. Sin embargo, el modelo basado en reglas está siendo desestimado en favor del modelo basado en costes, en el cual el optimizador examina rutas de acceso alternativas y algoritmos de operación y elige el plan de ejecución que tenga el menor coste estimado. El coste de consulta estimado es proporcional al tiempo transcurrido estimado que se necesita para ejecutar la consulta con el plan de ejecución dado. El optimizador de Oracle calcula este coste basándose en el uso estimado de los recursos, como la entrada/salida, el tiempo de CPU y la memoria necesaria. El objetivo de la estimación basada en costes de Oracle es minimizar el tiempo transcurrido para ejecutar la consulta completa.

Un añadido interesante al optimizador de consultas de Oracle es la capacidad que tiene un desarrollador de aplicaciones para especificar ***hints* (sugerencias)** al optimizador.[23] La idea es que es posible que el desarrollador de una aplicación tenga más información sobre los datos que el optimizador. Por ejemplo, observe la tabla EMPLEADO que aparece en la Figura 5.5. La columna Sexo de esa tabla sólo tiene dos valores distintos. Si existen 10.000 empleados, el optimizador supondría que la mitad son varones y la otra mitad son mujeres, suponiendo una distribución uniforme de los datos. Si existiese un índice secundario, lo más probable es que no se utilizase. Sin embargo, si el desarrollador de la aplicación sabe que sólo hay 100 empleados varones, se podría especificar una sugerencia en una consulta SQL cuya cláusula WHERE fuese Sexo = 'H', de modo que el índice asociado se utilizase en el procesamiento de la consulta. Se pueden especificar varias sugerencias, por ejemplo:

- El método de optimización para una sentencia SQL.
- El ruta de acceso a una tabla accedida por la sentencia.
- El orden de unión en una sentencia de concatenación.
- Una operación específica de unión en una sentencia de concatenación.

[22] Lo que aparece en esta sección se basa en la versión 7 de Oracle. Se han agregado más técnicas de optimización en versiones posteriores.

[23] A veces se llama *anotaciones* de consultas a las sugerencias de este tipo.

La optimización basada en costes de Oracle 8 es un buen ejemplo del sofisticado modelo utilizado para optimizar las consultas SQL en RDMS comerciales.

## 15.10 Optimización semántica de consultas

Existe una propuesta de un modelo diferente de optimización de consultas denominada **optimización semántica de consultas**. Esta técnica, que puede utilizarse en combinación con las técnicas descritas anteriormente, se basa en restricciones especificadas sobre el esquema de la base de datos (como atributos únicos y otro tipo de restricciones más complejas) para modificar una consulta dentro de otra consulta que sea más eficiente en su ejecución. No discutiremos este modelo en detalle, sino que lo explicaremos mediante un ejemplo sencillo. Tomemos esta consulta SQL:

```
SELECT   E.Apellido1, M.Apellido1
FROM     EMPLEADO AS E, EMPLEADO AS M
WHERE    E.SuperDni=M.Dni AND E.Sueldo > M.Sueldo
```

Esta consulta obtiene los nombres de los empleados que tienen un sueldo mayor que el de sus superiores. Supongamos que pusiésemos una restricción en el esquema de la base de datos que indicase que ningún empleado puede tener un salario mayor que el de su jefe directo. Si el optimizador semántico de consultas comprueba la existencia de esta restricción, no tendrá la necesidad de ejecutar la consulta, ya que sabrá que el resultado de la consulta estará vacío. Esto puede ahorrar mucho tiempo si la comprobación de restricciones se puede hacer de una manera eficiente. Sin embargo, la búsqueda en muchas restricciones para encontrar aquellas que son aplicables a una consulta dada y que pueden optimizarla semánticamente puede llegar a consumir mucho tiempo. Mediante la inclusión de reglas activas en los sistemas de bases de datos (consulte el Capítulo 24), la optimización semántica de consultas podría llegar a estar plenamente incorporada a los DBMSs en un futuro.

## 15.11 Resumen

En este capítulo hicimos un repaso de las técnicas utilizadas por los DBMS en el procesamiento y optimización de consultas de alto nivel. Comenzamos viendo cómo se traducen las consultas SQL a álgebra relacional y, posteriormente, vimos cómo los DBMSs ejecutan las distintas operaciones de álgebra relacional. Vimos que algunas de las operaciones, en particular SELECT y JOIN, se pueden ejecutar de diversas maneras. También vimos cómo se pueden combinar las operaciones durante el procesamiento de las consultas para crear un procesamiento en modo secuenciación o en modo flujo en lugar de ser procesadas en modo materializado.

Después de esto, describimos los modelos heurísticos utilizados en la optimización de consultas, que usan reglas heurísticas y técnicas algebraicas para mejorar la eficiencia en la ejecución de las consultas. Mostramos cómo se puede optimizar de forma heurística un árbol de consultas que representa una expresión de álgebra relacional reorganizando los nodos del árbol y transformándolo en otro árbol de consultas equivalente de ejecución más eficiente. También ofrecimos unas reglas de transformación para el mantenimiento de las equivalencias que pueden ser utilizadas en un árbol de consultas. Posteriormente, presentamos planes de ejecución de consultas para las consultas SQL que añaden planes de ejecución de métodos a las operaciones del árbol de consultas.

Vimos el modelo de optimización de consultas basado en costes. Mostramos cómo se desarrollan las funciones de coste en algunos algoritmos de acceso a bases de datos y cómo se utilizan estas funciones de coste en la estimación de los costes de las diferentes estrategias de ejecución. Hicimos un repaso del optimizador de consultas de Oracle y mencionamos la técnica de la optimización semántica de consultas.

## Preguntas de repaso

**15.1.** Comente las razones para convertir las consultas SQL en consultas de álgebra relacional antes de realizar la optimización.

**15.2.** Explique los diferentes algoritmos para la implementación de cada uno de los siguientes operadores relacionales y las circunstancias bajo las cuales se puede utilizar cada algoritmo: SELECT, JOIN, PROJECT, UNION, INTERSECT, SET DIFFERENCE, CARTESIAN PRODUCT.

**15.3.** ¿Qué es un plan de ejecución de consultas?

**15.4.** ¿Qué significa el término *optimización heurística*? Comente las principales heurísticas que se aplican durante la optimización de consultas.

**15.5.** ¿Cómo representa un árbol de consultas una expresión de álgebra relacional? ¿Qué implica la ejecución de un árbol de consultas? Comente las reglas de transformación de árboles de consulta e identifique cuándo se debería aplicar cada regla durante la optimización.

**15.6.** ¿Cuántas ordenaciones de concatenación existen en una consulta que concatene 10 relaciones?

**15.7.** ¿Qué significa *optimización de consultas basada en costes*?

**15.8.** ¿Cuál es la diferencia entre *secuenciación* y *materialización*?

**15.9.** Comente los componentes de coste para una función de coste que se use para el coste de ejecución de una consulta. ¿Qué componentes del coste se usan con más frecuencia como base para las funciones de coste?

**15.10.** Comente los diferentes tipos de parámetros que se usan en las funciones de coste. ¿Dónde se guarda la información?

**15.11.** Enumere las funciones de coste para los métodos SELECT y JOIN vistos en la Sección 15.8.

**15.12.** ¿Qué significa la optimización semántica de consultas? ¿En qué se diferencia de otras técnicas de optimización de consultas?

## Ejercicios

**15.13.** Tome las consultas SQL C1, C8, C1B, C4 y C27 del Capítulo 8.

- **a.** Dibuje al menos dos árboles de consultas que puedan representar *cada una* de estas consultas. ¿Bajo qué circunstancias utilizaría cada uno de sus árboles de consultas?
- **b.** Dibuje el árbol de consultas inicial para cada una de estas consultas y diga, a continuación, cómo se optimiza el árbol de consultas mediante el algoritmo descrito en la Sección 15.7.
- **c.** Para cada consulta, compare sus propios árboles de consulta del apartado (a) y los árboles de consulta inicial y final del apartado (b).

**15.14.** Un fichero de 4.096 bloques tiene que ser ordenado con un espacio de búferes disponible de 64 bloques. ¿Cuántas pasadas se necesitarán en la fase de mezcla del algoritmo de ordenación-mezcla externa?

**15.15.** Desarrolle funciones de coste para los algoritmos PROJECT, UNION, INTERSECTION, SET DIFFERENCE y CARTESIAN PRODUCT vistos en la Sección 15.4.

**15.16.** Desarrolle funciones de coste para un algoritmo formado por dos SELECT, una JOIN y una PROJECT final, en términos de las funciones de coste para las operaciones individuales.

**15.17.** ¿Puede un índice no denso ser usado en la implementación de un operador de agregación? ¿Por qué o por qué no?

**15.18.** Calcule las funciones de coste para diferentes opciones de ejecución de la operación JOIN OP7 vista en la Sección 15.3.2.

**15.19.** Desarrolle fórmulas para el algoritmo de concatenación de dispersión híbrida para calcular el tamaño del búfer del primer bloque de memoria. Desarrolle fórmulas de estimación de coste más precisas para el algoritmo.

**15.20.** Calcule el coste de las operaciones OP6 y OP7, utilizando las fórmulas desarrolladas en el Ejercicio 5.9.

**15.21.** Extienda el algoritmo de concatenación de ordenación-mezcla para implementar la operación LEFT OUTER JOIN.

**15.22.** Compare el coste de dos planes diferentes de ejecución de consultas para la siguiente consulta:

$$\sigma_{\text{Sueldo} > 40000}(\text{EMPLEADO} \bowtie_{\text{Dno}=\text{NúmeroDpto}} \text{DEPARTAMENTO})$$

Utilice las estadísticas de base de datos de la Figura 15.8.

## Bibliografía seleccionada

El estudio de Graefe (1993) estudia la ejecución de consultas en sistemas de bases de datos e incluye una extensa bibliografía. El estudio de Jarke y Koch (1984) proporciona una clasificación de la optimización de consultas e incluye una bibliografía del trabajo en este área. Smith and Chang (1975) proporciona un algoritmo detallado para la optimización del álgebra relacional. La tesis doctoral de Kooi (1980) proporciona una base para las técnicas de optimización de consultas.

Whang (1985) revisa la optimización de consultas en OBE (Office-By-Example), que es un sistema basado en QBE. La optimización basada en costes se presentó en el DBMS experimental SYSTEM R y se revisa en Astrahan y otros (1976). Selinger y otros (1979) revisa la optimización de concatenaciones multivía en SYSTEM R. Los algoritmos de concatenación se revisan en Gotlieb (1975), Blasgen y Eswaran (1976), y Whang y otros (1982). Los algoritmos de dispersión para la implementación de concatenaciones se describen y analizan en Blakeley y Martin (1990), entre otros. Los modelos para encontrar un buen orden de concatenación se muestran en Ioannidis and Kang (1990) y en Swami and Gupta (1989). Una revisión de las implicaciones de los árboles izquierdos de concatenación se presenta en Ioannidis y Kang (1991). Kim (1982) repasa las transformaciones de consultas SQL anidadas en representaciones canónicas. La optimización de funciones agregadas aparece en Klug (1982) y Muralikrishna (1992). Salzberg y otros (1990) describe un algoritmo rápido de ordenación externa. El cálculo del tamaño de las relaciones temporales es crucial para la optimización de las consultas. En Haas y otros (1995) y en Haas y Swami (1995) se presentan esquemas de estimación basados en muestreos. Lipton y otros (1990) también repasa la estimación de selectividades. El almacenamiento en sistemas de bases de datos y el uso de estadísticas más detalladas en forma de histogramas se trata en Muralikrishna y DeWitt (1988) y Poosala y otros (1996).

Kim y otros (1985) hacen un repaso a temas avanzados en la optimización de consultas. La optimización semántica de consultas se revisa en King (1981) y en Malley y Zdonick (1986). Un informe sobre trabajos más recientes sobre optimización semántica de consultas aparece en Chakravarthy y otros (1990), Shenoy y Ozsoyoglu (1989), y Siegel y otros (1992).

CAPÍTULO 16

# Diseño físico y refinación de la base de datos

En el último capítulo explicamos diversas técnicas para poder procesar eficazmente las consultas. En éste explicamos algunos problemas que afectan al rendimiento de una aplicación que se ejecuta en un DBMS. En particular, explicamos las opciones disponibles que los administradores de bases de datos tienen para almacenar bases de datos y algunas de las soluciones y técnicas que utilizar para refinar la base de datos a fin de mejorar su rendimiento. En primer lugar, la Sección 16.1 explica los problemas que surgen en el diseño físico de la base de datos que tienen que ver con el almacenamiento y el acceso a los datos. Después, en la Sección 16.2 explicamos cómo mejorar el rendimiento de la base de datos a través de la refinación, la indexación de datos, el diseño y las propias consultas.

## 16.1 Diseño físico de las bases de datos relacionales

En esta sección explicamos los factores de diseño físico que afectan al rendimiento de las aplicaciones y las transacciones; y después comentamos las directrices específicas para los RDBMSs.

### 16.1.1 Factores que influyen en el diseño físico de una base de datos

El diseño físico es una actividad cuyo objetivo no sólo es crear la estructuración adecuada de los datos en su almacenamiento, sino también hacer algo que garantice un buen rendimiento. Para un esquema conceptual dado, hay muchas alternativas de diseño físico en un DBMS dado. No es posible tomar decisiones significativas sobre el diseño físico y analizar el rendimiento mientras no conozcamos las consultas, las transacciones y las aplicaciones que se esperan ejecutar sobre la base de datos. Debemos analizar estas aplicaciones, su frecuencia de ejecución esperada, cualquier restricción de tiempo en su ejecución y la frecuencia estimada de las operaciones de actualización. A continuación explicamos estos factores.

**A. Análisis de las consultas y las transacciones de la base de datos.** Antes de emprender el diseño físico de la base de datos, debemos tener una buena idea del uso que se espera hacer de la base de datos, que lo definiremos en un formulario de alto nivel, así como de las consultas y las transacciones que es previsible ejecutar en la base de datos. Por cada consulta, debemos especificar lo siguiente:

1. Los ficheros a los que la consulta accederá.[1]
2. Los atributos con los que se especificará cualquier condición de selección para la consulta.
3. Si la condición de selección es una condición de igualdad, desigualdad o de rango.
4. Los atributos con los que se especificará cualquier condición de concatenación para enlazar varias tablas u objetos para la consulta.
5. Los atributos cuyos valores la consulta recuperará.

Los atributos mencionados en los puntos 2 y 4 son candidatos para la definición de estructuras de acceso. Para cada transacción u operación de actualización, debemos especificar lo siguiente:

1. Los ficheros que se actualizarán.
2. El tipo de operación en cada fichero (inserción, actualización o eliminación).
3. Los atributos con los que se especifican las condiciones de selección para una eliminación o una actualización.
4. Los atributos cuyos valores cambiarán a causa de una operación de actualización.

Una vez más, los atributos mencionados en el punto 3 son candidatos para las estructuras de acceso. Por el contrario, los atributos del punto 4 son candidatos a evitarse en una estructura de acceso, puesto que la modificación de esos atributos requeriría la actualización de las estructuras de acceso.

**B. Análisis de la frecuencia de ejecución esperada de consultas y transacciones.** Además de identificar las características de las consultas y transacciones esperadas, debemos considerar la frecuencia de invocación que esperamos. La información sobre esta frecuencia, junto con la información de los atributos recopilada en cada consulta y transacción, se utiliza para recopilar una lista acumulativa de frecuencias de uso esperadas para todas las consultas y transacciones. Esto se expresa como la frecuencia esperada de uso de cada atributo de cada fichero como un atributo de selección o un atributo de concatenación, sobre todas las consultas y transacciones. Por regla general, para grandes volúmenes de procesamiento, se aplica la *regla informal del 80-20*, que dice que aproximadamente el 80% del procesamiento se debe al 20% de las consultas y las transacciones. Por consiguiente, en situaciones prácticas, rara vez es necesario recopilar unas estadísticas y tasas de invocación exhaustivas para todas las consultas y transacciones; es suficiente determinar el 20% de las más importantes.

**C. Análisis de las restricciones de tiempo para consultas y transacciones.** Algunas consultas y transacciones pueden tener restricciones de rendimiento severas. Por ejemplo, una transacción puede tener la restricción de que debe terminar en 5 segundos el 95% de las veces que se la invoca y que nunca debe tardar más de 20 segundos. Dichas restricciones de rendimiento fijan prioridades más extensas en los atributos que son candidatos a las rutas de acceso. Los atributos de selección que las consultas y las transacciones utilizan con restricciones de tiempo se convierten en candidatos de alta prioridad a las estructuras de acceso principales.

**D. Análisis de las frecuencias esperadas de las operaciones de actualización.** Debemos especificar una cantidad mínima de rutas de acceso para un fichero que se actualiza con frecuencia, porque la actualización de las propias rutas de acceso ralentiza las operaciones de actualización.

**E. Análisis de las restricciones de unicidad en los atributos.** Las rutas de acceso deben especificarse en todos los atributos (o conjuntos de atributos) clave que son candidatos a ser la clave principal o que están restringidos para ser únicos. La existencia de un índice (o de otra ruta de acceso) hace suficiente explorar el índice sólo cuando se verifica esta restricción, puesto que todos los valores del atributo existirán en los nodos hoja del índice.

[1] Por simplicidad utilizamos el término *ficheros*. Lo podemos sustituir por tablas o relaciones.

Una vez recopilada la información precedente, podemos afrontar las decisiones sobre el diseño físico de la base de datos, que consisten principalmente en decidir las estructuras de almacenamiento y las rutas de acceso para los ficheros de base de datos.

## 16.1.2 Decisiones sobre el diseño físico de la base de datos

La mayoría de los sistemas relacionales representan cada relación base como un fichero de base de datos físico. Las opciones de ruta de acceso incluyen especificar el tipo de fichero para cada relación y los atributos sobre los que deben especificarse los índices. Como máximo, uno de los índices de cada fichero puede ser un índice principal o agrupado. Es posible crear cualquier cantidad de índices secundarios.[2]

**Decisiones de diseño sobre la indexación.** Los atributos cuyos valores son requeridos por las condiciones de igualdad o de rango (operación de selección) y los que son claves o participan en condiciones de concatenación (operación de concatenación) requieren rutas de acceso.

El rendimiento de las consultas depende ampliamente de los índices o de los esquemas de dispersión existentes para acelerar el procesamiento de las selecciones y las concatenaciones. Por el contrario, durante las operaciones de inserción, eliminación o actualización, la existencia de índices supone un coste, que se puede justificar por una mayor eficacia a la hora de tramitar las consultas y las transacciones.

Las decisiones sobre el diseño físico de la indexación se encuadran en las siguientes categorías:

1. **Cuándo indexar un atributo.** Si un atributo es clave o si se utiliza para alguna consulta, para una condición de selección (igualdad o rango de valores) o para una concatenación, hay una justificación inicial para crear un índice por ese atributo. Un factor a favor de crear muchos índices es que algunas consultas puedan procesarse explorando simplemente los índices, sin necesidad de recuperar datos.
2. **Qué atributo o atributos indexar.** Un índice se puede crear con uno o varios atributos. Si varios atributos de una relación están implicados todos ellos en varias consultas, [por ejemplo, (NumEstiloPrenda, Color) en una base de datos de inventario de prendas de vestir], está justificado un índice multiatributo. La ordenación de los atributos dentro de un índice multiatributo debe corresponder a las consultas. Por ejemplo, el índice anterior asume que las consultas estarían basadas en una ordenación de los colores por cada NumEstiloPrenda, y no a la inversa.
3. **Cuándo configurar un índice agrupado.** Como máximo, en una tabla sólo puede haber un índice principal o agrupado porque esto implica que el fichero está ordenado físicamente por ese atributo. En la mayoría de los RDBMSs, esto se especifica con la palabra clave CLUSTER. (Si el atributo es una clave, se crea un índice principal, mientras que un índice agrupado se crea si el atributo no es una clave). Si una tabla requiere varios índices, la decisión sobre cuál debe ser un índice agrupado depende de si es necesario mantener la tabla ordenada por ese atributo. Los índices agrupados favorecen las consultas por rango. Si varios atributos requieren consultas por rango, deben evaluarse los beneficios relativos antes de decidir el atributo por el que agrupar. Si la consulta se puede responder con una simple búsqueda en el índice (sin recuperar los registros de datos), el índice correspondiente *no* debe agruparse, puesto que el beneficio principal del agrupamiento se consigue si hay que recuperar los registros. Se puede configurar un índice agrupado como un índice multiatributo si la recuperación de rango por esa clave compuesta resulta de utilidad en la creación de un informe (por ejemplo, un índice sobre CodPostal, IdAlmacén e IdProducto pueden ser un índice agrupado para los datos de ventas).
4. **Cuándo utilizar un índice de dispersión sobre un índice de árbol.** En general, los RDBMSs utilizan árboles $B^+$ para la indexación. No obstante, en algunos sistemas también se proporcionan índices ISAM y de dispersión (consulte el Capítulo 14). Los árboles $B^+$ soportan consultas de igualdad y de

[2] El lector debe repasar los distintos tipos de índices descritos que describimos en la Sección 13.1. Para entender mejor esta explicación, también resulta útil familiarizarse con los algoritmos de procesamiento de consultas descritos en el Capítulo 15.

rango sobre el atributo utilizado como clave de búsqueda. Los índices de dispersión funcionan bien con las condiciones de igualdad, en particular durante las concatenaciones para encontrar el(los) registro(s) coincidente(s).

5. **Cuándo utilizar la dispersión dinámica para el fichero.** En los ficheros que son muy volátiles (es decir, aquellos cuyo tamaño aumenta y se reduce continuamente) sería conveniente uno de los esquemas de dispersión dinámica que explicamos en la Sección 13.9. Actualmente, la mayoría de los RDBMSs comerciales no los ofrecen.

**Desnormalización como decisión de diseño para acelerar las consultas.** El objetivo último durante la normalización (consulte los Capítulos 10 y 11) es separar los atributos relacionados lógicamente en tablas a fin de minimizar la redundancia, y así evitar las anomalías que pueden llevar a un procesamiento extra a la hora de mantener la coherencia de la base de datos. Los ideales que suelen seguirse son las formas normales Boyce-Codd (consulte el Capítulo 10).

Los ideales mencionados se sacrifican a veces en favor de una ejecución más rápida de las consultas y las transacciones más frecuentes. Este proceso de almacenamiento del diseño lógico de la base de datos (que puede estar en BCNF o 4NF) en una forma normal más débil, por ejemplo 2NF o 1NF, se denomina desnormalización. Normalmente, el diseñador añade a una tabla los atributos que son necesarios para responder consultas o producir informes, de modo que se evita una concatenación con otra tabla, que contiene el atributo recién añadido. Esto reintroduce una dependencia funcional parcial o una dependencia transitiva en la tabla, surgiendo por tanto problemas de redundancia asociados (consulte el Capítulo 10). Existe una contrapartida entre la actualización adicional necesaria para mantener la coherencia frente al esfuerzo necesario para efectuar una concatenación a fin de incorporar los atributos adicionales necesarios en el resultado. Por ejemplo, considere la siguiente relación:

ASSIGN (IdEmp, IdProy, NombreEmp, TítuloTrabjEmp, PorcentajeAsignado, NombreProy, IdDirectorProy, NomDirectorProy),

que corresponde exactamente a las cabeceras de un informe denominado *Lista de asignación de empleados*.

Esta relación sólo está en 1NF debido a las siguientes dependencias funcionales:

IdProy → NombreProy, IdDirectorProy
IdDirectorProy → NomDirectorProy
e IdEmp → NombreEmp, TítuloTrabjEmp

Esta relación puede ser preferible al diseño en 2FN (y 3FN) y consiste en estas tres relaciones:

EMP (IdEmp, NombreEmp, TítuloTrabjEmp)
PROY (IdProy, NombreProy, IdDirectorProy)
EMP_PROY (IdEmp, IdProy, PorcentajeAsignado),

porque para producir el informe *Lista de asignación de empleados* (con todos los campos mostrados anteriormente en ASSIGN), el diseño de la última multi-relación requiere las siguientes concatenaciones:

$$\text{EMP_PROY} * \text{EMP} * \text{PROY} \bowtie_{\text{IdDirectorProy = IdEmp}} \text{EMP}$$

En la expresión anterior, se necesita la concatenación final para obtener NomDirectorProy a partir de IdDirectorProy.

Otras formas de desnormalización consisten en almacenar tablas adicionales para mantener las dependencias funcionales originales que se pierden durante una descomposición BCNF. Por ejemplo, la Figura 10.13 mostraba la relación ENSEÑAR(Estudiante, Curso, Profesor) con las dependencias funcionales {{Estudiante, Curso} → Profesor, Profesor → Curso}. Una descomposición sin pérdida de ENSEÑAR en T1(Estudiante, Profesor) y T2(Profesor, Curso) *no* permite responder consultas del tipo "*¿a qué curso asistió el estudiante*

*Pérez y qué impartió la profesora María?*" sin tener que concatenar T1 y T2. Por consiguiente, una posible solución sería almacenar T1, T2 y ENSEÑAR, lo que reduce el diseño de BCNF a 3NF. Aquí, ENSEÑAR es una concatenación consistente en las otras dos tablas, lo que representa una redundancia extrema. Cualquier actualización de T1 y T2 debería aplicarse a ENSEÑAR. Una estrategia alternativa es considerar que T1 y T2 son tablas actualizables, mientras que ENSEÑAR se puede crear como una vista.

## 16.2 Visión general de la refinación de una base de datos en los sistemas relacionales

Una vez implantada una base de datos y ya en funcionamiento, el uso real de aplicaciones, transacciones, consultas y vistas revela factores y áreas problemáticas que durante el diseño físico inicial pudieron no tenerse en cuenta. Los factores de diseño físico mencionados en la Sección 16.1.1 pueden revisarse mediante la obtención de estadísticas de patrones de uso reales. La utilización de recursos, así como el procesamiento interno de DBMS (por ejemplo, la optimización de consultas), pueden monitorizarse para revelar cuellos de botella, como la confrontación de los mismos datos o dispositivos. Es posible hacer una mejor estimación de los volúmenes de actividad y de los tamaños de los datos. Por consiguiente, es necesario monitorizar y revisar constantemente el diseño físico de la base de datos. Los objetivos de la refinación son los siguientes:

- Conseguir que las aplicaciones se ejecuten más rápidamente.
- Reducir el tiempo de respuesta de las consultas y las transacciones.
- Mejorar el rendimiento global de las transacciones.

La línea divisoria entre el diseño físico y la refinación es muy fina. Las mismas decisiones de diseño que vimos en la Sección 16.1.2 se vuelven a visitar durante la fase de refinación, que es un ajuste continuado del diseño. A continuación ofrecemos una breve panorámica del proceso de refinación.[3] Como entrada de este proceso tenemos las estadísticas relacionadas con los factores mencionados en la Sección 16.1.1. En particular, los DBMSs pueden recopilar internamente las siguientes estadísticas:

- Tamaños de las tablas individuales.
- Número de valores distintos en una columna.
- Número de veces que una consulta o transacción en particular se emite y ejecuta en un intervalo de tiempo.
- Las veces que las diferentes fases requieren el procesamiento de consultas y transacciones (para un conjunto dado de consultas o transacciones).

Éstas y otras estadísticas crean un perfil del contenido y uso de la base de datos. De la monitorización de las actividades del sistema de bases de datos y de los procesos se puede obtener esta otra información:

- **Estadísticas de almacenamiento**. Datos sobre la asignación de almacenamiento en los espacios de tablas, espacios de índices y almacenes de búfer.
- **Estadísticas de rendimiento de E/S y de los dispositivos**. Actividad de lectura/escritura total (paginación) en el disco.
- **Estadísticas de procesamiento de consultas/transacciones**. Tiempos de ejecución de las consultas y las transacciones, optimización de los tiempos durante la optimización de las consultas.

[3] Los lectores interesados deben consultar Shasha (1992) y Shasha y Bonnett (2002) si desean una explicación detallada de la refinación.

- **Estadísticas relacionadas con el bloqueo/inicio de sesión**. Tasas de emisión de los diferentes tipos de bloqueos, tasas de rendimiento de transacciones, y actividad de registro.[4]
- **Estadísticas sobre los índices**. Número de niveles de un índice, número de páginas hoja no contiguas, etcétera.

Muchas de las estadísticas anteriores están relacionadas con las transacciones, el control de la concurrencia y la recuperación, que se explican en los Capítulos 17 a 19. La refinación de una base de datos puede enfrentarnos a estos problemas:

- Cómo evitar una contención de bloqueo excesiva, incrementándose en consecuencia la concurrencia entre las transacciones.
- Cómo minimizar el coste añadido del *logging* y de una descarga excesiva de datos.
- Cómo optimizar el tamaño del búfer y planificar los procesos.
- Cómo asignar recursos como los discos, la memoria RAM y los procesos a fin de conseguir el uso más eficaz.

La mayoría de estos problemas pueden resolverse configurando los parámetros físicos apropiados del DBMS, cambiando las configuraciones de dispositivos, modificando los parámetros del sistema operativo, y otras actividades parecidas. Las soluciones tienden a estar estrechamente relacionadas con sistemas específicos. Los DBAs normalmente están preparados para hacer frente a estos problemas de refinación en DBMSs específicos. Vamos a explicar la refinación de diversas decisiones de diseño de bases de datos.

### 16.2.1 Refinación de los índices

Es posible tener que revisar la decisión tomada inicialmente sobre los índices por las siguientes razones:

- Ciertas consultas pueden tardar demasiado en ejecutarse por carecer de un índice.
- No es posible utilizar ciertos índices.
- Algunos índices pueden sufrir demasiadas actualizaciones porque el índice se ha creado sobre un atributo que experimenta cambios frecuentes.

La mayoría de los DBMSs tienen un comando o utilidad de rastreo (*trace*) que el DBA puede utilizar para saber cómo se ha ejecutado una consulta (qué operaciones se han llevado a cabo y en qué orden, y qué estructuras de acceso secundario se han utilizado). Al analizar esta planificación de ejecución, es posible diagnosticar las causas de los problemas anteriores. Basándonos en el análisis pueden descartarse algunos índices y crearse otros.

El objetivo de la refinación es evaluar dinámicamente los requisitos, que a veces fluctúan estacionalmente o durante periodos de tiempo de un mes o una semana, así como reorganizar los índices a fin de conseguir el mejor rendimiento global. La eliminación y la creación de índices supone un coste añadido que puede justificarse si se busca la mejora del rendimiento. La actualización de una tabla queda generalmente suspendida mientras se elimina o crea un índice; hay que considerar esta pérdida de servicio. Además de eliminar o crear índices, y de cambiar de un índice no agrupado a otro agrupado y viceversa, la **reconstrucción del índice** puede mejorar el rendimiento. La mayoría de los RDBMSs utilizan árboles $B^+$ para el índice. Si hay muchas eliminaciones en la clave del índice, las páginas del índice pueden contener espacio desaprovechado, que puede reclamarse durante una operación de reconstrucción. De forma parecida, demasiadas inserciones pueden provocar desbordamientos en un índice agrupado que afecten al rendimiento. La reconstrucción de un índice agrupado equivale a reorganizar la tabla entera ordenada por esa clave.

[4] El lector puede ojear los Capítulos 17–19 si desea una explicación de estos términos.

Las opciones disponibles para la indexación y la forma en que se define, crea y reorganiza, varía de un sistema a otro. A modo de ilustración, considere los índices escaso y denso del Capítulo 14. Los índices escasos tienen un puntero de índice por cada página (bloque de disco) en el fichero de datos; los índices densos tienen un puntero de índice por cada registro. Sybase proporciona índices agrupados como índices escasos en forma de árboles $B^+$, mientras que INGRES ofrece índices agrupados escasos como ficheros ISAM, e índices agrupados densos como árboles $B^+$. En algunas versiones de Oracle y DB2, la opción de configurar un índice agrupado está limitada a un índice denso (con muchas más entradas de índice), y el DBA tiene que trabajar con esta limitación.

### 16.2.2 Refinación del diseño de la base de datos

En la Sección 16.1.2 vimos la necesidad de una posible desnormalización, que es una opción de mantener todas las tablas como relaciones BCNF. Si el diseño físico de una base de datos no satisface los objetivos esperados, podemos volver al diseño lógico de la base de datos, realizar ajustes en dicho diseño y volver a mapearlo a un nuevo conjunto de tablas e índices físicos.

Como explicamos, el diseño entero de la base de datos tiene que estar guiado por los requisitos de procesamiento y los requisitos de datos. Si los requisitos de procesamiento cambian dinámicamente, el diseño tiene que responder introduciendo cambios en el esquema conceptual si es necesario y reflejando esos cambios en el esquema lógico y en el diseño físico. Estos cambios pueden ser de la siguiente naturaleza:

- Las tablas existentes pueden concatenarse (desnormalizarse) porque con frecuencia se necesitan juntos ciertos atributos de dos o más tablas: esto reduce el nivel de normalización de BCNF a 3NF, 2NF o 1NF.[5]
- Para el conjunto de tablas dado, puede haber opciones de diseño alternativas; todas ellas logran 3NF o BCNF. En el Capítulo 11 ilustramos los diseños equivalentes alternativos. Uno puede reemplazarse por otro.
- Una relación de la forma R(K,A, B, C, D, ...), con K como un conjunto de atributos clave, que está en BCNF puede almacenarse en varias tablas que también están en BCNF [por ejemplo, R1(K, A, B), R2(K, C, D, ), R3(K, ...)] replicando la clave K en cada tabla. Cada tabla agrupa conjuntos de atributos a los que se accede conjuntamente. Por ejemplo, la tabla EMPLEADO(Dni, Nombre, Tlf, Categ, Sueldo) puede dividirse en dos tablas: EMP1(Dni, Nombre, Tlf) y EMP2(Dni, Categ, Sueldo). Si la tabla original tiene muchas filas (por ejemplo, 100.000) y las consultas sobre los números de teléfono y los sueldos son completamente distintas y se producen con frecuencias muy diferentes, entonces esta separación de tablas puede funcionar mejor. Es lo que también se conoce como **división vertical**.
- El (los) atributo(s) de una tabla pueden repetirse en otra aunque esto introduzca redundancia y una anomalía potencial. Por ejemplo, NombrePieza puede duplicarse en las tablas dondequiera que aparezca el NumPieza (como *foreign key*), pero puede haber una tabla maestra denominada PIEZAS_MAESTRA(NumPieza, NombrePieza, ...) donde se garantice la actualización del nombre de la pieza.
- Así como la división vertical divide una tabla verticalmente en varias tablas, la **división horizontal** toma sectores horizontales de una tabla y los almacena como tablas distintas. Por ejemplo, los datos de ventas por producto se pueden separar en diez tablas basadas en las diez líneas de producto. Todas las tablas tienen el mismo conjunto de columnas (atributos), pero contienen un conjunto distinto de productos (tuplas). Si una consulta o una transacción se aplica a todos los datos de producto, puede que tenga que ejecutarse contra todas las tablas y que haya que combinar los resultados.

[5] 3NF y 2NF se dirigen a diferentes tipos de problemas de dependencia que son independientes entre sí; por tanto, el orden de normalización (o desnormalización) entre ellas es arbitrario.

En la práctica son comunes estos tipos de ajustes que se diseñan para satisfacer la gran cantidad de consultas y transacciones, sacrificando o no las formas normales.

### 16.2.3 Refinación de consultas

Ya hemos explicado cómo el rendimiento de una consulta depende de la selección adecuada de los índices y cómo éstos deben refinarse después de analizar las consultas que arrojan un rendimiento pobre, tarea para la que se utilizan los comandos del RDBMS que muestran la planificación de la ejecución de la consulta. Hay principalmente dos indicaciones que sugieren la necesidad de refinar una consulta:

1. Una consulta efectúa demasiados accesos al disco (por ejemplo, una consulta por coincidencia exacta explora una tabla entera).
2. La planificación de la consulta muestra que no se están utilizando los índices pertinentes.

Algunos casos típicos de situaciones que incitan a refinar una consulta son los siguientes:

1. Muchos optimizadores de consultas no utilizan los índices en presencia de expresiones aritméticas (por ejemplo, Sueldo/365 > 10.50), comparaciones numéricas de atributos de tamaños y precisiones diferentes (por ejemplo, Aqty = Bqty donde Aqty es del tipo INTEGER y Bqty es del tipo SMALLINTEGER), comparaciones con NULL (por ejemplo, FechaNac IS NULL), y comparaciones de subcadenas (por ejemplo, Apellido LIKE '%ez').
2. No se utilizan con frecuencia los índices en las consultas anidadas con IN; por ejemplo, la siguiente consulta:

```
SELECT   Dni   FROM EMPLEADO
WHERE    Dno  IN (SELECT NúmeroDpto FROM DEPARTAMENTO
                  WHERE DniDirector = '333445555' );
```

   no puede utilizar el índice por Dno en EMPLEADO, mientras que el uso de Dno = NúmeroDpto en la cláusula WHERE con la consulta de un solo bloque puede provocar que se utilice el índice.
3. Algunos DISTINCTs pueden ser redundantes y pueden evitarse sin tener que cambiar el resultado. DISTINCT a menudo provoca una operación de ordenación y debe evitarse siempre que sea posible.
4. El uso innecesario de tablas de resultado temporales pueden evitarse colapsando varias consultas en una sola *a menos que* se necesite la relación temporal para algún procesamiento intermedio.
5. En algunas situaciones que implican el uso de consultas correlativas, los temporales son útiles. Considere la siguiente consulta:

```
SELECT   Dni
FROM     EMPLEADO E
WHERE    Sueldo = SELECT MAX (Sueldo)
                  FROM EMPLEADO AS M
                  WHERE M.Dno = E.Dno;
```

   Esto tiene el peligro potencial de buscar en toda la tabla EMPLEADO M interior por *cada* tupla de la tabla EMPLEADO E exterior. Para que esto sea más eficaz, se puede dividir en dos consultas, de modo que la primera calcula el sueldo máximo de cada departamento:

```
SELECT   MAX (Sueldo) AS SueldoMax, Dno INTO TEMP
FROM     EMPLEADO
GROUP BY Dno;

SELECT   Dni
```

```
FROM      EMPLEADO, TEMP
WHERE     Sueldo = SueldoMax AND EMPLEADO.Dno = TEMP.Dno;
```

6. Si son posibles varias opciones para una condición de concatenación, elija una que utilice un índice agrupado y evite las que contienen comparaciones de cadena. Por ejemplo, asumiendo que el atributo Nombre es una clave candidata en EMPLEADO y ESTUDIANTE, es mejor utilizar EMPLEADO.Dni = ESTUDIANTE.Dni como condición de concatenación que EMPLEADO.Nombre = ESTUDIANTE.Nombre si Dni tiene un índice agrupado en una o en las dos tablas.
7. Una rareza de los optimizadores de consultas es que el orden de las tablas en la cláusula FROM puede afectar al procesamiento de la concatenación. Si es el caso, es posible tener que cambiar este orden para que se explore la más pequeña de las dos relaciones y se utilice la más grande con un índice adecuado.
8. Algunos optimizadores de consultas funcionan peor con las consultas anidadas que con las no anidadas. Hay cuatro tipos de consultas anidadas:
   - Subconsultas no correlativas con agregados en la consulta interior.
   - Subconsultas no correlativas sin agregados.
   - Subconsultas correlativas con agregados en la consulta interior.
   - Subconsultas correlativas sin agregados.

   De estos cuatro tipos, el primero normalmente no presenta problemas, puesto que la mayoría de los optimizadores evalúan la consulta interior una vez. Sin embargo, para una consulta del segundo tipo, como la del ejemplo del punto 2 anterior, la mayoría de los optimizadores de consultas no pueden utilizar un índice sobre Dno en EMPLEADO. Los mismos optimizadores lo podrán hacer si la consulta se escribe como una consulta no anidada. La transformación de las subconsultas correlativas puede implicar la configuración de tablas temporales. Mostrar ejemplos detallados queda fuera de los objetivos de este libro.[6]
9. Por último, muchas aplicaciones están basadas en vistas que definen los datos de interés para esas aplicaciones. A veces, una consulta puede plantearse directamente contra una tabla base, en lugar de pasar por una vista definida por una JOIN.

### 16.2.4 Directrices adicionales para la refinación de una consulta

En determinadas situaciones se aplican técnicas adicionales para mejorar las consultas:

1. Una consulta con varias condiciones de selección conectadas mediante OR no puede estar instando al optimizador de consultas a utilizar un índice. Dicha consulta puede dividirse y expresarse como una unión de consultas, cada una con una condición sobre un atributo que provoca la utilización de un índice. Por ejemplo,

```
SELECT    Nombre, Apellidos, Sueldo, Edad[7]
FROM      EMPLEADO
WHERE     Edad > 45 OR Sueldo < 50000;
```

puede ejecutarse con una exploración secuencial ofreciendo un rendimiento pobre. Con una división de este tipo:

---

[6] Si desea más información, consulte Shasha (1992).

[7] Hemos modificado el esquema y utilizado Edad en EMPLEADO en lugar de FechaNac.

```
SELECT    Nombre, Apellidos, Sueldo, Edad
FROM      EMPLEADO
WHERE     Edad > 45

UNION

SELECT    Nombre, Apellidos, Sueldo, Edad
FROM      EMPLEADO
WHERE     Sueldo < 50000;
```

pueden utilizarse los índices por Edad y Sueldo.

2. Para acelerar una consulta, se pueden intentar las siguientes transformaciones:
   - La condición NOT puede transformarse en una expresión positiva.
   - Los bloques SELECT incrustados con IN, = ALL, = ANY y = SOME pueden reemplazarse por concatenaciones.
   - Si se configura una concatenación de igualdad entre dos tablas, el predicado de rango (condición de selección) en el atributo de concatenación configurado en una tabla puede repetirse para la otra tabla.
3. Las condiciones WHERE pueden reescribirse para que utilicen los índices de varias columnas. Por ejemplo,

```
SELECT    NumRegión, TipoProd, Mes, Ventas
FROM      ESTADIS_VENTAS
WHERE     NumRegión = 3 AND ((TipoProd BETWEEN 1 AND 3) OR
          (TipoProd BETWEEN 8 AND 10));
```

puede utilizar sólo un índice por NumRegion y buscar por todas las páginas hoja del índice una coincidencia de TipoProd. En cambio, con

```
SELECT    NumRegión, TipoProd, Mes, Ventas
FROM      ESTADIS_VENTAS
WHERE     (NumRegión = 3 AND (TipoProd BETWEEN 1 AND 3))
          OR (NumRegión = 3 AND (TipoProd BETWEEN 8 AND 10));
```

puede utilizarse un índice compuesto (NumRegion, TipoProd) y funcionar de un modo mucho más eficaz.

En esta sección hemos cubierto la mayoría de las situaciones comunes en que la ineficacia de una consulta puede corregirse con alguna acción correctiva sencilla, como el uso de un temporal, evitar ciertos tipos de estructuras, o evitar el uso de vistas. El objetivo es utilizar, siempre que sea posible, los índices de un solo atributo o compuestos por varios atributos existentes. Esto evita la exploración completa de bloques de datos o de nodos hoja del índice. Los procesos redundantes, como la ordenación, deben evitarse a toda costa. Los problemas y los remedios dependerán de los trabajos de un optimizador de consultas dentro de un RDBMS. Los fabricantes de RDBMSs ofrecen a los administradores de bases de datos información detallada en forma de manuales.

# 16.3 Resumen

En este capítulo hemos visto los factores que afectan a las decisiones sobre el diseño físico de una base de datos, y ofrece directrices para elegir entre distintas alternativas de diseño físico. Como parte de la refinación

de una base de datos, hemos hablado de los cambios en el diseño lógico, las modificaciones de indexación y la introducción de cambios en las consultas. Lo explicado es sólo una muestra representativa de una gran cantidad de medidas y técnicas adoptadas en el diseño de grandes aplicaciones comerciales de DBMSs relacionales.

## Preguntas de repaso

**16.1.** ¿Cuáles son los factores que influyen en el diseño físico de una base de datos?

**16.2.** Explique las decisiones tomadas durante el diseño físico de una base de datos.

**16.3.** Explique las directrices de los RDBMSs en cuanto al diseño físico de una base de datos.

**16.4.** Explique los tipos de modificaciones que pueden aplicarse al diseño lógico de una base de datos relacional.

**16.5.** ¿Bajo qué situaciones se utilizaría la desnormalización del esquema de una base de datos? Exponga algunos ejemplos de desnormalización.

## Bibliografía seleccionada

Wiederhold (1986) abarca todas las fases del diseño de una base de datos, haciendo hincapié en el diseño físico. O'Neil (1994) ofrece una detallada explicación del diseño físico y de los problemas de transacción en referencia a los RDBMSs comerciales. Navathe y Kerschberg (1986) explica todas las fases del diseño de una base de datos y apunta el rol de los diccionarios de datos. Rozen y Shasha (1991) y Carlis y March (1984) presentan diferentes modelos para el problema del diseño físico de una base de datos. Shasha (1992) desarrolló unas directrices para la refinación de las bases de datos, que también aparecen, pero más elaboradas, en Shasha y Bonnett (2002).

PARTE 5

# Conceptos del procesamiento de transacciones

CAPÍTULO 17

# Introducción a los conceptos y la teoría sobre el procesamiento de transacciones

El concepto de transacción proporciona un mecanismo para definir las unidades lógicas del procesamiento de una base de datos. Los **sistemas de procesamiento de transacciones** son sistemas con grandes bases de datos y cientos de usuarios concurrentes ejecutando transacciones de bases de datos. Entre estos sistemas podemos citar los de reservas en aerolíneas, banca, procesamiento de tarjetas de crédito, mercado de acciones, etcétera. Estos sistemas requieren una alta disponibilidad y una respuesta rápida para cientos de usuarios simultáneos. En este capítulo presentamos los conceptos relacionados con los sistemas de procesamiento de transacciones. Definimos el concepto de transacción, que se utiliza para representar una unidad lógica de procesamiento de base de datos que debe completarse en su totalidad para garantizar su exactitud. También explicamos el problema de controlar la concurrencia, que surge cuando varias transacciones enviadas por varios usuarios interfieren entre sí de un modo que produce unos resultados incorrectos. Asimismo, explicamos cómo recuperarnos de los fallos en las transacciones.

La Sección 17.1 explica por qué el control de la concurrencia y la recuperación son necesarios en un sistema de bases de datos. La Sección 17.2 introduce el concepto de transacción y explica otros conceptos relacionados con el procesamiento de transacciones. La Sección 17.3 presenta los conceptos de atomicidad, conservación de la consistencia (o de la coherencia), aislamiento, y durabilidad o permanencia (denominados propiedades ACID), que se consideran deseables en las transacciones. La Sección 17.4 introduce el concepto de la planificación (*schedule*) de la ejecución de transacciones y tipifica la recuperabilidad de las planificaciones. La Sección 17.5 explica el concepto de serialización (o seriabilidad) de las ejecuciones de transacciones concurrentes, que también se puede utilizar para definir las secuencias de ejecución correctas (o planificaciones) de las transacciones concurrentes. La Sección 17.6 presenta los servicios que soportan el concepto de transacción en SQL.

Los dos capítulos siguientes continúan con más detalles sobre las técnicas utilizadas para el procesamiento de transacciones. El Capítulo 18 describe las técnicas básicas de control de la concurrencia, y el Capítulo 19 presenta una panorámica de las técnicas de recuperación.

## 17.1 Introducción al procesamiento de transacciones

En esta sección introducimos, informalmente, los conceptos de ejecución concurrente de transacciones y recuperación ante el fallo de una transacción. La Sección 17.1.1 compara los sistemas de bases de datos mono-

usuario y multiusuario, y muestra cómo la ejecución concurrente de transacciones puede tener lugar en los sistemas multiusuario. La Sección 17.1.2 define el concepto de transacción y presenta un modelo sencillo de ejecución de transacciones (basado en operaciones de lectura y escritura en bases de datos) que se utiliza para formalizar los conceptos de control de la concurrencia y recuperación. La Sección 17.1.3 muestra, mediante ejemplos informales, por qué son necesarias las técnicas de control de la concurrencia en los sistemas multiusuario. Por último, la Sección 17.1.4 explica por qué se necesitan técnicas que permitan la recuperación ante un fallo, y lo hace explicando las distintas maneras en que las transacciones pueden fallar mientras se ejecutan.

### 17.1.1 Sistema monousuario frente a sistema multiusuario

Un criterio que sirve para clasificar un sistema de bases de datos es el número de usuarios que lo pueden utilizar **al mismo tiempo**. Un DBMS es **monousuario** si sólo lo puede utilizar un usuario a la vez, y es **multiusuario** si varios usuarios pueden utilizar el sistema (y, por tanto, acceder a la base de datos) simultáneamente. Los DBMSs monousuario están principalmente restringidos a los sistemas de computación personal; la mayoría de los demás DBMSs son multiusuario. Por ejemplo, un sistema de reservas en aerolíneas lo utilizan simultáneamente cientos de agentes de viajes. Los sistemas de bancos, aseguradoras, supermercados, etcétera, también operan con muchos usuarios que envían transacciones simultáneamente al sistema.

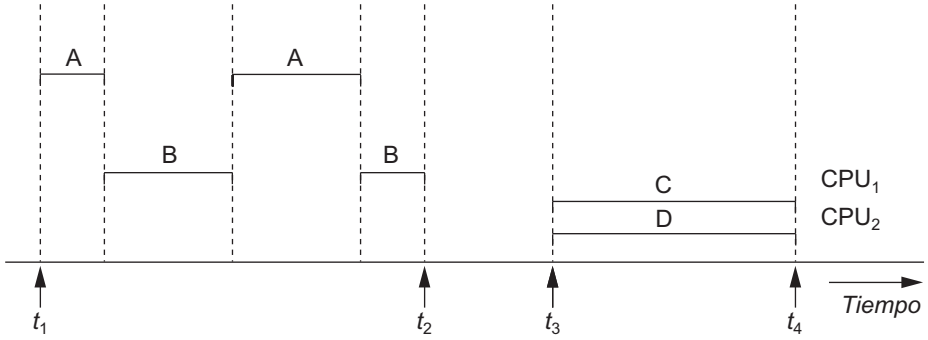
A la base de datos pueden acceder simultáneamente múltiples usuarios (y utilizar los computadores) debido al concepto de **multiprogramación**, que permite al computador ejecutar varios programas (o **procesos**) al mismo tiempo. Si sólo hay una unidad central de procesamiento (CPU), realmente sólo se ejecuta un proceso a la vez. Sin embargo, los **sistemas operativos multiprogramación** ejecutan algunos comandos de un proceso, después suspenden ese proceso y ejecutan algunos comandos del siguiente proceso, y así sucesivamente. El proceso se reanuda en el mismo punto en que se suspendió, siempre que consiga turno para utilizar de nuevo la CPU. Por tanto, la ejecución concurrente de procesos realmente es **interpolada**, como se ilustra en la Figura 17.1, que muestra dos procesos A y B ejecutándose concurrentemente en modo interpolado. La interpolación mantiene ocupada la CPU cuando un proceso requiere una operación de entrada o salida (E/S), como la lectura de un bloque del disco. La CPU pasa a ejecutar otro proceso en lugar de permanecer ociosa durante el tiempo de E/S. La interpolación también evita que un proceso largo retrase otros procesos.

Si el computador tiene varios procesadores (CPUs), es posible el **procesamiento paralelo** de varios procesos, como se ilustra en la Figura 17.1 con los procesos C y D. La mayoría de la teoría relativa al control de la concurrencia en las bases de datos está desarrollada en términos de **concurrencia interpolada**, por lo que en el resto de este capítulo asumimos este modelo. En un DBMS multiusuario, los elementos de datos almacenados son los recursos principales a los que pueden acceder los usuarios interactivos o programas de aplicación (que están recuperando y modificando constantemente información de la base de datos).

### 17.1.2 Transacciones, operaciones de lectura y escritura y búferes DBMS

Una **transacción** es un programa en ejecución que constituye una unidad lógica del procesamiento de una base de datos. Una transacción incluye una o más operaciones de acceso a la base de datos (operaciones de inserción, eliminación, modificación o recuperación). Las operaciones con bases de datos que forman una transacción pueden estar incrustadas dentro de una aplicación o pueden especificarse interactivamente mediante un lenguaje de consulta de alto nivel como SQL. Una forma de definir los límites de una transacción es especificando explícitamente las sentencias **begin transaction** y **end transaction** en un programa de aplicación; en este caso, todas las operaciones de acceso a la base de datos que se encuentran entre las dos sentencias son consideradas como una transacción. Un programa de aplicación puede contener más de una transacción si contiene varios límites de transacción. Si las operaciones de bases de datos de una transacción no actualizan la base de datos, sino que únicamente recuperan datos, se dice que la **transacción es de sólo lectura**.

**Figura 17.1.** Procesamiento interpolado frente a procesamiento paralelo de transacciones concurrentes.




El modelo de una base de datos que se utiliza para explicar los conceptos de procesamiento de transacciones es muy simplificado. Una **bases de datos** está representada básicamente como una colección de **elementos de datos con nombre**. El tamaño de un elemento de datos se conoce como **granularidad**. Puede ser un campo de algún registro de la base de datos, o puede ser una unidad más grande, como un registro o incluso un bloque de disco entero, pero los conceptos que explicamos son independientes de la granularidad del elemento de datos. Con este modelo de base de datos simplificado, las operaciones básicas de acceso a la base de datos que una transacción puede incluir son las siguientes:

- **read_item(*X*)** (leer elemento). Lee un elemento de base de datos denominado *X* y lo almacena en una variable de programa. Para simplificar nuestra notación, asumimos que *la variable de programa también se llama X*.
- **write_item(*X*)** (escribir elemento). Escribe el valor de la variable de programa *X* en un elemento de base de datos denominado *X*.

Como explicamos en el Capítulo 13, la unidad básica de transferencia de datos desde el disco a la memoria principal es un bloque. La ejecución de un comando read_item(*X*) incluye estos pasos:

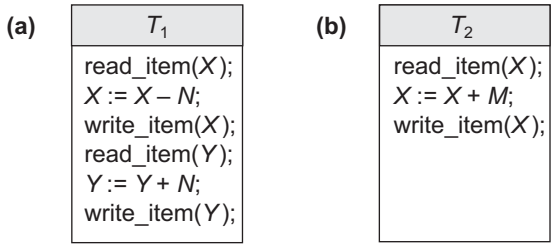
1. Encontrar la dirección del bloque de disco que contiene el elemento *X*.
2. Copiar ese bloque de disco en un búfer de la memoria principal (si dicho bloque no se encuentra ya en algún búfer de la memoria principal).
3. Copiar el elemento *X* desde el búfer a la variable de programa *X*.

La ejecución de un comando write_item(*X*) abarca estos pasos:

1. Encontrar la dirección del bloque de disco que contiene el elemento *X*.
2. Copiar ese bloque de disco en un búfer de la memoria principal (si dicho bloque no se encuentra ya en algún búfer de la memoria principal).
3. Copiar el elemento *X* desde la variable de programa *X* a su ubicación correcta en el búfer.
4. Almacenar el bloque actualizado desde el búfer al disco (puede ser inmediatamente o en un momento posterior en el tiempo).

El paso 4 realmente actualiza la base de datos en el disco. En algunos casos, el búfer no se almacena inmediatamente en el disco, en caso de haber introducido cambios adicionales en el búfer. Normalmente, la decisión sobre cuándo almacenar un bloque de disco modificado que se encuentra en un búfer de la memoria principal la toma el gestor de recuperación del DBMS en cooperación con el sistema operativo subyacente. El DBMS generalmente mantendrá varios **búferes** de la memoria principal para albergar los bloques de disco de la base de datos que contienen los elementos de datos que se están procesando. Cuando todos estos búferes están ocupados, y es preciso copiar otros bloques de la base de datos en la memoria, se utiliza alguna polí-

**Figura 17.2.** Dos ejemplos de transacciones. (a) Transacción $T_1$. (b) Transacción $T_2$.




tica de sustitución de búferes para elegir cuál de ellos sustituir. Si el búfer elegido se ha modificado, debe escribirse en el disco antes de su reutilización.[1]

Una transacción incluye operaciones read_item y write_item para acceder y actualizar la base de datos. La Figura 17.2 muestra ejemplos de dos transacciones muy sencillas. El **conjunto de lectura** (*read-set*) de una transacción es el conjunto de todos los elementos que la transacción lee, y el **conjunto de escritura** (*write-set*) es el conjunto de todos los elementos que la transacción escribe. Por ejemplo, el conjunto de lectura de $T_1$ en la Figura 17.2 es $\{X, Y\}$ y su conjunto de escritura también es $\{X, Y\}$.

Los mecanismos de control de la concurrencia y de recuperación están principalmente ligados a los comandos de acceso a la base de datos en una transacción. Las transacciones emitidas por varios usuarios pueden ejecutarse concurrentemente y pueden acceder y actualizar los mismos elementos de la base de datos. Si esta ejecución concurrente no está controlada, pueden surgir problemas, como una base de datos inconsistente (incoherente). En la siguiente sección introducimos algunos de los problemas que pueden surgir.

### 17.1.3 Por qué es necesario controlar la concurrencia

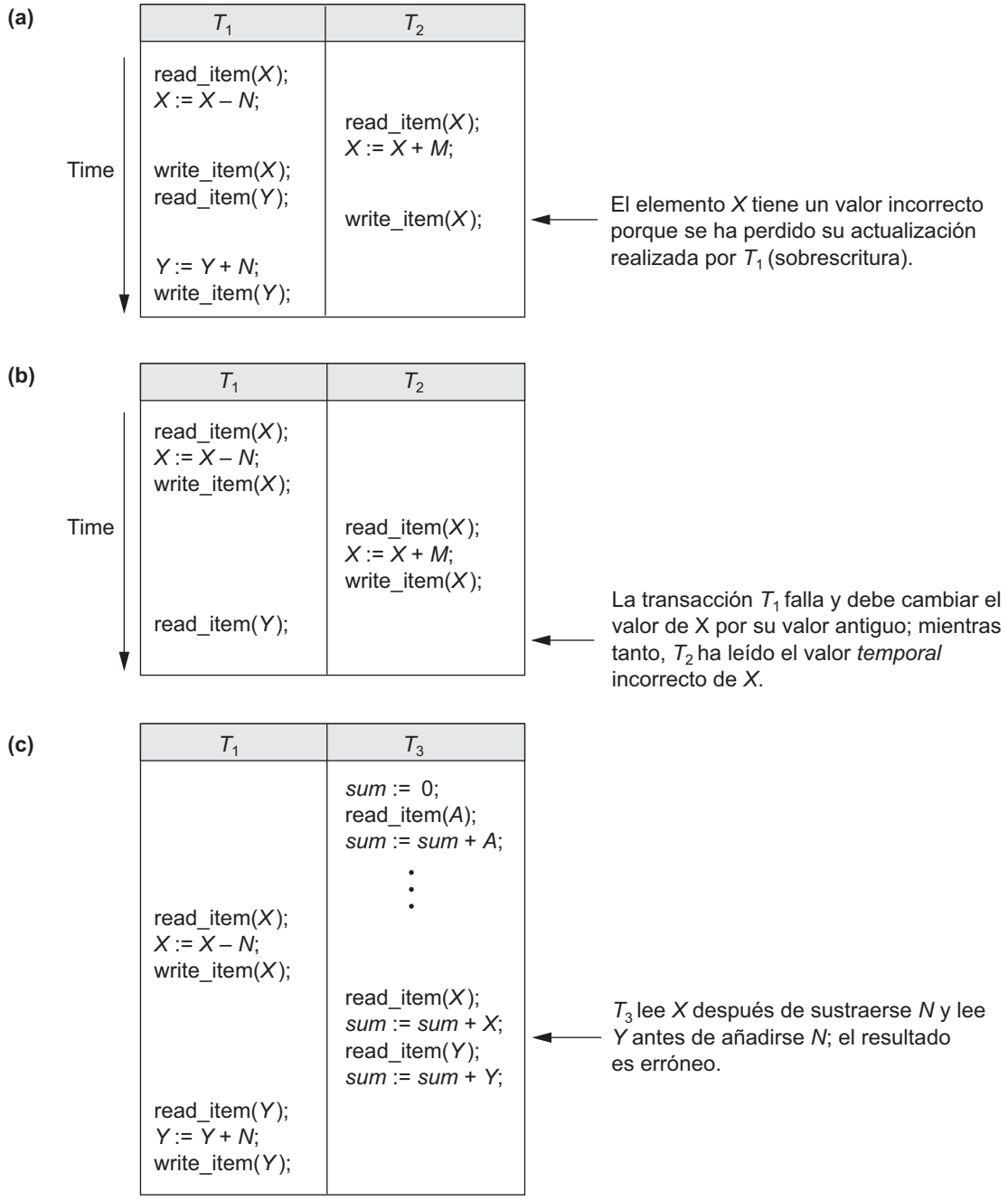
Cuando las transacciones se ejecutan de manera incontrolada pueden surgir algunos problemas. Vamos a ilustrarlos haciendo referencia a una base de datos de reservas en aerolíneas simplificada en la que se almacena un registro por cada vuelo de una aerolínea. Cada registro incluye el número de plazas reservadas en ese vuelo como un *elemento de datos con nombre*, junto con otra información. La Figura 17.2(a) muestra una transacción $T_1$ que *transfiere N* reservas de un vuelo cuyo número de plazas reservadas está almacenado en el elemento de base de datos $X$ a otro vuelo cuyo número de plazas reservadas se almacena en el elemento de base de datos $Y$. La Figura 17.2(b) muestra una transacción $T_2$ más sencilla que *reserva M* plazas en el primer vuelo ($X$) al que se hace referencia en la transacción $T_1$.[2] Para simplificar nuestro ejemplo, no mostramos ciertas porciones de las transacciones, como la comprobación de si un vuelo tiene suficientes plazas disponibles antes de efectuar la reserva de plazas adicionales.

Cuando se escribe un programa de acceso a una base de datos, tiene como parámetros los números de vuelo, sus fechas y el número de plazas que se pueden reservar; por tanto, el mismo programa puede utilizarse para ejecutar muchas transacciones, cada una con diferentes vuelos y cantidades de plazas a reservar. A fin de controlar la concurrencia, una transacción es una *ejecución particular* de un programa en una fecha, vuelo y número de plazas dadas. En la Figura 17.2(a) y (b), las transacciones $T_1$ y $T_2$ son *ejecuciones específicas* de los programas que se refieren a vuelos específicos cuyos números de plazas se almacenan en los elementos

[1] No explicaremos las políticas de sustitución de búferes porque normalmente se explican en los libros dedicados a los sistemas operativos.

[2] Un ejemplo parecido y muy utilizado es el de la base de datos de un banco, con una transacción haciendo una transferencia de fondos de la cuenta $X$ a la cuenta $Y$, y otra transacción haciendo un depósito en la cuenta $X$.

**Figura 17.3.** Algunos problemas que surgen cuando no se controla la concurrencia. (a) Problema por pérdida de actualización. (b) Problema de la actualización temporal. (c) Problema de la suma incorrecta.




de datos $X$ e $Y$ de la base de datos. A continuación explicamos los tipos de problemas que nos podemos encontrar con estas transacciones si se ejecutan simultáneamente.

**Problema por pérdida de actualización.** Este problema surge cuando dos transacciones que acceden a los mismos elementos de la base de datos tienen sus operaciones interpoladas de un modo que hace que el

valor de algunos elementos de la base de datos sean incorrectos. Suponga que las transacciones $T_1$ y $T_2$ se envían aproximadamente al mismo tiempo, y suponga que sus operaciones están interpoladas como se aprecia en la Figura 17.3(a); el valor final del elemento $X$ es incorrecto porque $T_2$ lee el valor de $X$ *antes* de que $T_1$ lo cambie en la base de datos; por tanto, se pierde el valor actualizado resultante de $T_1$. Por ejemplo, si $X = 80$ al principio (originalmente, el vuelo tiene 80 reservas), $N = 5$ ($T_1$ transfiere 5 reservas de plaza del vuelo correspondiente a $X$ al vuelo correspondiente a $Y$), y $M = 4$ ($T2$ reserva 4 plazas en $X$), el resultado final debe ser $X = 79$; pero en la interpolación de operaciones de la Figura 17.3(a), el resultado es $X = 84$ porque se ha perdido la actualización de $T_1$ que eliminaba las cinco plazas de $X$.

**Problema de la actualización temporal (o lectura sucia).** Este problema ocurre cuando una transacción actualiza un elemento de la base de datos y, después, falla por alguna razón (consulte la Sección 17.1.4). El elemento actualizado es accedido por otra transacción antes de cambiar a su valor original. La Figura 17.3(b) muestra un ejemplo donde $T_1$ actualiza el elemento $X$ y después falla antes de concluir, por lo que el sistema debe devolver $X$ a su valor original. Sin embargo, antes de poder hacerlo, la transacción $T_2$ lee el valor *temporal* de $X$, que no se grabará permanentemente en la base de datos debido al fallo de $T_1$. El valor del elemento $X$ que es leído por $T_2$ se denomina *dato sucio* porque lo ha creado una transacción que no se ha completado y confirmado todavía; por tanto, este problema también se conoce como *problema de la lectura sucia*.

**El problema de la suma incorrecta.** Si una transacción está calculando una función de suma agregada sobre varios registros, mientras otras transacciones están actualizando algunos de esos registros, la función agregada puede calcular algunos valores antes de que sean actualizados y otros después de ser actualizados. Por ejemplo, suponga que una transacción $T_3$ está calculando el número total de reservas en todos los vuelos; mientras tanto, se está ejecutando la transacción $T_1$. Si se produce la interpolación de operaciones de la Figura 17.3(c), el resultado de $T_3$ estará desfasado una cantidad $N$ porque $T_3$ lee el valor de $X$ *después* de que se hayan sustraído $N$ plazas de ella, pero lee el valor de $Y$ *antes* de que esas $N$ plazas se hayan añadido a él.

Otro problema que puede surgir es la denominada **lectura irrepetible**, en la que una transacción $T$ lee un elemento dos veces y el elemento es modificado por otra transacción $T'$ entre las dos lecturas. Por tanto, $T$ recibe *valores diferentes* en sus dos lecturas del mismo elemento. Esto sucede, por ejemplo, si durante una tran-sacción de reserva en una aerolínea, un cliente busca información sobre la disponibilidad de plaza en varios vuelos. Cuando el cliente opta por un vuelo en particular, la transacción lee el número de plazas de ese vuelo una segunda vez antes de completar la reserva.

### 17.1.4 Por qué es necesaria la recuperación

Siempre que se envía una transacción a un DBMS para su ejecución, el sistema es responsable de garantizar que todas las operaciones de la transacción se completen satisfactoriamente y que su efecto se grabe permanentemente en la base de datos, o de que la transacción no afecte a la base de datos o a cualquier otra transacción. El DBMS no debe permitir que algunas operaciones de una transacción $T$ se apliquen a la base de datos mientras otras no. Esto puede ocurrir si una transacción **falla** después de ejecutar algunas de sus operaciones, pero antes de ejecutar todas ellas.

**Tipos de fallos.** Los fallos se clasifican generalmente como fallos de transacción del sistema y del medio. Hay varias razones posibles por las que una transacción puede fallar en medio de su ejecución:

1. **Un fallo del computador (caída del sistema).** Durante la ejecución de una transacción se produce un error del hardware, del software o de la red. Las caídas del hardware normalmente se deben a fallos en los medios (por ejemplo, un fallo de la memoria principal).
2. **Un error de la transacción o del sistema.** Alguna operación de la transacción puede provocar que falle, como un desbordamiento de entero o una división por cero. El fallo de una transacción también

se puede deber a unos valores erróneos de los parámetros o debido a un error lógico de programación.[3] Además, el usuario puede interrumpir la transacción durante su ejecución.

3. **Errores locales o condiciones de excepción detectados por la transacción.** Durante la ejecución de una transacción, se pueden dar ciertas condiciones que necesitan cancelar la transacción. Por ejemplo, puede que no se encuentren los datos para la transacción. Una condición de excepción,[4] como un saldo de cuenta insuficiente en una base de datos bancaria, puede provocar que una transacción, como la retirada de fondos, sea cancelada. Esta excepción debe programarse en la propia transacción, en cuyo caso, no sería considerada un fallo.
4. **Control de la concurrencia.** El método de control de la concurrencia (consulte el Capítulo 18) puede optar por abortar la transacción, para restablecerla más tarde, porque viola la serialización (consulte la Sección 17.5) o porque varias transacciones se encuentran en estado de bloqueo.
5. **Fallo del disco.** Algunos bloques del disco pueden perder sus datos debido a un mal funcionamiento de la lectura o la escritura o porque se ha caído la cabeza de lectura/escritura del disco. Esto puede ocurrir durante una operación de lectura o escritura de la transacción.
6. **Problemas físicos y catástrofes.** Se refiere a una lista interminable de problemas que incluye fallos de alimentación o aire acondicionado, fuego, robo, sabotaje, sobrescritura de discos y cintas por error, y montaje de la cinta errónea por parte del operador.

Los fallos de los tipos 1, 2, 3 y 4 son más comunes que los de los tipos 5 o 6. Siempre que se produce un fallo de tipo 1 a 4, el sistema debe guardar información suficiente para recuperarse del fallo. El fallo de un disco u otros fallos catastróficos de tipo 5 o 6 no se dan con frecuencia; si ocurren, la recuperación es la tarea principal. En el Capítulo 19 explicamos cómo recuperarnos de un fallo.

El concepto de transacción es fundamental para muchas técnicas de control de la concurrencia y de recuperación ante fallos.

# 17.2 Conceptos de transacción y sistema

En esta sección explicamos conceptos adicionales relativos al procesamiento de transacciones. La Sección 17.2.1 describe los distintos estados de una transacción y explica las operaciones pertinentes adicionales necesarias en su procesamiento. La Sección 17.2.2 explica el registro del sistema, que guarda información necesaria para la recuperación. La Sección 17.2.3 describe los puntos de confirmación (*commit points*) de las transacciones, y por qué son importantes en el procesamiento de las mismas.

## 17.2.1 Estados de una transacción y operaciones adicionales

Una transacción es una unidad atómica de trabajo que se completa en su totalidad o no se lleva a cabo en absoluto. Para fines de recuperación, el sistema debe hacer el seguimiento de cuándo se inicia, termina y confirma o aborta una transacción (consulte la Sección 17.2.3). Por consiguiente, el gestor de recuperación hace un seguimiento de las siguientes operaciones:

- BEGIN_TRANSACTION. Marca el inicio de la ejecución de una transacción.
- READ o WRITE. Especifican operaciones de lectura o escritura en los elementos de la base de datos que se ejecutan como parte de una transacción.

[3] En general, una transacción debe probarse cuidadosamente para asegurarnos de que no tiene errores (errores de programación lógicos).

[4] Las condiciones de excepción, si se programan correctamente, no constituyen fallos de transacción.

- END_TRANSACTION. Especifica que las operaciones READ y WRITE de la transacción han terminado y marca el final de la ejecución de la transacción. Sin embargo, en este punto puede ser necesario comprobar si los cambios introducidos por la transacción pueden aplicarse de forma permanente a la base de datos (confirmados) o si la transacción se ha cancelado porque viola la serialización (consulte la Sección 17.5) o por alguna otra razón.
- COMMIT_TRANSACTION. Señala una *finalización satisfactoria* de la transacción, por lo que los cambios (actualizaciones) ejecutados por la transacción se pueden **enviar** con seguridad a la base de datos y no se desharán.
- ROLLBACK (o ABORT). Señala que la transacción *no ha terminado satisfactoriamente*, por lo que deben *deshacerse* los cambios o efectos que la transacción pudiera haber aplicado a la base de datos.

La Figura 17.4 muestra un diagrama que describe los estados de ejecución de una transacción. Una transacción entra en **estado activo** inmediatamente después de iniciarse su ejecución; en este estado puede emitir operaciones READ y WRITE. Cuando la transacción termina, pasa al **estado de parcialmente confirmada**. En este punto, se necesitan algunos protocolos de recuperación para garantizar que un fallo del sistema no supondrá la imposibilidad de registrar los cambios de la transacción de forma permanente (normalmente, grabando los cambios en el registro del sistema, que explicamos en la siguiente sección).[5] Una vez que esta comprobación es satisfactoria, se dice que la transacción ha alcanzado su punto de confirmación y entra en el **estado de confirmada**. Los puntos de confirmación se explican en detalle en la Sección 17.2.3. Una vez confirmada la transacción, ha concluido satisfactoriamente su ejecución y todos sus cambios deben grabarse permanentemente en la base de datos.

No obstante, una transacción puede entrar en el **estado de fallo** si falla una de las comprobaciones o si la transacción es cancelada durante su estado activo. Entonces es posible anular la transacción para deshacer el efecto de sus operaciones de escritura en la base de datos. El **estado terminado** se alcanza cuando la transacción abandona el sistema. La información relativa a la transacción que se guarda en tablas del sistema mientras se está ejecutando, se elimina cuando la transacción termina. Las transacciones fallidas o canceladas pueden *restablecerse* más tarde (automáticamente o después de haber sido reenviadas por el usuario) como transacciones completamente nuevas.

### 17.2.2 El registro del sistema (*log*)

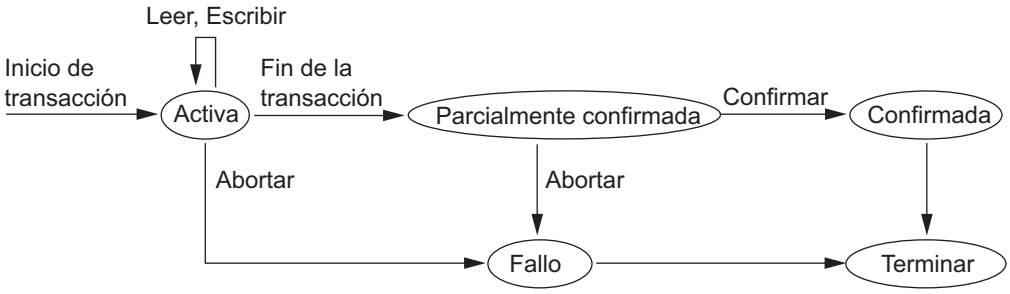
Para poder recuperarse de los fallos que afectan a las transacciones, el sistema mantiene un **registro**[6] para hacer un seguimiento de todas las operaciones de las transacciones que afectan a los valores de los elementos de una base de datos. Esta información puede ser necesaria para recuperarse ante un fallo. Este registro se guarda en el disco, por lo que no se ve afectado por ningún tipo de fallo, excepto los de disco o catastróficos. Además, periódicamente se realiza una copia de seguridad del registro para protegerse contra dichos fallos catastróficos. La siguiente lista muestra los tipos de entradas (denominadas **entradas del registro**) que se escriben en el registro y la acción que cada una de ellas lleva a cabo. En estas entradas, *T* se refiere a un **id de transacción** único que el sistema genera automáticamente y que utiliza para identificar las transacciones:

1. [**start_transaction**, *T*]. Indica que se ha iniciado la ejecución de la transacción *T*.
2. [**write_item**, *T*, *X*, *valor_antiguo*, *valor_nuevo*]. Indica que la transacción *T* ha cambiado el valor del elemento *X* de *valor_antiguo* a *valor_nuevo*.

[5] Un control optimista de la concurrencia (consulte la Sección 18.4) también requiere que en este punto se realicen ciertas comprobaciones que garanticen que la transacción no interfiere con otras transacciones en ejecución.

[6] El registro se ha denominado a veces diario del DBMS.

**Figura 17.4.** Diagrama de estado de una transacción ilustrando los estados de su ejecución.




3. [**read_item**, *T*, *X*]. Indica que la transacción *T* ha leído el valor del elemento *X* de la base de datos.
4. [**commit**, *T*]. Indica que la transacción *T* se ha completado satisfactoriamente, y afirma que su efecto puede confirmarse (grabarse permanentemente) en la base de datos.
5. [**abort**, *T*]. Indica que la transacción *T* se abortó.

Los protocolos para recuperación que evitan las anulaciones en cascada (consulte la Sección 17.4.2), entre los que podemos citar casi todos los protocolos prácticos, no requieren que las operaciones de lectura se escriban en el registro del sistema. Sin embargo, si el registro también se utiliza con otros fines (como la auditoría, que hace un seguimiento de todas las operaciones de bases de datos), entonces pueden incluirse esas entradas. Además, algunos protocolos de recuperación requieren entradas WRITE más sencillas que no incluyen *valor_nuevo* (consulte la Sección 17.4.2).

Observe que estamos asumiendo que todos los cambios permanentes en la base de datos ocurren dentro de las transacciones, por lo que la noción de recuperarse ante el fallo de una transacción equivale a deshacer o rehacer individualmente las operaciones de esa transacción desde el registro. Si el sistema se cae, podemos recuperar un estado consistente de la base de datos examinando el registro y utilizando una de las técnicas que se describen en el Capítulo 19. Como el registro del sistema contiene un registro o entrada por cada operación de escritura que cambia el valor de algún elemento de la base de datos, es posible **deshacer** el efecto de esas operaciones de escritura de una transacción *T* rastreando hacia atrás el registro y restableciendo todos los elementos modificados por una operación de escritura de *T* a sus valores antiguos (*valor_antiguo*). También es posible que tengamos que **rehacer** las operaciones de una transacción si todas sus actualizaciones se grabaron en el registro pero se produjo un fallo antes de poder garantizarse que todos los *valores_nuevos* se grabaron permanentemente en la base de datos.[7] La tarea de rehacer las operaciones de la transacción *T* se consigue moviéndonos hacia delante en el registro y configurando todos los elementos modificados por una operación de escritura de *T* a sus *valores_nuevos*.

### 17.2.3 Punto de confirmación de una transacción

Una transacción *T* alcanza su **punto de confirmación** cuando todas sus operaciones que acceden a la base de datos se han ejecutado satisfactoriamente y el efecto de todas ellas se ha grabado en el registro. Más allá del punto de confirmación, se dice que la transacción se ha **confirmado**, y se asume que su efecto se ha *registrado permanentemente* en la base de datos. La transacción escribe entonces un registro de envío [commit, *T*] en el registro del sistema. Si se produce un fallo del sistema, se buscan hacia atrás en el registro del sistema todas las transacciones *T* que han escrito un registro [start_transaction, *T*] pero que no han escrito todavía su registro [commit, *T*]; es posible que haya que anular estas transacciones para deshacer su efecto en la base de datos durante el proceso de recuperación. Las transacciones que han escrito su registro *commit* en el registro del

[7] Las tareas de rehacer y deshacer se explican más en profundidad en el Capítulo 19.

sistema también deben haber registrado todas sus operaciones de escritura, para que su efecto en la base de datos pueda *rehacerse* a partir de las entradas del registro.

Observe que el fichero del registro debe guardarse en el disco. Como explicamos en el Capítulo 13, la actualización de un fichero en disco supone copiar el bloque adecuado del fichero a un búfer de la memoria principal, actualizar el búfer en la memoria principal y copiar el búfer en el disco. Es común mantener uno o más bloques del fichero del registro del sistema en los búferes de la memoria principal hasta que se llenan para después escribirlos de una sola vez en el disco, en lugar de escribir en disco cada vez que se añade una entrada al registro. Esto ahorra el coste que suponen varias escrituras en disco del mismo bloque del fichero del registro. Cuando el sistema se cae, el proceso de recuperación sólo tiene en cuenta las entradas del registro que se *escribieron en el disco*, porque el contenido de la memoria principal se puede haber perdido. Por tanto, *antes* de que una transacción alcance su punto de confirmación, cualquier porción del registro que no se haya escrito todavía en el disco, debe escribirse ahora. Este proceso se denomina **escritura forzosa** del fichero del registro antes de la confirmación de una transacción.

## 17.3 Propiedades deseables de las transacciones

Las transacciones deben poseer varias propiedades, a menudo denominadas propiedades **ACID**, que deben ser implementadas por el control de la concurrencia y los métodos de recuperación del DBMS. Las propiedades ACID son las siguientes:

- **Atomicidad.** Una transacción es una unidad atómica de procesamiento; o se ejecuta en su totalidad o no se ejecuta en absoluto.
- **Conservación de la consistencia.** Una transacción está conservando la consistencia si su ejecución completa lleva a la base de datos de un estado consistente a otro.
- **Aislamiento.** Una transacción debe aparecer como si estuviera ejecutándose de forma aislada a las demás. Es decir, la ejecución de una transacción no debe interferir con la ejecución de ninguna otra transacción simultánea.
- **Durabilidad.** Los cambios aplicados a la base de datos por una transacción confirmada deben persistir en la base de datos. Estos cambios no deben perderse por culpa de un fallo.

La propiedad de la atomicidad requiere que la transacción se ejecute hasta su finalización. El subsistema de recuperación de transacciones del DBMS tiene la responsabilidad de garantizar la atomicidad. Si una transacción falla y no se completa por alguna razón (por ejemplo, por una caída del sistema en medio de su ejecución), la técnica de recuperación debe deshacer sus efectos en la base de datos.

Se considera que la conservación de la consistencia es responsabilidad de los programadores que escriben los programas de bases de datos o del módulo DBMS que implementa las restricciones de integridad. Recuerde que un **estado de la base de datos** es una colección de todos los elementos de datos (valores) almacenados en la base de datos en un momento dado. Un **estado consistente** de la base de datos satisface las restricciones especificadas en el esquema, así como cualquier otra restricción que deba cumplirse en la base de datos. Un programa de base de datos debe escribirse de forma que garantice que, si la base de datos se encuentra en un estado consistente antes de ejecutar la transacción, estará en un estado consistente después de haberse completado la ejecución de la transacción, asumiendo que *no interfiere con ninguna otra transacción*.

El aislamiento es tarea del subsistema de control de la concurrencia del DBMS.[8] Si cada transacción no hace visibles sus actualizaciones a otras transacciones hasta que se confirma, se implementa una forma de aislamiento que soluciona el problema de la actualización temporal y elimina las anulaciones en cascada (consulte el Capítulo 19). Ha habido intentos de definir el **nivel de aislamiento** de una transacción. Una transacción

[8] En el Capítulo 18 veremos los protocolos de control de la concurrencia.

tiene un nivel 0 (cero) de aislamiento si no sobrescribe las lecturas sucias de las transacciones de nivel más alto. El aislamiento de nivel 1 (uno) no pierde actualizaciones; y el aislamiento de nivel 2 no pierde actualizaciones y no hace lecturas sucias. Por último, el aislamiento de nivel 3 (también conocido como *aislamiento verdadero*) tiene, además de las propiedades del grado 2, lecturas que se pueden repetir.

Por último, la propiedad de la durabilidad es responsabilidad del subsistema de recuperación del DBMS. En el Capítulo 19 veremos cómo los protocolos de recuperación implementan la durabilidad y la atomicidad.

# 17.4 Clasificación de las planificaciones en base a la recuperabilidad

Cuando las transacciones se están ejecutando concurrentemente en modo interpolado, el orden de ejecución de las operaciones de las distintas transacciones se conoce como **planificación**. En esta sección, primero definimos el concepto de planificación, y después veremos los tipos de planificaciones que posibilitan la recuperación cuando se produce un fallo. En la Sección 17.5 clasificamos las planificaciones en términos de interferencia de las transacciones participantes, lo que nos llevará a los conceptos de serialización y planificaciones serializables.

## 17.4.1 Planificaciones de transacciones

Una **planificación** $S$ de $n$ transacciones $T_1$, $T_2$, ..., $T_n$ es una ordenación de las operaciones de esas transacciones sujeta a la restricción de que, por cada transacción $T_i$ que participa en $S$, las operaciones de $T_i$ en $S$ deben aparecer en el mismo orden en el que tienen lugar en $T_i$. No obstante, observe que las operaciones de otras transacciones $T_j$ pueden interpolarse con las operaciones de $T_i$ en $S$. Por ahora, considere que el orden de las operaciones en $S$ responde a una *ordenación total*, aunque es teóricamente posible tratar con planificaciones cuyas operaciones forman *ordenaciones parciales* (como veremos más tarde).

En lo referente a la recuperación y el control de la concurrencia, nuestro interés se centra en las operaciones read_item y write_item de las transacciones, así como en las operaciones commit y abort. Una notación abreviada para describir una planificación utiliza los símbolos $r$, $w$, $c$ y $a$ para las operaciones read_item, write_item, commit y abort, respectivamente, y añade como subíndice el id de la transacción (número de la transacción) a cada operación de la planificación. En esta notación, el elemento $X$ de la base de datos que se lee o escribe se escribe entre paréntesis a continuación de las operaciones $r$ y $w$. Por ejemplo, la planificación de la Figura 17.3(a), que llamaremos $S_a$, puede escribirse de este modo según esta notación:

$$S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$$

De forma parecida, la planificación para la Figura 17.3(b), que llamaremos $S_b$, puede escribirse de este modo, si asumimos que la transacción $T_1$ abortó después de su operación read_item($Y$):

$$S_b: r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); a_1;$$

Dos operaciones de una planificación entran en **conflicto** si satisfacen estas tres condiciones: (1) pertenecen a transacciones diferentes; (2) acceden al mismo elemento $X$; y (3) al menos una de las operaciones es write_item($X$). Por ejemplo, en la planificación $S_a$, las operaciones $r_1(X)$ y $w_2(X)$ están en conflicto, al igual que $r_2(X)$ y $w_1(X)$, y $w_1(X)$ y $w_2(X)$. Sin embargo, las operaciones $r_1(X)$ y $r_2(X)$ no están en conflicto, puesto que se trata de dos operaciones de lectura; las operaciones $w_2(X)$ y $w_1(Y)$ no entran en conflicto porque operan sobre elementos de datos distintos, $X$ e $Y$; y las operaciones $r_1(X)$ y $w_1(X)$ tampoco entran en conflicto porque pertenecen a la misma transacción.

Se dice que una planificación $S$ de $n$ transacciones $T_1$, $T_2$, ..., $T_n$ es una **planificación completa** si se dan las siguientes condiciones:

1. Las operaciones en $S$ son exactamente las operaciones en $T_1$, $T_2$, ..., $T_n$, incluyendo una operación commit o abort como última operación de cada transacción de la planificación.
2. Para cualquier par de operaciones de la misma transacción $T_i$, su orden de aparición en $S$ es el mismo que su orden de aparición en $T_i$.
3. Para cualesquiera dos operaciones en conflicto, una de las dos debe producirse antes que la otra en la planificación.[9]

La condición 3 permite que dos *operaciones no conflictivas* ocurran en la planificación sin tener que definir cuál de ellas se produce primero, lo que nos lleva a la definición de una planificación como una **ordenación parcial** de las operaciones en las $n$ transacciones.[10] Sin embargo, en la planificación debe especificarse un orden total para cualquier par de operaciones conflictivas (condición 3) y para cualquier par de operaciones de la misma transacción (condición 2). La condición 1 simplemente dice que todas las operaciones de las transacciones deben aparecer en la planificación completa. Como cada transacción se confirma o anula, una planificación completa no contendrá ninguna transacción activa al final.

En general, es difícil encontrar planificaciones completas en un sistema de procesamiento de transacciones porque se envían transacciones nuevas continuamente al sistema. Por tanto, es útil definir el concepto de **proyección confirmada** $C(S)$ de una planificación $S$, que sólo incluye las operaciones de $S$ que pertenecen a las transacciones confirmadas (es decir, las transacciones $T_i$ cuya operación commit $c_i$ está en $S$).

### 17.4.2 Clasificación de las planificaciones en base a la recuperabilidad

Con algunas planificaciones es fácil la recuperación ante fallos, mientras que no lo es tanto con otras. Por tanto, es importante distinguir los tipos de planificaciones para los que la recuperación es posible, así como aquellas para las que la recuperación es relativamente simple. Estas clasificaciones no proporcionan realmente el algoritmo de recuperación; sólo intentan clasificar teóricamente los distintos tipos de planificaciones.

En primer lugar, nos gustaría asegurar que, una vez confirmada una transacción $T$, *nunca* debe ser necesario anularla. Las planificaciones que teóricamente cumplen este criterio, se denominan planificaciones recuperables, y aquellas que no lo cumplen, **irrecuperables** (y, por tanto, no deben permitirse). Una planificación $S$ es recuperable si ninguna transacción $T$ en $S$ se confirma hasta que todas las transacciones $T'$ que han escrito un elemento que $T$ lee se han confirmado. Una transacción $T$ **lee** de la transacción $T'$ en una planificación $S$ si algún elemento $X$ es escrito en primer lugar por $T'$ y más tarde leído por $T$. Además, $T'$ no debe cancelarse antes de que $T$ lea el elemento $X$, y no puede haber transacciones que escriban $X$ después de que $T'$ lo escriba y antes de que $T$ lo lea (a menos que esas transacciones, si las hay, se hayan cancelado antes de que $T$ lea $X$).

Como veremos, las planificaciones recuperables requieren un proceso de recuperación más complejo, pero si se guarda la información suficiente (en el registro), podemos idear un algoritmo de recuperación. Las planificaciones (parciales) $S_a$ y $S_b$ de la sección anterior son recuperables, ya que satisfacen la definición anterior. Considere la siguiente planificación, $S_a'$, que es idéntica a $S_a$ excepto que se han añadido dos operaciones commit a $S_a$:

$$S_a'\text{: } r_1(X);\ r_2(X);\ w_1(X);\ r_1(Y);\ w_2(X);\ c_2;\ w_1(Y);\ c_1;$$

[9] En teoría, no es necesario determinar un orden entre pares de operaciones *no conflictivas*.

[10] En la práctica, la mayoría de las planificaciones tienen un orden total de las operaciones. Si se emplea el procesamiento paralelo, es teóricamente posible tener planificaciones con operaciones no conflictivas parcialmente ordenadas.

$S_a$' es recuperable, aunque padece el problema de la pérdida de actualización. Sin embargo, considere estas dos planificaciones (parciales), $S_c$ y $S_d$:

$S_c$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $c_2$; $a_1$;

$S_d$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; $c_1$; $c_2$;

$S_e$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; $a_1$; $a_2$;

$S_c$ no es recuperable porque $T_2$ lee el elemento $X$ de $T_1$, y después se confirma $T_2$ antes que $T_1$. Si $T_1$ se cancela después de la operación $c_2$ en $S_c$, entonces el valor de $X$ que $T_2$ lee ya no es válido y $T_2$ debe cancelarse *después* de ser confirmada, lo que lleva a una planificación que no es recuperable. Para que la planificación sea recuperable, la operación $c_2$ en $S_c$ debe posponerse hasta después de confirmarse $T_1$, como se muestra en $S_d$; si $T_1$ se cancela en lugar de confirmarse, entonces $T_2$ también debe cancelarse como se muestra en $S_e$, porque el valor de $X$ que leyó ya no es válido.

En una planificación recuperable, ninguna transacción confirmada debe ser anulada. Sin embargo, es posible que se produzca un fenómeno conocido como **anulación en cascada**, en el que una transacción no *confirmada* tiene que ser anulada porque lee un elemento de una transacción que falló. Es lo que se ilustra en la planificación $S_e$, donde la transacción $T_2$ tiene que ser anulada porque lee el elemento $X$ de $T_1$, y $T_1$ se canceló entonces.

Como la anulación en cascada puede consumir mucho tiempo (puesto que puede darse sobre muchas transacciones [consulte el Capítulo 19]), es importante distinguir las transacciones en las que está garantizado que este fenómeno no ocurra. Una transacción es ***cascadeless***, o se dice que **evita la anulación en cascada**, si cada transacción de la planificación sólo lee elementos escritos por las transacciones confirmadas. En este caso, no se descartarán todos los elementos leídos, para que no se produzca una anulación en cascada. Para satisfacer este criterio, el comando $r_2(X)$ de las planificaciones $S_d$ y $S_e$ debe posponerse hasta después de haberse confirmado (o cancelado) $T_1$; en consecuencia, $T_2$ se retrasa pero se garantiza la ausencia de la anulación en cascada si $T_1$ se cancela.

Por último, hay un tercer tipo, más restrictivo, de planificación, denominado **planificación estricta**, en la que las transacciones *no pueden leer ni escribir* un elemento $X$ hasta haberse confirmado (o cancelado) la última transacción que escribió $X$. Las planificaciones estrictas simplifican el proceso de recuperación. En una planificación estricta, el proceso de deshacer una operación write_item($X$) de una transacción cancelada es simplemente recuperar la **imagen "antes"** (*valor_antiguo* o BFIM) del elemento de datos $X$. Este sencillo procedimiento siempre funciona correctamente para las planificaciones estrictas, pero puede que no funcione para las planificaciones recuperables o *cascadeless*. Por ejemplo, considere la planificación $S_f$:

$S_f$ : $w_1(X, 5)$; $w_2(X, 8)$; $a_1$;

Suponga que el valor de $X$ era originalmente 9, que es la imagen "antes" almacenada en el registro del sistema junto con la operación $w_1(X, 5)$. Si $T_1$ se cancela, como en $S_f$, el procedimiento de recuperación que almacena la imagen "antes" de una operación de escritura cancelada restaurará el valor de $X$ a 9, aunque la transacción $T_2$ ya lo cambió por 8, lo que nos lleva a unos resultados potencialmente incorrectos. Aunque la planificación $S_f$ evita la anulación en cascada, no es una planificación estricta, puesto que permite a $T_2$ escribir el elemento $X$ aunque la transacción $T_1$ que escribió $X$ en último lugar todavía no se haya confirmado (o cancelado). Una planificación estricta no tiene este problema.

Hemos clasificado las planificaciones de acuerdo con los siguientes términos: (1) recuperabilidad, (2) la posibilidad de evitar la anulación en cascada, y (3) la rigurosidad. Después vimos que estas propiedades de las planificaciones son condiciones sucesivamente más estrictas. Por tanto, la condición (2) implica la condición (1), y la condición (3) implica las condiciones (2) y (1). Así, todas las planificaciones estrictas evitan la anulación en cascada, y todas las planificaciones que evitan la anulación en cascada son recuperables.

# 17.5 Clasificación de las planificaciones basándose en la serialización

En la sección anterior clasificamos las planificaciones en base a sus propiedades de recuperabilidad. Ahora, clasificamos los tipos de planificaciones que se consideran correctos cuando varias transacciones se están ejecutando simultáneamente. Suponga que dos usuarios (agentes de viajes reservando en una aerolínea) remiten al DBMS, aproximadamente al mismo tiempo, las transacciones $T_1$ y $T_2$ de la Figura 17.2. Si no está permitida la interpolación de operaciones, sólo hay dos posibles resultados:

1. Ejecutar todas las operaciones de la transacción $T_1$ (en secuencia) seguidas por todas las operaciones de la transacción $T_2$ (en secuencia).
2. Ejecutar todas las operaciones de la transacción $T_2$ (en secuencia) seguidas por todas las operaciones de la transacción $T_1$ (en secuencia).

Estas alternativas se muestran en la Figura 17.5(a) y (b), respectivamente. Si está permitida la interpolación de operaciones, el sistema puede ejecutar las operaciones individuales de las transacciones obedeciendo a diversas ordenaciones. En la Figura 17.5(c) se muestran dos planificaciones posibles. El concepto de **serialización de planificaciones** se utiliza para identificar las planificaciones que son correctas cuando las ejecuciones de transacción tienen interpoladas sus operaciones en las planificaciones. Esta sección define la serialización y explica cómo puede utilizarse en la práctica.

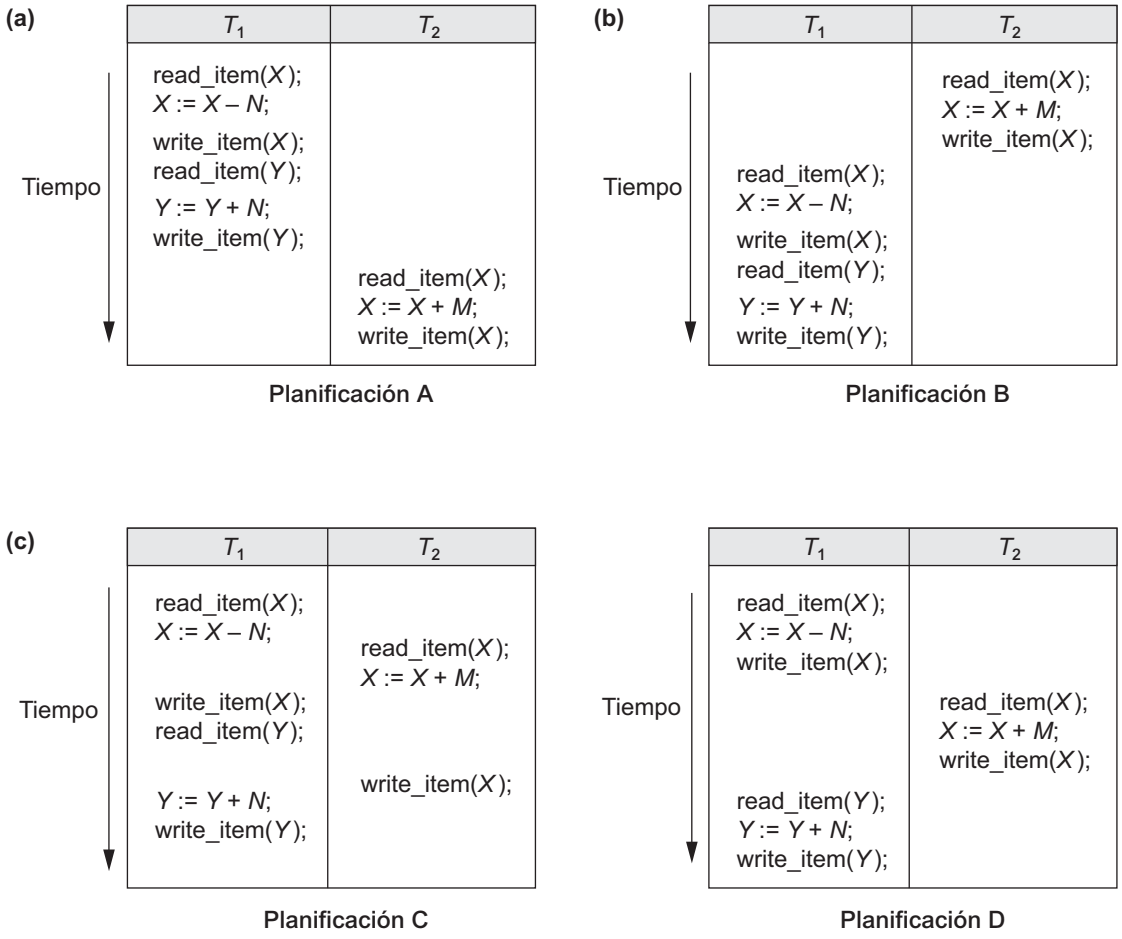
## 17.5.1 Planificaciones en serie, no serie y serializables por conflicto

Las planificaciones A y B de la Figura 17.5(a) y (b) están en *serie* porque las operaciones de cada transacción se ejecutan consecutivamente, sin que se interpolen las operaciones de otra transacción. En una planificación en serie, las transacciones se ejecutan enteras y en serie: $T_1$ y después $T_2$ en la Figura 17.5(a), y $T_2$ y después $T_1$ en la Figura 17.5(b). Las planificaciones C y D de la Figura 17.5(c) se denominan *no serie* porque cada secuencia interpola las operaciones de las dos transacciones.

Formalmente, una planificación $S$ es **serie** si, por cada transacción $T$ que participa en la planificación, todas las operaciones de $T$ se ejecutan consecutivamente en la planificación; en caso contrario, se dice que la planificación **no es serie**. Por consiguiente, en una planificación en serie, sólo hay una transacción activa al mismo tiempo (la confirmación, o la cancelación, de la transacción activa inicia la ejecución de la siguiente transacción). En una planificación en serie no se produce la interpolación. Una suposición razonable que podemos hacer, si consideramos que las transacciones son *independientes*, es considerar que cada planificación en serie es correcta. Podemos asumir esto porque se asume que una transacción es correcta si se ejecuta sola (según la propiedad de la conservación de la consistencia de la Sección 17.3). Por tanto, no importa qué transacción se ejecuta primero. Siempre y cuando cada transacción se ejecute desde el principio hasta el final sin que interfieran las operaciones de otras transacciones, conseguiremos un resultado correcto en la base de datos. El problema con las planificaciones serie es que limitan la concurrencia o la interpolación de operaciones. En una planificación en serie, si una transacción está esperando a que una operación de E/S se complete, no podemos hacer que el procesador de la CPU cambie a otra transacción, derrochándose de este modo un valioso tiempo de procesamiento de CPU. Además, si alguna transacción $T$ es bastante larga, las otras transacciones deben esperar a que $T$ complete todas sus operaciones antes de comenzar. Por tanto, y en la práctica, las planificaciones serie están generalmente consideradas inaceptables.

Para ilustrar nuestra explicación, considere las planificaciones de la Figura 17.5, y asuma que los valores iniciales de los elementos de la base de datos son $X$=90 e $Y$=90, y que $N$=3 y $M$=2. Después de ejecutar las transacciones $T_1$ y $T_2$, sería de esperar que los valores de la base de datos fueran $X$=89 e $Y$=93, de acuerdo con el significado de las transacciones. Sin duda, la ejecución de cualquiera de las planificaciones A o B ofrece los

**Figura 17.5.** Ejemplos de planificaciones serie y no serie en las que se ven implicadas las transacciones $T_1$ y $T_2$. (a) Planificación en serie A: $T_1$ seguida por $T_2$. (b) Planificación en serie B: $T_2$ seguida por $T_1$. (c) Dos planificaciones no serie C y D con interpolación de operaciones.

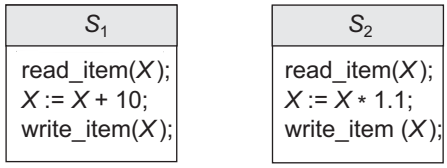



resultados correctos. Ahora, considere las planificaciones no serie C y D. La planificación C (que es la misma que la de la Figura 17.3[a]) ofrece los resultados $X$=92 e $Y$=93, siendo el valor de $X$ erróneo, mientras que la planificación D ofrece los resultados correctos.

La planificación C ofrece un resultado incorrecto debido al problema de la pérdida de actualización explicado en la Sección 17.1.3; la transacción $T_2$ lee el valor de $X$ antes de que sea cambiado por la transacción $T_1$, de modo que en la base de datos sólo se refleja el efecto de $T_2$ sobre $X$. El efecto de $T_1$ sobre $X$ se *pierde*, al ser sobrescrito por $T_2$, lo que nos lleva a un resultado incorrecto para el elemento $X$. Sin embargo, algunas planificaciones no serie ofrecen el resultado correcto, como la planificación $D$. Nos gustaría determinar cuáles de las planificaciones no serie *siempre* ofrecen un resultado correcto y cuáles pueden proporcionar resultados erróneos. El concepto utilizado para clasificar las planificaciones de este modo es la serialización de una planificación.

Una planificación $S$ de $n$ transacciones es **serializable** si es *equivalente a alguna planificación en serie* de las mismas $n$ transacciones. Definiremos brevemente el concepto de equivalencia de planificaciones. A partir de $n$ transacciones, podemos obtener $n!$ planificaciones en serie posibles y muchas más planificaciones no serie. Podemos formar dos grupos disjuntos con las planificaciones no serie: aquellas que son equivalentes a una (o

**Figura 17.6.** Dos planificaciones equivalentes en cuanto a resultado para el valor inicial de $X = 100$, pero que en general no son de resultado equivalente.




más) de las planificaciones en serie y, por tanto, serializables; y aquellas que no son equivalentes a ninguna planificación en serie y, por tanto, no son serializables.

Decir que una planificación no serie $S$ es serializable es equivalente a decir que es correcta, porque es equivalente a una planificación en serie, que se considera correcta. La cuestión es: ¿Cuándo se considera que dos planificaciones son *equivalentes*? Hay varias formas de definir la equivalencia de planificación. La definición más sencilla, pero menos satisfactoria, implica comparar los efectos de las planificaciones sobre la base de datos. Dos planificaciones son de **resultado equivalente** si producen el mismo estado final de la base de datos. Sin embargo, dos planificaciones diferentes pueden producir accidentalmente el mismo estado final. Por ejemplo, en la Figura 17.6, las planificaciones $S_1$ y $S_2$ producirán el mismo estado final de la base de datos si se ejecutan en una base de datos con un valor inicial de $X$=100; pero para otros valores iniciales de $X$, las planificaciones no son de resultado equivalente. Además, estas planificaciones ejecutan transacciones diferentes, por lo que no deben considerarse definitivamente equivalentes. Por tanto, la equivalencia de resultado por sí sola no puede utilizarse para definir la equivalencia de planificaciones. La metodología más segura y general para definir la equivalencia de planificaciones es no hacer ninguna asunción sobre los tipos de operaciones incluidos en las transacciones. Para que dos planificaciones sean equivalentes, las operaciones aplicadas a cada elemento de datos afectado por las planificaciones deben aplicarse a ese elemento en ambas planificaciones *en el mismo orden*. Generalmente se utilizan dos definiciones de equivalencia de planificaciones: *equivalencia por conflicto* y *equivalencia por vista*. A continuación explicamos la equivalencia por conflicto, que es la definición más utilizada.

Dos planificaciones son **equivalentes por conflicto** si el orden de cualquier par de *operaciones en conflicto* es el mismo en las dos planificaciones. Como recordará de la Sección 17.4.1, dos operaciones de una planificación entran en conflicto si pertenecen a transacciones diferentes, acceden al mismo elemento de la base de datos, y al menos una de las dos operaciones es una operación write_item. Si dos operaciones conflictivas se aplican con un orden diferente en dos planificaciones, el efecto puede ser diferente en la base de datos o en otras transacciones de la planificación y, por tanto, las planificaciones no son equivalentes por conflicto. Por ejemplo, si en la planificación $S_1$ se produce una operación de lectura y escritura en el orden $r_1(X)$, $w_2(X)$, y en el orden inverso, $w_2(X)$, $r_1(X)$, en la planificación $S_2$, el valor leído por $r_1(X)$ puede ser diferente en las dos planificaciones. De forma parecida, si dos operaciones de escritura tienen lugar en el orden $w_1(X)$, $w_2(X)$ en $S_1$, y en el orden inverso, $w_2(X)$, $w_1(X)$, en $S_2$, es probable que la siguiente operación $r(X)$ en las dos planificaciones lean valores diferentes; o si son las últimas operaciones que escriben el elemento $X$ en las planificaciones, el valor final del elemento $X$ en la base de datos será diferente.

Utilizando el concepto de equivalencia por conflicto, decimos que una planificación $S$ es **serializable por conflicto**[11] si es equivalente (por conflicto) con alguna planificación en serie $S'$. En tal caso, podemos reordenar las operaciones *no conflictivas* de $S$ para formar la planificación serie equivalente $S'$. De acuerdo con esta definición, la planificación $D$ de la Figura 17.5(c) es equivalente a la planificación en serie $A$ de la Figura 17.5(a). En las dos planificaciones, la operación read_item($X$) de $T_2$ lee el valor de $X$ escrito por $T_1$, mientras

[11] Utilizaremos *serializable* para dar a entender serializable por conflicto. Otra definición de serializable que se utiliza en la práctica (consulte la Sección 17.6) es tener lecturas repetitivas, no lecturas sucias, y sin registros fantasma (consulte la Sección 18.7.1 si desea una explicación de fantasma).

que las otras operaciones read_item leen los valores correspondientes al estado inicial de la base de datos. Además, $T_1$ es la última transacción que escribe $Y$, y $T_2$ es la última transacción que escribe $X$ en ambas planificaciones. Como A es una planificación en serie y la planificación D es equivalente a A, D es una planificación serializable. Las operaciones $r_1(Y)$ y $w_1(Y)$ de la planificación D no entran en conflicto con las operaciones $r_2(X)$ y $w_2(X)$, puesto que acceden a elementos de datos diferentes. Por consiguiente, podemos mover $r_1(Y)$, $w_1(Y)$ antes de $r_2(X)$, $w_2(X)$, lo que nos lleva a la planificación serie equivalente de $T_1$, $T_2$.

La planificación C de la Figura 17.5(c) no es equivalente a ninguna de las dos posibles planificaciones en serie, A y B, y, por tanto, no es *serializable*. El intento de reordenar las operaciones de la planificación C para encontrar una planificación en serie equivalente falla porque $r_2(X)$ y $w_1(X)$ entran en conflicto, lo que significa que no podemos mover $r_2(X)$ hacia abajo para obtener la planificación en serie equivalente de $T_1$, $T_2$. De forma parecida, como $w_1(X)$ y $w_2(X)$ entran en conflicto, no podemos mover $w_1(X)$ hacia abajo para obtener la planificación serie equivalente de $T_2$, $T_1$.

En la Sección 17.5.4 explicamos otra definición más compleja de equivalencia (denominada equivalencia por vista, que conduce al concepto de serialización por vista).

### 17.5.2 Comprobación de la serialización por conflicto de una planificación

Hay un algoritmo sencillo para determinar la serialización por conflicto de una planificación. La mayoría de los métodos de control de la concurrencia *no* comprueban realmente la serialización. Más bien, se desarrollan protocolos, o reglas, que garantizan que una planificación será serializable. Aquí explicamos el algoritmo de comprobación de la serialización por conflicto de las planificaciones para entender mejor esos protocolos de control de la concurrencia, que explicamos en el Capítulo 18.

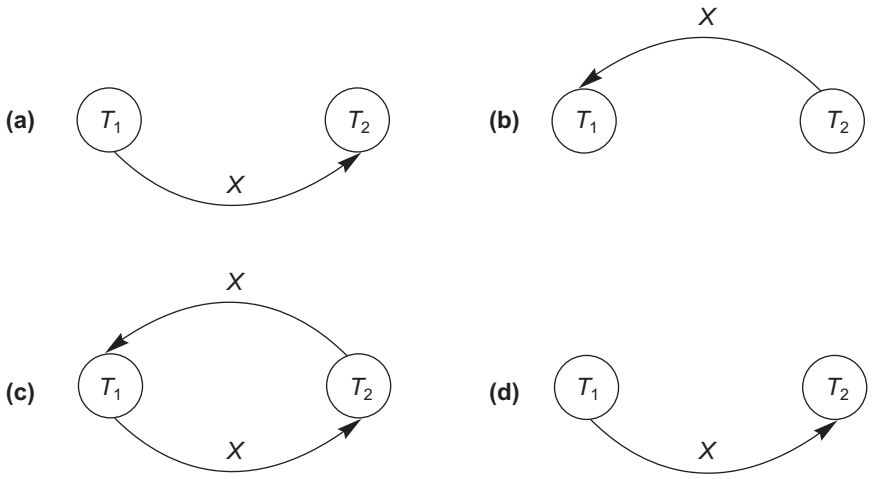
El Algoritmo 17.1 puede utilizarse para comprobar la serialización por conflicto de una planificación. El algoritmo sólo examina las operaciones read_item y write_item de una planificación para construir un **gráfico de precedencia** (o **gráfico de serialización**), que es un **gráfico tendente** $G = (N, E)$ consistente en un conjunto de nodos $N = \{T_1, T_2, ..., T_n\}$ y un conjunto de bordes o arcos tendentes $E = \{e_1, e_2, ..., e_m\}$. En el gráfico hay un nodo por cada transacción $T_i$ de la planificación. Cada arco $e_i$ del gráfico tiene la forma $(T_j \rightarrow T_k)$, 1 ? $j$ = ?, 1 ? $k = n$, donde $T_j$ es el **nodo inicial** de $e_i$ y $T_k$ es el **nodo final** de $e_i$. Se crea un arco de estas características si una de las operaciones de $T_j$ aparece en la planificación antes que alguna *operación en conflicto* de $T_k$.

**Algoritmo 17.1. Comprobación de la serialización por conflicto de una planificación *S*.**

1. Por cada transacción $T_i$ participante en la planificación $S$, crear un nodo etiquetado como $T_i$ en el gráfico de precedencia.
2. Por cada caso de $S$ donde $T_j$ ejecute una operación read_item($X$) después de que $T_i$ ejecute write_item($X$), crear un arco $(T_i \rightarrow T_j)$ en el gráfico de precedencia.
3. Por cada caso de S donde $T_j$ ejecute una operación write_item($X$) después de que $T_i$ ejecute read_item($X$), crear un arco $(T_i \rightarrow T_j)$ en el gráfico de precedencia.
4. Por cada caso de S donde $T_j$ ejecute una operación write_item($X$) después de que $T_i$ ejecute write_item($X$), crear un arco $(T_i \rightarrow T_j)$ en el gráfico de precedencia.
5. La planificación $S$ es serializable si, y sólo si, el gráfico de precedencia no tiene ciclos.

El gráfico de precedencia se construye como se describe en el Algoritmo 17.1. Si hay un ciclo en el gráfico de precedencia, la planificación $S$ no es serializable (por conflicto); si no hay ningún ciclo, $S$ es serializable. Un **ciclo** en un gráfico tendente es una **secuencia de arcos** $C = ((T_j \rightarrow T_k), (T_k \rightarrow T_p), ..., (T_i \rightarrow T_j))$ con la propiedad de que el nodo inicial de cada arco (excepto en el primer arco) es el mismo que el nodo final del

**Figura 17.7.** Construcción de los gráficos de precedencia para las planificaciones A a D de la Figura 17.5 para probar la serialización por conflicto. (a) Gráfico de precedencia para la planificación en serie A. (b) Gráfico de precedencia para la planificación en serie B. (c) Gráfico de precedencia para la planificación C (no serializable). (d) Gráfico de precedencia para la planificación D (serializable, equivalente a la planificación A).




arco anterior, y el nodo inicial del primer arco es el mismo que el nodo final del último arco (la secuencia empieza y termina en el mismo nodo).

En el gráfico de precedencia, un arco de $T_i$ a $T_j$ significa que la transacción $T_i$ debe ir antes que la transacción $T_j$ en cualquier planificación en serie que sea equivalente a $S$, porque aparecen dos operaciones en conflicto en ese orden en la planificación. Si no hay ningún ciclo en el gráfico de precedencia, podemos crear una **planificación en serie equivalente** $S'$ que sea equivalente a $S$, ordenando de este modo las transacciones que participan en $S$: siempre que exista un arco en el gráfico de precedencia de $T_i$ a $T_j$, $T_i$ debe aparecer antes que $T_j$ en la planificación en serie equivalente $S'$.[12] Los arcos ($T_i \rightarrow T_j$) en un gráfico de precedencia pueden etiquetarse opcionalmente con el(los) nombre(s) del(de los) elemento(s) que llevan a crear el arco. La Figura 17.7 muestra dichas etiquetas en los arcos.

En general, varias planificaciones en serie pueden ser equivalentes a $S$ si el gráfico de precedencia para $S$ no tiene ciclos. Sin embargo, si el gráfico de precedencia tiene un ciclo, es fácil mostrar que no podemos crear ninguna planificación en serie equivalente, por lo que $S$ no es serializable. Los gráficos de precedencia creados para las planificaciones A a D, respectivamente, de la Figura 17.5 aparecen en la Figura 17.7(a) a (d). El gráfico para la planificación C tiene un ciclo, por lo que no es serializable. El gráfico para la planificación D no tiene ciclos, por lo que es serializable, y la planificación serie equivalente es $T_1$ seguida de $T_2$. Los gráficos para las planificaciones A y B no tienen ciclo, como era de esperar, porque las planificaciones son serie y, por tanto, serializables.

La Figura 17.8 muestra otro ejemplo, en el que participan tres transacciones. La Figura 17.8(a) muestra las operaciones read_item y write_item de cada transacción. En la Figura 17.8(b) y (c) se muestran, respectivamente, las dos planificaciones, $E$ y $F$, para esas transacciones, mientras que en las partes (d) y (e) se muestran los gráficos de precedencia para dichas planificaciones. La planificación $E$ no es serializable porque el gráfico de precedencia correspondiente tiene ciclos. La planificación $F$ es serializable; su planificación en serie

[12] Este proceso de ordenación de los nodos de un gráfico se conoce como ordenación topológica.

**Figura 17.8.** Otro ejemplo de comprobación de la serialización. (a) Operaciones de lectura y escritura de las tres transacciones, $T_1$, $T_2$ y $T_3$. (b) Planificación *E*. (c) Planificación *F*. (d) Gráfica de precedencia para la planificación *E*.

**(a)**

| Transacción $T_1$ |
|---|
| read_item($X$); |
| write_item($X$); |
| read_item($Y$); |
| write_item($Y$); |

| Transacción $T_2$ |
|---|
| read_item($Z$); |
| read_item($Y$); |
| write_item($Y$); |
| read_item($X$); |
| write_item($X$); |

| Transacción $T_3$ |
|---|
| read_item($Y$); |
| read_item($Z$); |
| write_item($Y$); |
| write_item($Z$); |

**(b)**

Tiempo

| Transacción $T_1$ | Transacción $T_2$ | Transacción $T_3$ |
|---|---|---|
| | read_item($Z$); | |
| | read_item($Y$); | |
| | write_item($Y$); | |
| | | read_item($Y$); |
| read_item($X$); | | read_item($Z$); |
| write_item($X$); | | |
| | | write_item($Y$); |
| | | write_item($Z$); |
| | read_item($X$); | |
| read_item($Y$); | | |
| write_item($Y$); | write_item($X$); | |

**Planificación E**

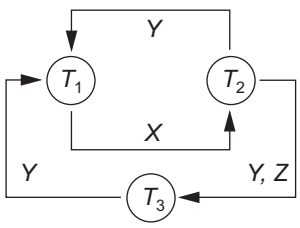
**(c)**

Tiempo

| Transacción $T_1$ | Transacción $T_2$ | Transacción $T_3$ |
|---|---|---|
| | | read_item($Y$); |
| | | read_item($Z$); |
| read_item($X$); | | |
| write_item($X$); | | |
| | | write_item($Y$); |
| | | write_item($Z$); |
| | read_item($Z$); | |
| read_item($Y$); | | |
| write_item($Y$); | read_item($Y$); | |
| | write_item($Y$); | |
| | read_item($X$); | |
| | write_item($X$); | |

**Planificación F**

**(d)**



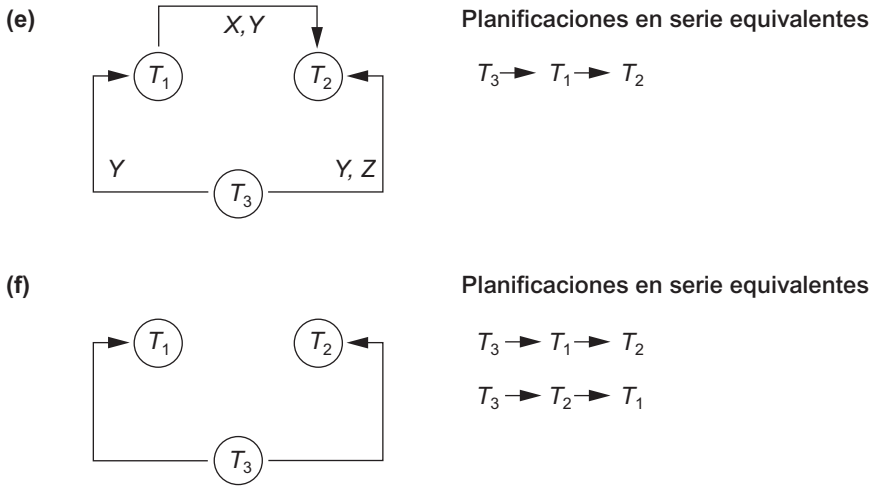

**Planificaciones en serie equivalentes**

Ninguna

**Razón**

*Ciclo* $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$
*Ciclo* $X(T_1 \rightarrow T_2), YZ\ (T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

**Figura 17.8.** Otro ejemplo de comprobación de la serialización. (e) Gráfica de precedencia para la planificación *F*. (f) Gráfica de precedencia con dos planificaciones serie equivalentes.




equivalente se muestra en la Figura 17.8(e). Aunque para *F* sólo existe una planificación en serie equivalente, en general puede haber más de una planificación en serie equivalente para una planificación serializable. La Figura 17.8(f) muestra un gráfico de precedencia que representa una planificación que tiene dos planificaciones en serie equivalentes.

## 17.5.3 Usos de la serialización

Como explicamos anteriormente, decir que una planificación *S* es serializable (por conflicto) (es decir, *S* es equivalente [por conflico] a una planificación en serie), es como decir que *S* es correcta. Sin embargo, ser *serializable* es distinto a estar *en serie*. Una planificación en serie representa un procesamiento ineficaz, porque no está permitida la interpolación de las operaciones de transacciones diferentes. Esto puede llevar a una baja utilización de la CPU mientras una transacción espera por una E/S de disco, o porque se está esperando a que otra transacción termine; esto ralentiza considerablemente el procesamiento. Una planificación serializable proporciona los beneficios de la ejecución concurrente sin ninguna corrección. En la práctica, es muy difícil probar la serialización de una planificación. Por regla general, la interpolación de operaciones de las transacciones concurrentes (que normalmente las ejecuta el sistema operativo como procesos) es determinada por el planificador del sistema operativo, que asigna recursos a todos los procesos. Factores como la carga del sistema, el tiempo de envío de una transacción y las prioridades de los procesos contribuyen a la ordenación de las operaciones en una planificación. Por tanto, es difícil determinar de antemano cómo se interpolarán las operaciones de una planificación para garantizar la serialización.

Si las transacciones se ejecutan a voluntad y, después, se comprueba la serialización de la planificación resultante, debemos cancelar el efecto de la planificación si resulta no ser serializable. Esto es un serio problema, por lo que esta metodología no es práctica. Por tanto, la metodología que se toma en la mayoría de los sistemas prácticos es determinar los métodos que garantizan la serialización, sin tener que probar las propias planificaciones. La metodología tomada en la mayoría de los DBMSs comerciales es diseñar **protocolos** (conjuntos de reglas) que (seguidos por *cada* transacción individual o implementados por un subsistema de control de la concurrencia de un DBMS) garantizarán la serialización de *todas las planificaciones en las que las transacciones participan*.

Aquí aparece otro problema: cuando se envían transacciones continuamente al sistema, es difícil determinar cuándo empieza una planificación y cuándo termina. La teoría de la serialización puede adaptarse para tratar con este problema considerando únicamente la proyección confirmada de una planificación *S*. Como recordará de la Sección 17.4.1, la *proyección confirmada* $C(S)$ de una planificación $S$ sólo incluye las operaciones de $S$ que pertenecen a las transacciones confirmadas. Podemos definir teóricamente una planificación $S$ como serializable si su proyección confirmada $C(S)$ es equivalente a alguna planificación en serie, puesto que el DBMS sólo garantiza las transacciones confirmadas.

En el Capítulo 18 veremos diferentes protocolos de control de la concurrencia que garantizan la serialización. La técnica más común, denominada *bloqueo de dos fases*, está basada en el bloqueo de los elementos de datos para evitar que las transacciones concurrentes interfieran entre sí, y la implementación de una condición adicional que garantice la serialización. Es la técnica que se utiliza en la mayoría de los DBMSs comerciales. También se han propuesto otros protocolos;[13] *ordenación por marca de tiempo*, con la que a cada transacción se le asigna una marca de tiempo única y el protocolo garantiza que cualquier operación conflictiva se ejecuta en el orden de las marcas de tiempo de las transacciones; *protocolos multiversión*, que están basados en el mantenimiento de varias versiones de elementos de datos; y *protocolos optimistas* (también denominados *certificación* o *validación*), que comprueban las posibles violaciones de la serialización una vez terminadas las transacciones, pero antes de que esté permitida su confirmación.

### 17.5.4 Equivalencia por vista y serialización por vista

En la Sección 17.5.1 definimos los conceptos de equivalencia por conflicto de las planificaciones y la serialización por conflicto. Otra definición menos restrictiva de equivalencia de las planificaciones se conoce como *equivalencia por vista*. Esto nos lleva a otra definición de serialización denominada *serialización por vista*. Se dice que dos planificaciones $S$ y $S'$ son **equivalentes en vista** si se dan estas tres condiciones:

1. El mismo conjunto de transacciones participa en $S$ y $S'$, y $S$ y $S'$ incluyen las mismas operaciones de esas transacciones.
2. Para cualquier operación $r_i(X)$ de $T_i$ en $S$, si el valor de $X$ leído por la operación ha sido escrito por una operación $w_j(X)$ de $T_j$ (o si es el valor original de $X$ antes de iniciarse la planificación), la misma condición debe mantenerse para el valor de $X$ leído por la operación $r_i(X)$ de $T_i$ en $S'$.
3. Si la operación $w_k(Y)$ de $T_k$ es la última operación en escribir el elemento $Y$ en $S$, entonces $w_k(Y)$ de $T_k$ también debe ser la última operación para escribir el elemento $Y$ en $S'$.

La idea tras la equivalencia por vista es que, siempre y cuando cada operación de lectura de una transacción lea el resultado de la misma operación de escritura en las dos transacciones, las operaciones de escritura de cada transacción deben producir los mismos resultados. Se dice, por tanto, que las operaciones de lectura *ven la misma vista* en las dos planificaciones. La condición 3 garantiza que la operación de escritura final en cada elemento de datos es la misma en las dos planificaciones, por lo que el estado de la base de datos debe ser el mismo al final de ambas planificaciones. Una planificación $S$ se dice que es **serializable por vista** si es la vista equivalente a una planificación en serie.

Las definiciones de serialización por conflicto y serialización por vista son similares si se cumple una condición conocida como **suposición de escritura restringida** en todas las transacciones de la planificación. Esta condición establece que cualquier operación de escritura $w_i(X)$ en $T_i$ va precedida por una operación $r_i(X)$ en $T_i$ y que el valor escrito por $w_i(X)$ en $T_i$ sólo depende del valor de $X$ leído por $r_i(X)$. Esto supone que el cálculo del valor nuevo de $X$ es una función $f(X)$ basada en el valor antiguo de $X$ leído de la base de datos. No obstante, la definición de serialización por vista es menos restrictiva que la de serialización por conflicto bajo

[13] Estos otros protocolos no se han utilizado mucho en la práctica hasta ahora; la mayoría de los sistemas utilizan alguna variante del protocolo de bloqueo de dos fases.

la **suposición de escritura restringida**, donde el valor escrito por una operación $w_i(X)$ en $T_i$ puede ser independiente de su valor antiguo de la base de datos. Es lo que se conoce como **escritura ciega**, y se ilustra con la siguiente planificación $S_g$ de tres transacciones, $T_1$: $r_1(X)$; $w_1(X)$; $T_2$: $w_2(X)$; y $T_3$: $w_3(X)$:

$$S_g: r_1(X); w_2(X); w_1(X); w_3(X); c_1; c_2; c_3;$$

En $S_g$ las operaciones $w_2(X)$ y $w_3(X)$ son escrituras ciegas, porque $T_2$ y $T_3$ no leen el valor de $X$. La planificación $S_g$ es serializable por vista, puesto que es el equivalente por vista de la planificación en serie $T_1$, $T_2$, $T_3$. Sin embargo, $S_g$ no es serializable por conflicto, puesto que no es equivalente por conflicto a ninguna planificación en serie. Se ha mostrado que cualquier planificación serializable por conflicto también es serializable por vista, pero no a la inversa, como mostraba el ejercicio anterior. Hay un algoritmo para probar si una planificación $S$ es o no serializable por vista. Sin embargo, el problema de probar la serialización por vista es difícil, lo que significa que encontrar un algoritmo de tiempo polinomial eficaz para este problema es altamente improbable.

### 17.5.5 Otros tipos de equivalencia de planificaciones

La serialización de las planificaciones es a veces considerada demasiado restrictiva como condición para garantizar la exactitud de las ejecuciones concurrentes. Algunas aplicaciones pueden producir planificaciones que son correctas satisfaciendo las condiciones menos severas que la serialización por conflicto o la serialización por vista. Un ejemplo es el tipo de transacciones conocidas como **transacciones de débito-crédito** (por ejemplo, las que aplican depósitos y retiradas de fondos a un elemento de datos cuyo valor es el saldo actual de una cuenta bancaria). La semántica de las operaciones de débito-crédito es que actualizan el valor de un elemento de datos $X$ sustrayendo o añadiendo al valor de ese elemento de datos. Como las operaciones de adición y sustracción son conmutativas (es decir, pueden aplicarse en cualquier orden), es posible generar planificaciones correctas que no sean serializables. Por ejemplo, considere las siguientes transacciones, cada una de las cuales puede utilizarse para transferir una cantidad de dinero entre dos cuentas bancarias:

$$T_1: r_1(X); X := X - 10; w_1(X); r_1(Y); Y := Y + 10; w_1(Y);$$
$$T_2: r_2(Y); Y := Y - 20; w_2(Y); r_2(X); X := X + 20; w_2(X);$$

Considere la siguiente planificación no serializable, $S_h$, para las dos transacciones:

$$S_h: r_1(X); w_1(X); r_2(Y); w_2(Y); r_1(Y); w_1(Y); r_2(X); w_2(X);$$

Con el conocimiento, o **semántica**, adicional de que las operaciones entre cada $r_i(I)$ y $w_i(I)$ son conmutativas, sabemos que el orden de ejecución de las secuencias consistentes en (lectura, actualización, escritura) no es importante siempre y cuando cada secuencia (lectura, actualización, escritura) por una transacción particular $T_i$ en un elemento $I$ concreto no sea interrumpida por operaciones en conflicto. Por tanto, la planificación $S_h$ es considerada correcta aunque no sea serializable. Los investigadores han estado trabajando para extender la teoría del control de la concurrencia al tratamiento de casos en los que se considera que la serialización es demasiado restrictiva como condición para medir la exactitud de las planificaciones.

## 17.6 Soporte de transacciones en SQL

La definición de transacción SQL es parecida a nuestro concepto de transacción ya definido. Es decir, es una unidad lógica de trabajo cuya atomicidad está garantizada. Una simple sentencia SQL se considera que siempre es atómica (tanto si completa la ejecución sin errores, como si falla y deja la base de datos sin cambios).

Con SQL, no hay ninguna sentencia de inicio de transacción explícita. El inicio de una transacción se hace implícitamente cuando se encuentran sentencias SQL particulares. Sin embargo, cada transacción debe tener una sentencia explícita de terminación, que puede ser COMMIT o ROLLBACK. Una transacción tiene ciertas

características atribuibles a ella, que se especifican en SQL con una sentencia SET TRANSACTION. Las características son el *modo de acceso*, el *tamaño del área de diagnóstico* y el *nivel de aislamiento*.

El **modo de acceso** puede especificarse como READ ONLY o READ WRITE. Lo predeterminado es READ WRITE, a menos que especifiquemos el nivel de aislamiento READ UNCOMMITTED, en cuyo caso se asume READ ONLY. El modo READ WRITE permite seleccionar, actualizar, insertar, eliminar y crear comandos para su ejecución. El modo READ ONLY, como su nombre indica, es simplemente para la recuperación de datos.

La opción del **tamaño del área de diagnóstico**, DIAGNOSTIC SIZE *n*, especifica un valor entero *n*, que indica el número de condiciones que se pueden mantener simultáneamente en el área de diagnóstico. Estas condiciones proporcionan retroalimentación (errores o excepciones) al usuario o al programa en la sentencia SQL recién ejecutada.

La opción de **nivel de aislamiento** se especifica con la sentencia ISOLATION LEVEL <aislamiento>, donde el valor para <aislamiento> puede ser READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ o SERIALIZABLE.[14] El nivel de aislamiento predeterminado es SERIALIZABLE, aunque algunos sistemas utilizan READ COMMITTED como su valor predeterminado. El uso del término SERIALIZABLE aquí está basado en no permitir violaciones que provoquen lecturas sucias, irrepetibles y fantasmas,[15] y, por tanto, no es idéntico a como se definió la serialización anteriormente en la Sección 17.5. Si una transacción se ejecuta a un nivel de aislamiento más bajo que SERIALIZABLE, entonces se pueden producir una o más de estas violaciones:

1. **Lectura sucia.** Una transacción $T_1$ puede leer la actualización de una transacción $T_2$, que todavía no se ha confirmado. Si $T_2$ falla y es cancelada, entonces $T_1$ habría leído un valor que no existe y es incorrecto.
2. **Lectura irrepetible.** Una transacción $T_1$ puede leer un valor dado de una tabla. Si otra transacción $T_2$ actualiza más tarde ese valor y $T_1$ lee de nuevo el valor, $T_1$ verá un valor diferente.
3. **Fantasmas.** Una transacción $T_1$ puede leer un conjunto de filas de una tabla, quizá basándose en alguna condición especificada en la cláusula WHERE de SQL. Ahora, suponga que una transacción $T_2$ inserta una fila nueva que también satisface la condición de la cláusula WHERE utilizada en $T_1$, en la tabla utilizada por $T_1$. Si $T_1$ se repite, entonces $T_1$ verá un fantasma, una fila que anteriormente no existía.

La Tabla 17.1 resume las posibles violaciones para los distintos niveles de aislamiento. Una entrada con la palabra "Sí" indica que es posible una violación, y una entrada con "No" indica que no es posible. READ UNCOMMITTED es la opción más clemente, mientras que SERIALIZABLE es la más restrictiva al evitar los tres problemas mencionados anteriormente.

**Tabla 17.1.** Posibles violaciones según el nivel de aislamiento.

| | Tipo de violación | | |
|---|---|---|---|
| **Nivel de aislamiento** | **Lectura sucia** | **Lectura irrepetible** | **Fantasma** |
| READ UNCOMMITTED | Sí | Sí | Sí |
| READ COMMITTED | No | Sí | Sí |
| REPEATABLE READ | No | No | Sí |
| SERIALIZABLE | No | No | No |

Una transacción SQL podría parecerse a lo siguiente:

[14] Son similares a los *niveles de aislamiento* que explicamos brevemente al final de la Sección 17.3.

[15] Los problemas de lectura sucia y de lectura irrepetible se explicaron en la Sección 17.1.3. Los fantasmas se explican en la Sección 18.6.1.

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
   READ WRITE
   DIAGNOSTIC SIZE 5
   ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT INTO EMPLEADO (Nombre, Apellidos, Dni, Dno, Sueldo)
   VALUES ('Luis', 'Campos', '991004321', 2, 35000);
EXEC SQL UPDATE EMPLEADO
   SET Sueldo = Sueldo * 1.1 WHERE Dno = 2;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ... ;
```

La transacción anterior primero inserta una nueva fila en la tabla EMPLEADO y, después, actualiza el sueldo de todos los empleados del departamento 2. Si se produce un error en cualquiera de las sentencias SQL, la transacción entera es anulada. Esto implica que los sueldos actualizados (por esta transacción) deberían restaurarse a su valor antiguo y que la fila recién insertada debería eliminarse.

Como hemos visto, SQL proporciona algunas características orientadas a las transacciones. El DBA o los programadores de la base de datos pueden beneficiarse de esas opciones para intentar mejorar el rendimiento de la transacción relajando la serialización, siempre y cuando sea aceptable para sus aplicaciones.

# 17.7 Resumen

En este capítulo hemos explicado los conceptos DBMS relacionados con el procesamiento de las transacciones. Hemos introducido el concepto de transacción, así como las operaciones relacionadas con el procesamiento de las mismas. Asimismo, hemos comparado los sistemas monousuario con los sistemas multiusuario, y hemos presentado algunos ejemplos de cómo una ejecución incontrolada de transacciones concurrentes en un sistema multiusuario puede llevar a resultados y valores incorrectos en la base de datos. También hemos explicado los distintos tipos de fallos que pueden surgir durante la ejecución de una transacción.

A continuación, vimos los estados típicos por los que pasa una transacción durante su ejecución, y explicamos diversos conceptos que se utilizan en los métodos de recuperación y control de la concurrencia. El registro del sistema hace un seguimiento de los accesos a la base de datos, y el sistema utiliza esa información para recuperarse de los fallos. Una transacción puede tener éxito y alcanzar su punto de confirmación, o puede fallar y tener que anularse. Una transacción confirmada implica que sus cambios se han grabado permanentemente en la base de datos. Ofrecimos una visión general de las propiedades deseables de las transacciones (atomicidad, conservación de la consistencia, aislamiento y durabilidad), que con frecuencia reciben el nombre de propiedades ACID (por sus iniciales en inglés).

Después definimos una planificación como una secuencia de ejecución de las operaciones de varias transacciones, con una posible interpolación. Clasificamos las planificaciones según su recuperabilidad. Las planificaciones recuperables garantizan que, una vez confirmada una transacción, nunca habrá necesidad de que haya que deshacerla. Las planificaciones que evitan la anulación en cascada añaden una condición adicional para garantizar que ninguna transacción cancelada requiera la cancelación en cascada de otras transacciones. Las planificaciones estrictas proporcionan una condición aún más fuerte que permite un esquema de recuperación sencillo consistente en restaurar los valores antiguos de los elementos modificados por una transacción cancelada.

Hemos definido la equivalencia de planificaciones y visto que una planificación serializable es equivalente a alguna planificación en serie. Hemos definido los conceptos de equivalencia por conflicto y de equivalencia por vista, que conducen a las definiciones de serialización por conflicto y serialización por vista. Una planificación serializable se considera correcta. Presentamos los algoritmos para probar la serialización (por conflicto) de una planificación. Explicamos por qué probar la serialización no es práctico en un sistema real, aun-

que puede utilizarse para definir y verificar los protocolos de control de la concurrencia, y mencionamos brevemente definiciones menos restrictivas de la equivalencia de planificaciones. Por último, ofrecimos una breve panorámica de cómo se utilizan los conceptos de transacción en la práctica dentro de SQL.

En el Capítulo 18 veremos los protocolos de control de la concurrencia, y en el Capítulo 19 los protocolos de recuperación.

## Preguntas de repaso

**17.1.** ¿Qué se entiende por ejecución concurrente de las transacciones de una base de datos en un sistema multiusuario? Explique por qué es necesario controlar la concurrencia, y ofrezca algunos ejemplos informales.

**17.2.** Explique los diferentes tipos de fallos. ¿A qué hace referencia un fallo catastrófico?

**17.3.** Explique las acciones llevadas a cabo por las operaciones read_item y write_item en una base de datos.

**17.4.** Dibuje un diagrama de estado y explique los estados típicos por los que pasa una transacción durante su ejecución.

**17.5.** ¿Para qué se utiliza el registro del sistema? ¿Cuáles son las clases típicas de entradas en un registro del sistema? ¿Qué son los puntos de confirmación de una transacción, y por qué son importantes?

**17.6.** Explique las propiedades de atomicidad, durabilidad, aislamiento y conservación de la consistencia de una transacción de base de datos.

**17.7.** ¿Qué es una planificación? Defina los conceptos de planificaciones recuperables, planificaciones que evitan la anulación en cascada y planificaciones estrictas, y compárelas según su recuperabilidad.

**17.8.** Explique las diferentes medidas de la equivalencia de transacciones. ¿Cuál es la diferencia entre equivalencia por conflicto y equivalencia por vista?

**17.9.** ¿Qué es una planificación en serie? ¿Qué es una planificación serializable? ¿Por qué se considera que una planificación en serie es correcta? ¿Por qué se considera que una planificación serializable es correcta?

**17.10.** ¿Cuál es la diferencia entre las suposiciones de escritura restringida y escritura no restringida? ¿Cuál es más realista?

**17.11.** Explique cómo se utiliza la serialización para implementar el control de la concurrencia en un sistema de base de datos. ¿Por qué a veces se considera que la serialización es demasiado restrictiva como medida de exactitud para las planificaciones?

**17.12.** Describa los cuatro niveles de aislamiento en SQL.

**17.13.** Defina las violaciones causadas por cada una de estas circunstancias: lectura sucia, lectura irrepetible y fantasmas.

## Ejercicios

**17.14.** Cambie la transacción $T_2$ de la Figura 17.2(b) para que se lea:

```
read_item(X);
X := X + M;
if X > 90 then exit
else write_item(X);
```

Explique el resultado final de las diferentes planificaciones de la Figura 17.3(a) y (b), donde $M = 2$ y $N = 2$, respecto a las siguientes cuestiones. ¿La adición de la condición anterior cambia

el resultado final? ¿El resultado obedece la regla de consistencia implícita (que la capacidad de $X$ es 90)?

**17.15.** Repita el Ejercicio 17.14, añadiendo una comprobación en $T_1$ para que $Y$ no exceda de 90.

**17.16.** Añada la operación commit al final de cada una de las transacciones, $T_1$ y $T_2$, de la Figura 17.2; después, liste todas las posibles planificaciones para las transacciones modificadas. Determine cuál de las planificaciones es recuperable, cuál evita la anulación en cascada, y cuál es estricta.

**17.17.** Liste todas las planificaciones posibles para las transacciones $T_1$ y $T_2$ de la Figura 17.2, y determine cuáles son serializables por conflicto (correctas) y cuáles no.

**17.18.** ¿Cuántas planificaciones *en serie* existen para las tres transacciones de la Figura 17.8(a)? ¿Cuáles son? ¿Cuál es la cantidad total de planificaciones posibles?

**17.19.** Escriba un programa para crear todas las planificaciones posibles para las tres transacciones de la Figura 17.8(a), y para determinar cuáles de ellas son serializables por conflicto y cuáles no. Por cada planificación serializable por conflicto, su programa debe imprimir la planificación y listar todas las planificaciones en serie equivalentes.

**17.20.** ¿Por qué en SQL se necesita una sentencia explícita de final de transacción, pero no una sentencia explícita de inicio?

**17.21.** Describa situaciones en las que cada uno de los distintos niveles de aislamiento sea de utilidad para el procesamiento de transacciones.

**17.22.** ¿Cuáles de las siguientes planificaciones son serializables (por conflicto)? Por cada planificación serializable, determine las planificaciones en serie equivalentes.

**a.** $r_1(X)$; $r_3(X)$; $w_1(X)$; $r_2(X)$; $w_3(X)$;

**b.** $r_1(X)$; $r_3(X)$; $w_3(X)$; $w_1(X)$; $r_2(X)$;

**c.** $r_3(X)$; $r_2(X)$; $w_3(X)$; $r_1(X)$; $w_1(X)$;

**d.** $r_3(X)$; $r_2(X)$; $r_1(X)$; $w_3(X)$; $w_1(X)$;

**17.23.** Considere las transacciones $T_1$, $T_2$ y $T_3$, y las planificaciones $S_1$ y $S_2$ que se ofrecen a continuación. Dibuje los gráficos de serialización (precedencia) para $S_1$ y $S_2$, y explique si cada planificación es o no es serializable. Si una planificación es serializable, escriba la(s) planificación(es) en serie equivalente(s).

$T_1$: $r_1(X)$; $r_1(Z)$; $w_1(X)$;

$T_2$: $r_2(Z)$; $r_2(Y)$; $w_2(Z)$; $w_2(Y)$;

$T_3$: $r_3(X)$; $r_3(Y)$; $w_3(Y)$;

$S_1$: $r_1(X)$; $r_2(Z)$; $r_1(Z)$; $r_3(X)$; $r_3(Y)$; $w_1(X)$; $w_3(Y)$; $r_2(Y)$; $w_2(Z)$; $w_2(Y)$;

$S_2$: $r_1(X)$; $r_2(Z)$; $r_3(X)$; $r_1(Z)$; $r_2(Y)$; $r_3(Y)$; $w_1(X)$; $w_2(Z)$; $w_3(Y)$; $w_2(Y)$;

**17.24.** Considere las planificaciones *S*3, *S4* y *S5*. Determine si cada planificación es estricta, que evita la anulación en cascada, recuperable o no recuperable. (Determine la condición de recuperabilidad más estricta que cada planificación satisfaga).

$S_3$: $r_1(X)$; $r_2(Z)$; $r_1(Z)$; $r_3(X)$; $r_3(Y)$; $w_1(X)$; $c_1$; $w_3(Y)$; $c_3$; $r_2(Y)$; $w_2(Z)$; $w_2(Y)$; $c_2$;

$S_4$: $r_1(X)$; $r_2(Z)$; $r_1(Z)$; $r_3(X)$; $r_3(Y)$; $w_1(X)$; $w_3(Y)$; $r_2(Y)$; $w_2(Z)$; $w_2(Y)$; $c_1$; $c_2$; $c_3$;

$S_5$: $r_1(X)$; $r_2(Z)$; $r_3(X)$; $r_1(Z)$; $r_2(Y)$; $r_3(Y)$; $w_1(X)$; $c_1$; $w_2(Z)$; $w_3(Y)$; $w_2(Y)$; $c_3$; $c_2$;

## Bibliografía seleccionada

Los conceptos de serialización se introdujeron en Gray y otros (1975). El concepto de transacción de base de datos se explicó por primera vez en Gray (1981). Gray ganó el codiciado ACM Turing Award en 1998 por su

trabajo sobre las transacciones en las bases de datos y la implementación de transacciones en los DBMSs relacionales. Bernstein, Hadzilacos, y Goodman (1987) se centra en las técnicas de control de la concurrencia y de recuperación en los sistemas de bases de datos centralizados y distribuidos; es una referencia excelente. Papadimitriou (1986) ofrece una perspectiva más teórica. Un gran libro de referencia de más de mil páginas es Gray y Reuter (1993), que ofrece una perspectiva más práctica de los conceptos y las técnicas del procesamiento de transacciones. Elmagarmid (1992) y Bhargava (1989) ofrecen colecciones de artículos de investigación sobre el procesamiento de transacciones. El soporte de transacciones en SQL se describe en Date y Darwen (1993). La serialización por vista se define en Yannakakis (1984). La recuperación de las planificaciones se explica en Hadzilacos (1983, 1988).

CAPÍTULO 18

# Técnicas de control de la concurrencia

En este capítulo explicamos algunas técnicas de control de la concurrencia que se utilizan para garantizar la ausencia de interferencias o la propiedad de aislamiento de las transacciones que se ejecutan simultáneamente. La mayoría de estas técnicas garantizan la serialización de las planificaciones (consulte la Sección 17.5), utilizando **protocolos** (conjuntos de reglas) que garantizan esa serialización. Un importante conjunto de protocolos emplea la técnica del **bloqueo** de los elementos de datos para evitar que varias transacciones accedan concurrentemente a los elementos; en la Sección 18.1 se describen algunos protocolos de bloqueo, que se utilizan en casi todos los DBMSs comerciales. Otro conjunto de protocolos de control de la concurrencia utilizan las **marcas de tiempo**. Una marca de tiempo es un identificador único generado por el sistema para cada transacción. Los protocolos de control de la concurrencia que utilizan la ordenación de marcas de tiempo para garantizar la serialización se describen en la Sección 18.2. En la Sección 18.3 explicamos los protocolos de control de la concurrencia **multiversión**, que utilizan varias versiones de un elemento de datos. En la Sección 18.4 presentamos un protocolo basado en el concepto de **validación** o **certificación** de una transacción después de que haya ejecutado sus operaciones; estos protocolos se denominan a veces **protocolos optimistas**.

Otro factor que afecta al control de la concurrencia es la **granularidad** de los elementos de datos (es decir, qué porción de la base de datos es representada por un elemento de datos). Un elemento puede ser tan pequeño como el valor de un atributo (campo) o tan grande como un bloque de disco, o incluso un fichero entero o la base de datos entera. En la Sección 18.5 explicamos la granularidad de los elementos. En la Sección 18.6 explicamos los problemas que pueden surgir con el control de la concurrencia cuando se utilizan los índices para procesar las transacciones. Por último, en la Sección 18.7 explicamos algunos problemas relacionados con el control de la concurrencia.

Son suficientes las Secciones 18.1, 18.5, 18.6 y 18.7, y posiblemente la 18.3.2, si el interés principal es una introducción a las técnicas de control de la concurrencia que más a menudo se utilizan en la práctica. Las demás técnicas tienen un interés principalmente teórico.

## 18.1 Técnicas de bloqueo en dos fases para controlar la concurrencia

Algunas de las principales técnicas que se utilizan para controlar la ejecución concurrente de transacciones están basadas en el concepto de bloqueo de elementos de datos. Un **bloqueo** es una variable asociada a un ele-

mento de datos que describe el estado de ese elemento respecto a las posibles operaciones que se le puedan aplicar. Generalmente, hay un bloqueo por cada elemento de datos de la base de datos. Los bloqueos se utilizan como un medio para sincronizar el acceso de las transacciones concurrentes a los elementos de la base de datos. En la Sección 18.1.1 explicamos la naturaleza y los tipos de bloqueos. Después, en la Sección 18.1.2 presentamos los protocolos que utilizan el bloqueo para garantizar la serialización de las planificaciones de transacciones. Por último, en la Sección 18.1.3 explicamos dos problemas asociados con el uso de bloqueos (interbloqueo e inanición), así como su manipulación.

## 18.1.1 Tipos de bloqueos y tablas de bloqueo del sistema

En el control de la concurrencia se utilizan varios tipos de bloqueos. A fin de introducir gradualmente los conceptos de bloqueo, primero explicamos los bloqueos binarios, que son sencillos pero restrictivos, por lo que no se utilizan en la práctica. Después explicamos los bloqueos compartidos/exclusivos, que ofrecen unas capacidades de bloqueo más generales y que se utilizan en la práctica en los esquemas de bloqueo de bases de datos. En la Sección 18.3.2 describimos un bloqueo de certificación y mostramos cómo puede utilizarse para mejorar el rendimiento de los protocolos de bloqueo.

**Bloqueos binarios.** Un **bloqueo binario** puede tener dos **estados** o **valores:** bloqueado y desbloqueado (o 1 y 0, para simplificar). Cada elemento *X* de la base de datos tiene un bloqueo distinto. Si el valor del bloqueo sobre *X* es 1, el elemento *X no podrá ser accedido* por una operación de base de datos que solicite el elemento. Si el valor del bloqueo sobre *X* es 0, es posible acceder al elemento cuando es solicitado. Nos referiremos al valor (o estado) actual del bloqueo asociado con el elemento *X* como **bloquear(*X*)**.

Con el bloqueo binario se utilizan dos operaciones, bloquear_elemento y desbloquear_elemento. Una transacción solicita acceso a un elemento *X* emitiendo primero una operación **bloquear_elemento(*X*)**. Si BLOQUEAR(*X*)=1, la transacción está obligada a esperar. Si BLOQUEAR(*X*)=0, se establece a 1 (la transacción **bloquea** el elemento) y la transacción tiene permiso para acceder al elemento *X*. Cuando la transacción ha terminado de utilizar el elemento, emite una operación **desbloquear_elemento(*X*)**, que asigna 0 a BLOQUEAR(*X*) (**desbloquea** el elemento) por lo que otras transacciones pueden acceder a *X*. Por tanto, un bloqueo binario impone la **exclusión mutua** en el elemento de datos. En la Figura 18.1 se muestra una descripción de las operaciones bloquear_elemento(*X*) y desbloquear_elemento(*X*).

Las operaciones bloquear_elemento y desbloquear_elemento deben implementarse como unidades indivisibles (conocidas como **secciones críticas** en los sistemas operativos); es decir, no debe permitirse la interpolación una vez iniciada una operación de bloqueo o desbloqueo hasta que la operación termina o la transacción espera. En la Figura 18.1, el comando esperar dentro de la operación bloquear_elemento(*X*) se implementa normalmente colocando la transacción en una cola de espera para el elemento *X* hasta que se desbloquea éste y la transacción obtiene acceso a él. Las demás transacciones que también quieren acceder a *X* se colocan en la misma cola. Por tanto, se considera que el comando esperar está fuera de la operación bloquear_elemento.

Es muy fácil implementar un bloqueo binario; basta con una variable de tipo binario, BLOQUEAR, asociada a cada elemento de datos *X* de la base de datos. En su forma más sencilla, un bloqueo puede ser un registro con tres campos: <Nombre_elemento_datos, BLOQUEAR, Transacción_de_bloqueo>, más una cola para las transacciones que están esperando a acceder al elemento. El sistema sólo necesita guardar registros de este tipo para los elementos que actualmente están bloqueados, y lo hace en una **tabla de bloqueo**, que puede organizarse como un fichero de dispersión. Los elementos que no figuran en esta tabla están desbloqueados. El DBMS tiene un **subsistema gestor de bloqueos** que rastrea y controla el acceso a los bloqueos.

En caso de utilizar este sencillo esquema de bloqueo binario, cada transacción debe cumplir las siguientes reglas:

1. Una transacción *T* debe emitir la operación bloquear_elemento(*X*) antes de que se ejecute cualquier operación leer_elemento(*X*) o escribir_elemento(*X*) en *T*.

**Figura 18.1.** Operaciones de bloqueo y desbloqueo para lo bloqueos binarios.

```
bloquear_elemento(X):
B: Si BLOQUEAR(X) = 0 (* elemento desbloqueado *)
       then BLOQUEAR(X) ← 1 (* bloquear el elemento *)
   entonces
       inicio
       esperar (hasta BLOQUEAR(X) = 0
           y el gestor de bloqueo retoma la transacción);
       ir a B
       fin;

desbloquear_elemento(X):
   BLOQUEAR(X) ← 0; (* desbloquear el elemento *)
   Si hay transacciones esperando
       entonces retomar una de las transacciones que espera;
```

2. Una transacción $T$ debe emitir la operación desbloquear_elemento($X$) después de haberse completado todas las operaciones leer_elemento($X$) y escribir_elemento($X$) de $T$.
3. Una transacción $T$ no emitirá una operación bloquear_elemento($X$) si ya posee el bloqueo del elemento $X$.[1]
4. Una transacción $T$ no emitirá una operación desbloquear_elemento($X$) a menos que ya posea el bloqueo del elemento $X$.

Estas reglas puede implementarlas el módulo gestor de bloqueos del DBMS. Entre las operaciones bloquear_elemento($X$) y desbloquear_elemento($X$) de la transacción $T$, se dice que $T$ **posee el bloqueo** del elemento $X$. A lo sumo, una transacción puede poseer el bloqueo de un elemento en particular. De este modo, dos transacciones no pueden acceder simultáneamente al mismo elemento.

**Bloqueos compartidos/exclusivos (o lectura/escritura).** El esquema de bloqueo binario anterior es demasiado restrictivo para los elementos de base de datos porque, como máximo, sólo una transacción puede poseer un bloqueo sobre un elemento dado. Debemos permitir que varias transacciones tengan acceso al mismo elemento $X$ si todas ellas acceden a $X$ sólo para leer. Sin embargo, si una transacción va a escribir un elemento $X$, debe tener acceso exclusivo a $X$. Con este fin, se utiliza un tipo de bloqueo diferente denominado **bloqueo de modo múltiple**. En este esquema (denominado **bloqueos compartido/exclusivo** o **de lectura/escritura**) hay tres operaciones de bloqueo: bloquear_lectura($X$), bloquear_escritura($X$) y desbloquear($X$). Un bloqueo asociado con un elemento $X$, BLOQUEAR($X$), tiene ahora tres posibles estados: *bloqueado para lectura*, *bloqueado para escritura* o *desbloqueado*. Un **elemento bloqueado para lectura** también se denomina de **lectura compartida** porque otras transacciones pueden leer el elemento, mientras que un **elemento bloqueado para escritura** se denomina de **escritura exclusiva** porque una sola transacción posee en exclusiva el bloqueo de un elemento.

Un método para implementar las operaciones anteriores en un bloqueo de lectura/escritura es hacer un seguimiento del número de transacciones que poseen un bloqueo compartido (lectura) sobre un elemento de la tabla de bloqueo. Cada registro de dicha tabla tendrá cuatro campos: <Nombre_elemento_datos, BLOQUEAR, Número_de_lecturas, Transacción(es)_bloqueo(s)>. Una vez más, para ahorrar espacio, el sistema debe mantener registros de bloqueo sólo para los elementos bloqueados de la tabla de bloqueo. El valor (estado) de BLOQUEAR puede ser bloqueado para lectura o bloqueado para escritura, adecuadamente codificado (si

[1] Esta regla se puede eliminar si modificamos la operación bloquear_elemento($X$) de la Figura 18.1 para que, si el elemento está actualmente bloqueado *por la transacción solicitante*, el bloqueo sea concedido.

asumimos que no se guardan registros en la tabla de bloqueo para los elementos desbloqueados). Si BLOQUEAR($X$)=bloqueado para escritura, el valor de Transacción(es)_bloqueo(s) es una sola transacción que almacena el bloqueo exclusivo (escritura) de $X$. Si BLOQUEAR($X$)=bloqueado para lectura, el valor de Transacción(es)_bloqueo(s) es una lista de una o más transacciones que poseen el bloqueo compartido (lectura) de $X$. En la Figura 18.2 se describen las tres operaciones, bloquear_lectura($X$), bloquear_escritura($X$) y desbloquear($X$).[2] Como antes, cada una de las tres operaciones debe considerarse como indivisible; no debe permitirse la interpolación una vez iniciada una de las operaciones, hasta que la operación termina concediendo el bloqueo, o la transacción se coloca en una cola de espera para el elemento.

Cuando utilizamos el esquema de bloqueo compartido/exclusivo, el sistema debe implementar las siguientes reglas:

1. Una transacción $T$ debe emitir la operación bloquear_lectura($X$) o bloquear_escritura($X$) antes de que se ejecute cualquier operación leer_elemento($X$) de $T$.
2. Una transacción $T$ debe emitir la operación bloquear_escritura($X$) antes de que se ejecute cualquier operación escribir_elemento($X$) de $T$.
3. Una transacción $T$ debe emitir la operación desbloquear($X$) una vez que se hayan completado todas las operaciones leer_elemento($X$) y escribir_elemento($X$) de $T$.[3]
4. Una transacción $T$ no emitirá una operación bloquear_lectura($X$) si ya posee un bloqueo de lectura (compartido) o de escritura (exclusivo) para el elemento $X$. Esta regla se puede hacer menos estricta, como veremos en breve.
5. Una transacción $T$ no emitirá una operación bloquear_escritura($X$) si ya posee un bloqueo de lectura (compartido) o de escritura (exclusivo) para el elemento $X$. Esta regla se puede hacer menos estricta, como veremos en breve.
6. Una transacción $T$ no emitirá una operación desbloquear($X$) a menos que ya posea un bloqueo de lectura (compartido) o de escritura (exclusivo) sobre el elemento $X$.

**Conversión de bloqueos.** En ocasiones, es deseable hacer menos estrictas las condiciones 4 y 5 del listado anterior para permitir una **conversión del bloqueo**; es decir, una transacción que ya posee un bloqueo sobre el elemento $X$ tiene permitido, bajo ciertas condiciones, **convertir** el bloqueo de un estado a otro. Por ejemplo, es posible para una transacción $T$ emitir una operación bloquear_lectura($X$) y más tarde **promocionar** el bloqueo emitiendo una operación bloquear_escritura($X$). Si $T$ es la única transacción que posee un bloqueo de lectura sobre $X$ en el momento de emitir la operación bloquear_escritura($X$), el bloqueo puede promocionarse; en caso contrario, la transacción debe esperar. También es posible para una transacción $T$ emitir una operación bloquear_escritura($X$) y más tarde **degradar** el bloqueo emitiendo una operación bloquear_lectura($X$). Cuando se utiliza la promoción y la degradación de bloqueos, la tabla de bloqueo debe incluir los identificadores de transacción en la estructura de registro de cada bloqueo [en el campo Transacción(es)_bloqueo(s)] para almacenar la información sobre las transacciones que poseen bloqueos sobre el elemento. Las descripciones de las operaciones bloquear_lectura($X$) y bloquear_escritura($X$) de la Figura 18.2 deben modificarse adecuadamente. Lo dejamos como ejercicio para el lector.

Con los bloqueos binarios o bloqueos de lectura/escritura en las transacciones, como describimos anteriormente, no está garantizada la serialización de las planificaciones. La Figura 18.3 muestra un ejemplo en el que se han seguido las reglas de bloqueo anteriores pero el resultado puede ser una planificación no serializable. Esto se debe a que en la Figura 18.3(a) los elementos $Y$ de $T_1$ y $X$ de $T_2$ se desbloquearon demasiado pronto. Esto permite que se produzca una planificación como la de la Figura 18.3(c), que no es una planificación

[2] Estos algoritmos no permiten la *actualización* o la *degradación* de los bloqueos, como se describe posteriormente en esta sección. El lector puede ampliar los algoritmos para que permitan estas operaciones adicionales.

[3] Esta regla se puede relajar para que una transacción pueda desbloquear un elemento, y bloquearlo de nuevo más tarde.

**Figura 18.2.** Bloqueo y desbloqueo de operaciones para bloqueos de dos modos (lectura/escritura o compartido/exclusivo).

```
bloquear_lectura(X):
B: Si BLOQUEAR(X) 5 "desbloqueado"
    entonces  inicio BLOQUEAR(X) ← "boqueado para lectura";
        Número_de_lecturas(X) ←1
        fin
sino Si BLOQUEAR(X) 5 "bloqueado para lectura"
    entonces Número_de_lecturas(X) ← Número_de_lecturas(X) 1 1
sino    inicio
    esperar (hasta BLOQUEAR(X) 5 "desbloqueado"
        y el gestor de bloqueos retoma la transacción);
    ir a B
    fin;

bloquear_escritura(X):
B: Si BLOQUEAR(X) 5 "desbloqueado"
        entonces BLOQUEAR(X) ← "bloqueado para escritura"
    sino    inicio
            esperar (hasta BLOQUEAR(X) 5 "desbloqueado"
                y el gestor de bloqueos retoma la transacción);
            ir a B
            fin;

desbloquear(X):
Si BLOQUEAR(X) 5 "bloqueado para escritura"
    entonces  inicio BLOQUEAR(X) ← "desbloqueado";
              retomar una de las transacciones en espera, si las hay
              fin
sino Si BLOQUEAR(X) 5 "bloqueado para lectura"
    entonces  inicio
              Número_de_lecturas(X) ← Número_de_lecturas(X) – 1;
              Si Número_de_lecturas(X) 5 0
                  entonces   inicio BLOQUEAR(X) 5 "desbloqueado";
                             retomar una de las transacciones en espera, si las hay
                             fin
              fin;
```

serializable y, por tanto, ofrece unos resultados incorrectos. Para garantizar la serialización, debemos seguir *un protocolo adicional* relativo al posicionamiento de las operaciones de bloqueo y desbloqueo en cada transacción. El protocolo mejor conocido, el bloqueo en dos fases, se describe en la siguiente sección.

## 18.1.2 Garantía de la serialización por el bloqueo en dos fases

Una transacción obedece el **protocolo de bloqueo en dos fases** si *todas* las operaciones de bloqueo (bloquear_lectura, bloquear_escritura) preceden a la *primera* operación de desbloqueo de la transacción.[4] Una

[4] No tiene relación con el protocolo de confirmación en dos fases para la recuperación en las bases de datos distribuidas (consulte el Capítulo 25).

**Figura 18.3.** Transacciones que no obedecen el bloqueo en dos fases. (a) Dos transacciones $T_1$ y $T_2$. (b) Resultado de posibles planificaciones en serie de $T_1$ y $T_2$. (c) Una planificación no serializable $S$ que utiliza bloqueos.

**(a)**

| $T_1$ | $T_2$ |
|---|---|
| bloquear_lectura($Y$);<br>leer_elemento($Y$);<br>desbloquear($Y$);<br>bloquear_escritura($X$);<br>leer_elemento($X$);<br>$X := X + Y$;<br>escribir_elemento($X$);<br>desbloquear($X$); | bloquear_lectura($X$);<br>leer_elemento($X$);<br>desbloquear($X$);<br>bloquear_escritura($Y$);<br>leer_elemento($Y$);<br>$Y := X + Y$;<br>escribir_elemento($Y$);<br>desbloquear($Y$); |

**(b)** Valores iniciales: $X$=20, $Y$=30

Resultado de la planificación en serie $T_1$ seguida de $T_2$: $X$=50, $Y$=80

Resultado de la planificación en serie $T_2$ seguida de $T_1$: $X$=70, $Y$=50

**(c)**

| $T_1$ | $T_2$ |
|---|---|
| bloquear_lectura($Y$);<br>leer_elemento($Y$);<br>desbloquear($Y$); | |
| | bloquear_lectura($X$);<br>leer_elemento($X$);<br>desbloquear($X$);<br>bloquear_escritura($Y$);<br>leer_elemento($Y$);<br>$Y := X + Y$;<br>escribir_elemento($Y$);<br>desbloquear($Y$); |
| bloquear_escritura($X$);<br>leer_elemento($X$);<br>$X := X + Y$;<br>escribir_elemento($X$);<br>desbloquear($X$); | |

Tiempo

Resultado de la planificación $S$: $X$=50, $Y$=50 (no serializable)

transacción de este tipo puede dividirse en dos fases: una **primera fase de expansión** o **crecimiento**, durante la cual pueden adquirirse bloqueos nuevos sobre los elementos, pero no pueden liberarse; y una **segunda fase de reducción**, en la que los bloqueos existentes se pueden liberar pero no se pueden adquirir bloqueos nuevos. Si está permitida la conversión de bloqueos, la promoción de los bloqueos (de bloqueado para lectura a bloqueado para escritura) debe realizarse durante la fase de expansión, y la degradación de los bloqueos (de bloqueado para escritura a bloqueado para lectura) debe realizarse en la segunda fase. Por tanto, una operación bloquear_lectura($X$) que degrada un bloqueo de escritura que se posee sobre $X$ sólo puede aparecer en la segunda fase.

Las transacciones $T_1$ y $T_2$ de la Figura 18.3(a) no obedecen el protocolo de bloqueo en dos fases porque la operación bloquear_escritura($X$) sigue a la operación desbloquear($Y$) de $T_1$, y, de forma parecida, la operación bloquear_escritura($Y$) sigue a la operación desbloquear($X$) de $T_2$. Si implementamos el bloqueo en dos fases, las transacciones pueden reescribirse como $T_1$' y $T_2$' (véase la Figura 18.4). Ahora, la planificación de la Figura 18.3(c) no está permitida para $T_1$' y $T_2$' (con el orden modificado de sus operaciones de bloqueo y desbloqueo) bajo las reglas de bloqueo descritas en la Sección 18.1.1 porque $T_1$' emitirá su operación bloquear_escritura($X$)

**Figura 18.4.** Transacciones $T_1'$ y $T_2'$, que son iguales a las transacciones $T_1$ y $T_2$ de la Figura 18.3, pero obedecen el protocolo de bloqueo en dos fases. Observe que pueden provocar un interbloqueo.

| $T_1'$ |
|---|
| bloquear_lectura($Y$);<br>leer_elemento($Y$);<br>bloquear_escritura($X$);<br>desbloquear($Y$)<br>leer_elemento($X$);<br>$X := X + Y$;<br>escribir_elemento($X$);<br>desbloquear($X$); |

| $T_2'$ |
|---|
| bloquear_lectura($X$);<br>leer_elemento($X$);<br>bloquear_escritura($Y$);<br>desbloquear($X$)<br>leer_elemento($Y$);<br>$Y := X + Y$;<br>escribir_elemento($Y$);<br>desbloquear($Y$); |

*antes* de desbloquear el elemento $Y$; en consecuencia, cuando $T_2'$ emite su operación bloquear_lectura($X$), se ve obligada a esperar hasta que $T_1'$ libera el bloqueo emitiendo una operación desbloquear($X$) en la planificación.

Puede demostrarse que, si *cada* transacción de una planificación obedece el protocolo de bloqueo en dos fases, *la planificación es serializable*, obviando la necesidad de comprobar la serialización de las planificaciones. El mecanismo de bloqueo, por la implementación de las reglas de bloqueo en dos fases, también implementa la serialización.

El bloqueo en dos fases puede limitar la cantidad de concurrencia que puede darse en una planificación, porque una transacción $T$ no puede liberar un elemento $X$ después de haberlo usado si $T$ debe bloquear más tarde un elemento adicional $Y$; o, por el contrario, $T$ debe bloquear el elemento adicional $Y$ antes de que lo necesite para poder liberar $X$. Por tanto, $X$ debe permanecer bloqueado por $T$ hasta que todos los elementos que la transacción tiene que leer o escribir hayan sido bloqueados; sólo entonces podrá $T$ liberar $X$. Mientras tanto, otra transacción que pretenda acceder a $X$ puede verse obligada a esperar, aunque $T$ haya terminado con $X$; por el contrario, si $Y$ es bloqueado antes de que se necesite, otra transacción que pretende acceder a $Y$ está obligada a esperar aunque $T$ ya no esté utilizando $Y$. Éste es el precio por garantizar la serialización de todas las planificaciones sin tener que comprobar las propias planificaciones.

**Bloqueo en dos fases básico, conservador, estricto y riguroso.** Son variaciones del bloqueo en dos fases (2PL). La técnica que acabamos de describir se conoce como **2PL básico**. Una variación conocida como **2PL conservador** (o **2PL estático**) requiere una transacción para bloquear todos los elementos a los que tendrá acceso *antes de comenzar a ejecutarse*, mediante la declaración previa de los conjuntos de lectura y escritura. Como recordará de la Sección 17.1.2, el **conjunto de lectura** de una transacción es el conjunto de todos los elementos que la transacción lee, y el **conjunto de escritura** es el conjunto de todos los elementos que escribe. Si no es posible bloquear cualquiera de los elementos predeclarados necesarios, la transacción no bloqueará ningún elemento; en cambio, esperará a que puedan bloquearse todos los elementos. El 2PL conservador es un protocolo libre de interbloqueos, como veremos en la Sección 18.1.3 cuando expliquemos el problema del interbloqueo. Sin embargo, es difícil de utilizar en la práctica debido a la necesidad de predeclarar los conjuntos de lectura y escritura, algo que no es posible en la mayoría de las situaciones.

En la práctica la variación más popular de 2PL es 2PL estricto, que asegura las planificaciones estrictas (consulte la Sección 17.4). En esta variación, una transacción $T$ no libera ninguno de sus bloqueos exclusivos (escritura) hasta *después* de confirmarse o abortar. Por tanto, ninguna otra transacción puede leer o escribir un elemento que es escrito por $T$ a menos que ésta se confirme, lo que lleva a una planificación estricta en cuanto a recuperabilidad. 2PL estricto no está libre de los interbloqueos. Una variación más restrictiva de 2PL estricto es **2PL riguroso**, que también garantiza planificaciones estrictas. En esta variación, una transacción

*T* no libera ninguno de sus bloqueos (exclusivos o compartidos) hasta después de su confirmación o cancelación. Es más fácil de implementar que 2PL estricto. La diferencia entre 2PL conservador y riguroso es que el primero debe bloquear todos sus elementos *antes de iniciarse*, de modo que una vez que la transacción empieza se encuentra en la segunda fase; el riguroso no desbloquea ninguno de sus elementos hasta *después de terminar* (por confirmación o por cancelación), de modo que la transacción se encuentra en la primera fase hasta que termina.

En muchos casos, el **subsistema de control de la concurrencia** es el responsable de generar las solicitudes de bloqueo para lectura y bloqueo para escritura. Por ejemplo, suponga que el sistema está preparado para el protocolo 2PL estricto. Después, siempre que la transacción *T* emita una operación leer_elemento(*X*), el sistema llamará a una operación bloquear_lectura(*X*) en nombre de *T*. Si el estado de BLOQUEAR(*X*) es bloqueado para escritura por alguna otra transacción *T*', el sistema coloca *T* en la cola de espera para el elemento *X*; en caso contrario, concede la solicitud bloquear_lectura(*X*) y permite la ejecución de la operación leer_elemento(*X*) de *T*. Por otro lado, si la transacción *T* emite una operación escribir_elemento(*X*), el sistema llama a la operación bloquear_escritura(*X*) en nombre de *T*. Si el estado de BLOQUEAR(*X*) es bloqueado para escritura o bloqueado para lectura por alguna otra transacción *T*', el sistema coloca *T* en la cola de espera para el elemento *X*; si el estado de BLOQUEAR(*X*) es bloqueado para lectura y *T* es la única transacción que posee el bloqueo de lectura sobre *X*, el sistema promociona el bloqueo a bloqueado para escritura y permite la operación escribir_elemento(*X*) por *T*; por último, si el estado de BLOQUEAR(*X*) es desbloqueado, el sistema concede la solicitud bloquear_escritura(*X*) y permite la ejecución de la operación escribir_elemento(*X*). Después de cada acción, el sistema debe actualizar en consecuencia su tabla de bloqueo.

Aunque el protocolo de boqueo en dos fases garantiza la serialización (es decir, cada planificación que se permite es serializable), no permite *todas las posibles* planificaciones serializables (es decir, algunas planificaciones serializables serán prohibidas por el protocolo). Además, el uso de bloqueos puede provocar dos problemas adicionales: interbloqueo e inanición. En la siguiente sección explicamos estos problemas y sus soluciones.

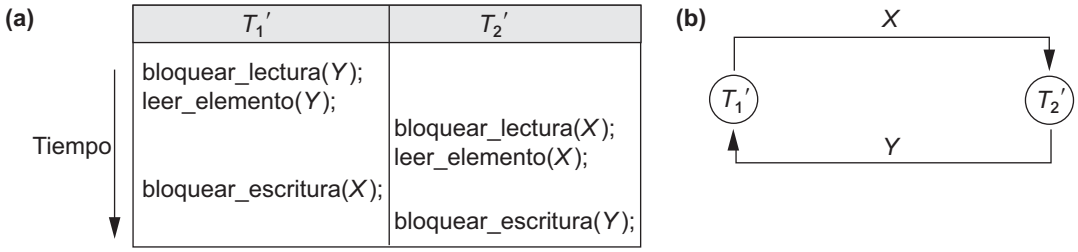
### 18.1.3 Interbloqueos e inanición

El **interbloqueo** se produce cuando *cada* transacción *T* en un conjunto de *dos o más transacciones* está esperando a algún elemento que está bloqueado por alguna otra transacción *T*' de dicho conjunto. Por tanto, cada transacción del conjunto está parada en espera a que una de las demás transacciones del conjunto libere el bloqueo sobre un elemento. En la Figura 18.5(a) se muestra un ejemplo sencillo, donde las transacciones $T_1$' y $T_2$' están interbloqueadas en una planificación parcial; $T_1$' está esperando a *X*, que está bloqueado por $T_2$', mientras que $T_2$' está parada en espera de *Y*, que está bloqueado por $T_1$'. Mientras tanto, ni $T_1$' ni $T_2$' ni cualquier otra transacción puede acceder a los elementos *X* e *Y*.

**Protocolos de prevención de interbloqueos.** Una forma de evitar el interbloqueo es utilizando un **protocolo de prevención del interbloqueo**,[5] que se utiliza en el bloqueo en dos fases conservador. Requiere que cada transacción bloquee *con antelación todos los elementos que necesita* (algo que normalmente no es un supuesto práctico); si alguno de los elementos no puede obtenerse, ninguno de los elementos se bloquea. En cambio, la transacción espera y después intenta de nuevo bloquear todos los elementos que necesita. Obviamente, esta solución limita la concurrencia. Un segundo protocolo, que también limita la concurrencia, consiste en *ordenar todos los elementos* de la base de datos y asegurarse de que una transacción que necesite varios elementos los bloqueará según ese orden. Esto obliga al programador (o al sistema) a conocer el orden de los elementos, que tampoco resulta práctico en el contexto de la base de datos.

[5] Estos protocolos no se utilizan en la práctica, ya sea debido a suposiciones poco realistas o una posible sobrecarga del sistema. La detección del interbloqueo y los tiempos limitados (siguiente sección) son más prácticos.

**Figura 18.5.** Ilustración del problema del interbloqueo. (a) Planificación parcial de $T_1'$ y $T_2'$ que está en estado de interbloqueo. (b) Un gráfico de espera para la planificación parcial de (a).




Se han propuesto otros esquemas de prevención del interbloqueo, que toman una decisión acerca de qué hacer con una transacción involucrada en una posible situación de interbloqueo: ¿debe bloquearse y hacerla esperar, o debe cancelarse, o debe apropiarse y abortar otra transacción? Estas técnicas utilizan el concepto de **marca de tiempo de transacción** TS($T$), que es un identificador único asignado a cada transacción. Normalmente, la marca de tiempo está basada en el orden en que las transacciones han sido iniciadas; por tanto, si la transacción $T_1$ empieza antes que la transacción $T_2$, entonces TS($T_1$) < TS($T_2$). La transacción *más antigua* tiene el valor de marca de tiempo *más pequeño*. Hay dos esquemas de prevención del interbloqueo, denominados esperar-morir y herir-esperar. Suponga que la transacción $T_i$ intenta bloquear un elemento $X$ pero no es capaz porque $X$ está bloqueado por alguna otra transacción $T_j$ con un bloqueo conflictivo. Las reglas que estos esquemas obedecen son las siguientes:

- **Esperar-morir.** Si TS($T_i$) < TS($T_j$), entonces ($T_i$ es más antigua que $T_j$) $T_i$ tiene permitido esperar; en caso contrario ($T_i$ es más nueva que $T_j$), aborta $T_i$ ($T_i$ *muere*) y se reinicia más tarde *con la misma marca de tiempo.*
- **Herir-esperar.** Si TS($T_i$) < TS($T_j$), entonces ($T_i$ es más antigua que $T_j$) aborta $T_j$ ($T_i$ *hiere a* $T_j$) y se reinicia más tarde *con la misma marca de tiempo;* en caso contrario ($T_i$ es más nueva que $T_j$), $T_i$ tiene que esperar.

En esperar-morir, una transacción más antigua tiene que esperar a una transacción más nueva, mientras que una transacción más nueva que solicita un elemento que posee una transacción más antigua es abortada y reiniciada. La metodología herir-esperar hace lo contrario: una transacción más nueva tiene que esperar a una transacción más antigua, mientras que una transacción más antigua que solicita un elemento que posee una transacción más nueva se *apropia* de la transacción más nueva abortándola. Los dos esquemas terminan con la cancelación de la transacción más nueva de las dos que pueden estar implicadas en un interbloqueo. Puede demostrarse que estas dos técnicas están *libres de interbloqueos*, puesto que en esperar-morir, las transacciones sólo esperan a las transacciones más nuevas, por lo que no se crean ciclos. De forma parecida, en herir-esperar, las transacciones sólo esperan a las transacciones más antiguas para que no se creen ciclos. Sin embargo, las dos técnicas pueden provocar que algunas transacciones aborten y reinicien innecesariamente, aunque estas transacciones *realmente nunca pueden provocar un interbloqueo*.

Otro grupo de protocolos que evitan el interbloqueo no requiere las marcas de tiempo; son los algoritmos de no espera y espera cautelosa. En el algoritmo de **no espera**, si una transacción no puede lograr un bloqueo, aborta y se reinicia tras un lapso de tiempo, sin comprobar si se ha producido o no un interbloqueo. Como este esquema puede provocar que las transacciones aborten y reinicien innecesariamente, se propuso el algoritmo de **espera cautelosa** para intentar reducir el número de abortos/reinicios innecesarios. Suponga que la transacción $T_i$ intenta bloquear un elemento $X$ pero no puede hacerlo porque $X$ está bloqueado por alguna otra transacción $T_j$ con un bloqueo por conflicto. Las reglas de la espera cautelosa son las siguientes:

- **Espera cautelosa.** Si $T_j$ no está bloqueada (no está esperando por algún otro elemento bloqueado), entonces $T_i$ es bloqueada y puede esperar; en caso contrario, $T_i$ aborta.

Se puede ver que la espera cautelosa está libre de interbloqueos, considerando el momento $b(T)$ en que cada transacción bloqueada $T$ fue bloqueada. Si las dos transacciones, $T_i$ y $T_j$, son bloqueadas, y $T_i$ está esperando a $T_j$, entonces $b(T_i) < b(T_j)$, puesto que $T_i$ sólo puede esperar a $T_j$ a la vez cuando $T_j$ no está bloqueada. Por tanto, los tiempos de bloqueo forman una ordenación total de todas las transacciones bloqueadas, por lo que no puede producirse ningún ciclo que provoque el interbloqueo.

**Detección del interbloqueo.** Esta metodología es más práctica para tratar con el interbloqueo; según ella, el sistema comprueba si realmente existe un estado de interbloqueo. Esta solución resulta atractiva si esperamos que haya poca interferencia entre las transacciones (es decir, si diferentes transacciones accederán raramente a los mismos elementos y al mismo tiempo). Esto puede suceder si las transacciones son cortas y cada una sólo bloquea unos cuantos elementos, o si la carga de transacciones es ligera. Por el contrario, si las transacciones son largas y cada una utiliza muchos elementos, o si la carga de la transacción es demasiado grande, puede ser beneficioso utilizar un esquema de prevención del interbloqueo.

Una forma sencilla de detectar un estado de interbloqueo es mediante un **gráfico de espera**. En este gráfico se crea un nodo por cada transacción que actualmente se está ejecutando. Siempre que una transacción $T_i$ está esperando a bloquear un elemento $X$ que actualmente está bloqueado por una transacción $T_j$, en el gráfico se crea un arco tendente ($T_i \rightarrow T_j$). Cuando $T_j$ libera el o los bloqueos sobre los elementos por los que $T_i$ está esperando, el arco tendente desaparece del gráfico de espera. Tenemos un estado de interbloqueo si, y sólo si, el gráfico de espera tiene un ciclo. Un problema de esta metodología es la cuestión de determinar *cuándo* el sistema debe comprobar si hay un interbloqueo. Pueden utilizarse criterios como el número de transacciones actualmente en ejecución o el lapso de tiempo que varias transacciones han estado esperando por los elementos bloqueados. La Figura 18.5(b) muestra el gráfico de espera para la planificación (parcial) de la Figura 18.5(a). Si el sistema se encuentra en estado de interbloqueo, algunas de las transacciones que lo provocan deben abortarse. La elección de cuáles hay que abortar se conoce como **selección de la víctima**. El algoritmo de selección de la víctima debe evitar generalmente la selección de transacciones que han estado ejecutándose durante mucho tiempo y que han realizado muchas actualizaciones, por lo que debe elegir transacciones que no han introducido muchos cambios.

**Tiempos limitados.** Otra solución más sencilla es el uso de tiempos limitados, un método práctico por su baja sobrecarga y simplicidad. En este método, si una transacción lleva en espera un tiempo superior a un tiempo definido por el sistema, éste asume que la transacción puede estar bloqueada y procede a su eliminación (independientemente de que exista o no un interbloqueo).

**Inanición (espera indefinida).** Otro problema que puede surgir cuando se utiliza el bloqueo es la inanición, que tiene lugar cuando una transacción no puede continuar durante un periodo de tiempo indeterminado mientras otras transacciones del sistema continúan con normalidad. Esto puede ocurrir si el esquema de espera para los elementos bloqueados es injusto, dando prioridad a algunas transacciones sobre otras. Una solución es tener un esquema de espera justo, como el uso de una cola del tipo **primero en llegar, primero en ser servido**; las transacciones pueden bloquear un elemento en el orden en que solicitaron originalmente el bloqueo. Otro esquema permite que algunas transacciones tengan prioridad sobre otras, pero aumenta la prioridad de una transacción que más tiempo lleva esperando, hasta que finalmente obtiene la prioridad más alta y procede. La inanición también puede darse debido a la selección de víctima, si el algoritmo selecciona repetidamente la misma transacción como víctima, lo que provoca su aborto y que nunca termine la ejecución. El algoritmo puede utilizar prioridades más altas para las transacciones que se han abortado varias veces para evitar este problema. Los esquemas esperar-morir y herir-esperar explicados anteriormente evitan la inanición.

# 18.2 Control de la concurrencia basado en la ordenación de marcas de tiempo

El uso de bloqueos, en combinación con el protocolo 2PL, garantiza la serialización de las planificaciones. Las planificaciones serializables producidas por 2PL tienen sus planificaciones en serie equivalentes basadas en el orden en que las transacciones en ejecución bloquean los elementos que adquieren. Si una transacción necesita un elemento que ya está bloqueado, puede verse obligada a esperar hasta que el elemento sea liberado. Una metodología diferente que garantiza la serialización implica el uso de marcas de tiempo para ordenar la ejecución de las transacciones para una planificación en serie equivalente. En la Sección 18.2.1 explicamos las marcas de tiempo y en la Sección 18.2.2 veremos cómo se implementa la serialización ordenando las transacciones en base a sus marcas de tiempo.

## 18.2.1 Marcas de tiempo

Una **marca de tiempo** es un identificador único creado por el DBMS para identificar una transacción. Normalmente, los valores de las marcas de tiempo son asignados en el orden en el que las transacciones son enviadas al sistema, por lo que una marca de tiempo puede verse como el *tiempo de inicio de la transacción*. Nos referiremos a la marca de tiempo de la transacción $T$ como **TS(*T*)**. Las técnicas de control de la concurrencia basadas en la ordenación de las marcas de tiempo no utilizan bloqueos; por tanto, *no se pueden producir interbloqueos*.

Las marcas de tiempo se pueden generar de varias formas. Una posibilidad es utilizar un contador que se incremente cada vez que su valor es asignado a una transacción. Las marcas de tiempo se numeran como 1, 2, 3, ... en este esquema. Un contador tiene un valor máximo finito, por lo que el sistema debe reiniciar periódicamente el contador a cero cuando no se ejecutan transacciones durante un corto periodo de tiempo. Otra forma de implementar las marcas de tiempo es utilizar el valor de fecha/hora actual del reloj del sistema y asegurarse de que no se han generado dos valores de marca de tiempo durante el mismo tictac del reloj.

## 18.2.2 Algoritmo de ordenación de marcas de tiempo

La idea para este esquema es ordenar las transacciones según sus marcas de tiempo. Una planificación en la que participan las marcas de tiempo es serializable, y la planificación en serie equivalente tendrá las transacciones ordenadas según sus marcas de tiempo. Es lo que se conoce como **ordenación por marcas de tiempo** (**TO**, *timestamp ordering*). Esto difiere de 2PL, en el que una planificación es serializable siendo equivalente a alguna planificación en serie permitida por los protocolos de bloqueo. No obstante, en la ordenación por marcas de tiempo, la planificación es equivalente a un *orden en serie particular* correspondiente al orden de las marcas de tiempo de las transacciones. El algoritmo debe garantizar que, para cada elemento accedido por *operaciones en conflicto* de la planificación, el orden con el que se accede a ese elemento no viola el orden de serialización. Para ello, el algoritmo asocia a cada elemento $X$ de la base de datos dos valores de marca de tiempo (**TS**):

1. **marca_lectura(*X*).** Es la **marca de tiempo de lectura** del elemento $X$; es la marca de tiempo más grande entre todas las marcas de tiempo de las transacciones que han leído satisfactoriamente el elemento $X$ (es decir, marca_lectura($X$) = TS($T$), donde $T$ es la transacción *más nueva* que ha leído $X$ con éxito).
2. **marca_escritura(*X*).** Es la **marca de tiempo de escritura** del elemento $X$; es la marca de tiempo más grande entre todas las marcas de tiempo de las transacciones que han escrito el elemento $X$ satisfactoriamente (es decir, marca_escritura($X$) = TS($T$), donde $T$ es la transacción *más nueva* que ha escrito satisfactoriamente el elemento $X$).

**Ordenación de marcas de tiempo (TO, *Timestamp Ordering*) básica.** Siempre que alguna transacción $T$ intenta emitir una operación leer_elemento($X$) o escribir_elemento($X$), el algoritmo **TO básico** compara la marca de tiempo de $T$ con marca_lectura($X$) y marca_escritura($X$) para garantizar que no se viola el orden de las marcas de tiempo de la transacción en ejecución. Si se viola este orden, la transacción $T$ se cancela y se vuelve a enviar al sistema como una transacción nueva con una *marca de tiempo nueva*. Si $T$ es abortada y anulada, también debe anularse cualquier transacción $T_1$ que pueda haber utilizado un valor escrito por $T$. De forma parecida, cualquier transacción $T_2$ que pueda haber utilizado un valor escrito por $T_1$ también debe anularse, y así sucesivamente. Este efecto se conoce como **anulación en cascada** y es uno de los problemas asociados con la TO básica, ya que no se garantiza que las planificaciones producidas sean recuperables. Debe implementarse un *protocolo adicional* para garantizar que las planificaciones son recuperables, evitan la anulación en cascada o son estrictas. Primero vamos a describir el algoritmo de la TO básica. El algoritmo de control de la concurrencia debe comprobar si las operaciones en conflicto violan la ordenación de las marcas de tiempo en estos dos casos:

1. Cuando la transaction $T$ emite una operación escribir_elemento($X$):
   a. Si marca_lectura($X$) > TS($T$) o si marca_escritura($X$) > TS($T$), abortar y anular $T$ y rechazar la operación. Debe hacerse esto porque alguna transacción *más nueva* con una marca de tiempo mayor que TS($T$) (y, por tanto, *posterior* a $T$ en el orden de las marcas de tiempo) ya ha leído o escrito el valor del elemento $X$ antes de que $T$ tuviera ocasión de escribir $X$, violándose así la ordenación de las marcas de tiempo.
   b. Si la condición del apartado (a) no se cumple, entonces se ejecuta la operación escribir_elemento($X$) de $T$ y se establece marca_escritura($X$) a TS($T$).
2. Cuando la transacción $T$ emite una operación leer_elemento($X$):
   a. Si marca_escritura($X$) > TS($T$), entonces se aborta y anula la transacción $T$ y se rechaza la operación. Debe hacerse esto porque alguna transacción *más nueva* con una marca de tiempo mayor que TS($T$) (y, por tanto, *posterior* a $T$ en el orden de las marcas de tiempo) ya ha escrito el valor del elemento $X$ antes de que $T$ tuviera la oportunidad de leer $X$.
   b. Si marca_escritura($X$) = TS($T$), entonces se ejecuta la operación leer_elemento($X$) de $T$ y se asigna a marca_lectura($X$) el valor *mayor* entre TS($T$) y la marca marca_lectura($X$) actual.

Por tanto, siempre que el algoritmo de TO básica detecta dos *operaciones en conflicto* que suceden en el orden incorrecto, rechaza la más tardía de las dos abortando la transacción que la emitió. Las planificaciones producidas por la TO básica son, por tanto, *serializables por conflicto*, como el protocolo 2PL. Sin embargo, algunas planificaciones que son posibles bajo un protocolo, no están permitidas bajo el otro. Por tanto, *ningún* protocolo permite *todas las posibles* planificaciones serializables. Como mencionamos anteriormente, con la ordenación de las marcas de tiempo no se produce el interbloqueo. Sin embargo, puede darse un reinicio cíclico (y, por tanto, la inanición) si una transacción es continuamente abortada y reiniciada.

**Ordenación de marcas de tiempo (TO, *Timestamp Ordering*) estricta.** Es una variante de la TO básica que garantiza que las planificaciones son tanto **estrictas** (para una recuperabilidad fácil) como serializables (por conflicto). En esta variación, una transacción $T$ que emite una operación leer_elemento($X$) o escribir_elemento($X$) de modo que TS($T$) > marca_escritura(X) ve *retardada* su operación de lectura o escritura hasta que la transacción $T'$ que *escribió* el valor de $X$ [por tanto, TS($T'$) = marca_escritura($X$)] es confirmada o abortada. Para implementar este algoritmo, es necesario simular el bloqueo de un elemento $X$ que ha sido escrito por la transacción $T'$ hasta que $T'$ es confirmada o abortada. Este algoritmo *no provoca el interbloqueo*, puesto que $T$ espera por $T'$ sólo si TS($T$) > TS($T'$).

**Regla de escritura de Thomas.** Es una modificación del algoritmo de TO básica, que no implementa la serialización por conflicto; pero rechaza menos operaciones de escritura, al modificar las comprobaciones de la operación escribir_elemento($X$) de este modo:

1. Si marca_lectura($X$) > TS($T$), entonces se aborta y anula $T$ y se rechaza la operación.
2. Si marca_escritura($X$) > TS($T$), entonces no se ejecuta la operación de escritura pero continúa el procesamiento. Esto es debido a que alguna transacción con una marca de tiempo mayor que TS($T$) (y, por tanto, posterior a $T$ en la ordenación de las marcas de tiempo) ya ha escrito el valor de $X$. Por consiguiente, debemos ignorar la operación escribir_elemento($X$) de $T$ porque ya está anticuada y obsoleta. Cualquier conflicto que surja de esta situación sería detectado por el caso (1).
3. Si no se cumple ninguna de las condiciones anteriores, se ejecuta la operación escribir_elemento($X$) de $T$ y se asigna a marca_escritura($X$) el valor de TS($T$).

## 18.3 Técnicas multiversión para controlar la concurrencia

Otros protocolos de control de la concurrencia se basan en conservar los valores antiguos de un elemento de datos cuando es actualizado. Es lo que se conoce como control multiversión de la concurrencia, porque se conservan varias versiones (valores) de un elemento. Cuando una transacción necesita acceder a un elemento, se elige una versión *adecuada* para mantener la serialización de la planificación actualmente en ejecución, si es posible. La idea es que algunas operaciones de lectura que serían rechazadas por otras técnicas, pueden ser aceptadas si *leen una versión más antigua* del elemento para mantener la serialización. Cuando una transacción escribe un elemento, escribe una *versión nueva* y se conserva la versión antigua. Algunos algoritmos de control de la concurrencia multiversión utilizan la serialización por vista, en lugar de la serialización por conflicto.

Un inconveniente obvio de las técnicas multiversión es que se necesita más almacenamiento para conservar las distintas versiones de los elementos de la base de datos. No obstante, es posible tener que conservar de todos modos las versiones más antiguas (por ejemplo, con fines de recuperación). Además, algunas aplicaciones de bases de datos requieren versiones más antiguas a fin de conservar un histórico de la evolución de los valores de los elementos de datos. El caso extremo es una *base de datos temporal* (consulte el Capítulo 24), que hace un seguimiento de todos los cambios y de las horas en las que se produjeron. En estos casos, no hay ninguna penalización por almacenamiento adicional para las técnicas multiversión, puesto que las versiones antiguas ya están guardadas.

Se han propuesto algunos esquemas de control multiversión de la concurrencia. Aquí vamos a ver dos esquemas, uno basado en la ordenación de las marcas de tiempo y otro basado en 2PL.

### 18.3.1 Técnica multiversión basada en la ordenación de las marcas de tiempo

En este método, el sistema guarda varias versiones $X_1$, $X_2$, ..., $X_k$ de cada elemento de datos $X$. Por *cada versión*, se conservan el valor de versión $X_i$ y estas dos marcas de tiempo:

1. marca_lectura($X_i$). La marca de tiempo de lectura de $X_i$ es el valor más grande de las marcas de tiempo de todas las transacciones que han leído satisfactoriamente la versión $X_i$.
2. marca_escritura($X_i$). La marca de tiempo de escritura de $X_i$ es la marca de tiempo de la transacción que escribió el valor de la versión $X_i$.

Siempre que una transacción $T$ tiene permitido ejecutar una operación escribir_elemento($X$), se crea una nueva versión, $X_{k+1}$, del elemento $X$, con los valores de marca_escritura($X_{k+1}$) y marca_lectura($X_{k+1}$) establecidos a TS($T$). Según el caso, cuando una transacción $T$ tiene permitido leer el valor de la versión $X_i$, a marca_lectura($X_i$) se le asigna el valor más grande entre la marca_lectura($X_i$) actual y TS($T$).

Para garantizar la serialización, se utilizan estas reglas:

1. Si la transacción $T$ emite una operación escribir_elemento($X$), y la versión $i$ de $X$ tiene el valor marca_escritura($X_i$) más alto de todas las versiones de $X$ que también es *menor o igual que* TS($T$), y marca_lectura($X_i$) > TS($T$), entonces se aborta y anula la transacción $T$; en caso contrario, se crea una nueva versión, $X_j$, de $X$ con marca_lectura($X_j$) = marca_escritura($X_j$) = TS($T$).
2. Si la transacción $T$ emite una operación leer_elemento($X$), se busca la versión $i$ de $X$ que tenga la marca_escritura($X_i$) más grande de todas las versiones de $X$ pero que sea *menor o igual que* TS($T$); entonces, se devuelve el valor de $X_i$ a la transacción $T$, y se establece el valor de marca_lectura($X_i$) al valor más grande entre TS($T$) y la marca_lectura($X_i$) actual.

Como vimos en el caso 2, una operación leer_elemento($X$) siempre es satisfactoria, ya que encuentra la versión apropiada $X_i$ que tiene que leer basándose en el valor de escribir_TS de las distintas versiones existentes de $X$. En el caso 1, no obstante, la transacción $T$ puede abortarse y anularse. Esto sucede si $T$ intenta escribir una versión de $X$ que podría haber leído otra transacción $T'$ cuya marca de tiempo es marca_lectura($X_i$); sin embargo, $T'$ ya ha leído la versión $X_i$, que fue escrita por la transacción con una marca de tiempo igual a marca_escritura($X_i$). Si surge este conflicto, se anula $T$; en caso contrario, se crea una nueva versión de $X$, escrita por la transacción $T$. Si se anula $T$, puede producirse una anulación en cascada. Por tanto, para garantizar la recuperabilidad, una transacción $T$ no debe poder confirmarse hasta que se hayan confirmado todas las transacciones que hayan escrito alguna versión leída por $T$.

### 18.3.2 Bloqueo en dos fases multiversión utilizando bloqueos de certificación

En este esquema de bloqueo multimodo hay *tres modos de bloqueo* para un elemento: lectura, escritura y certificación, en lugar de los dos modos (lectura y escritura) explicados anteriormente. Por tanto, el estado de BLOQUEAR($X$) para un elemento $X$ puede ser: bloqueado para lectura, bloqueado para escritura, bloqueado para certificación o desbloqueado. En el esquema de bloqueo estándar con los bloqueos de lectura y escritura (consulte la Sección 18.1.1), un bloqueo de escritura es un bloqueo exclusivo. Podemos describir la relación entre los bloqueos de lectura y escritura del esquema estándar mediante la **tabla de compatibilidad de bloqueos** de la Figura 18.6(a). Una entrada "Sí" significa que si una transacción $T$ posee el tipo de bloqueo especificado en la cabecera de la columna del elemento $X$ y si la transacción $T'$ solicita el tipo de bloqueo especificado en la cabecera de fila del mismo elemento $X$, entonces $T'$ *puede obtener el bloqueo* porque los modos de bloqueo son compatibles. Por el contrario, una entrada "No" en la tabla indica que los bloqueos son incompatibles, por lo que $T'$ *debe esperar* hasta que $T$ *libere* el bloqueo.

En el esquema de bloqueo estándar, una vez que una transacción obtiene un bloqueo de escritura sobre un elemento, ninguna otra transacción puede acceder a ese elemento. La idea tras un 2PL multiversión es permitir que otras transacciones $T'$ puedan leer un elemento $X$ mientras una transacción $T$ posee un bloqueo de escritura sobre $X$. Esto se consigue manteniendo *dos versiones* de cada elemento $X$; una de ellas deberá haber sido escrita por alguna transacción confirmada. La segunda versión $X'$ se crea cuando una transacción $T$ adquiere un bloqueo de escritura sobre el elemento. Otras transacciones pueden continuar leyendo la *versión confirmada* de $X$ mientras $T$ conserva el bloqueo de escritura. La transacción $T$ puede escribir el valor de $X'$ cuando lo necesite, sin que ello afecte al valor de la versión confirmada de $X$. Sin embargo, un vez que $T$ ya está confirmada, debe obtener un **bloqueo de certificación** sobre todos los elementos sobre los que actualmente tiene bloqueos de escritura antes de poder confirmarse. El bloqueo de certificación no es compatible con los bloqueos de lectura, por lo que la transacción puede que tenga que retrasar su confirmación hasta que todos sus elementos bloqueados para escritura sean liberados por cualquier transacción de lectura a fin de obtener los bloqueos de certificación. Una vez adquiridos estos bloqueos (que son bloqueos exclusivos), la versión confirmada de $X$ del elemento de datos se establece al valor de la versión $X'$, la versión $X'$ es descartada y los bloqueos de certificación son liberados. La tabla de compatibilidad de bloqueos para este esquema se muestra en la Figura 18.6(b).

**Figura 18.6.** Tablas de compatibilidad de bloqueos. (a) Tabla de compatibilidad para el esquema de bloqueo de lectura/escritura. (b) Tabla de compatibilidad para el esquema de bloqueo de lectura/escritura/certificación.

**(a)**

| | Lectura | Escritura |
|---|---|---|
| Lectura | Sí | No |
| Escritura | No | No |

**(b)**

| | Lectura | Escritura | Certificación |
|---|---|---|---|
| Lectura | Sí | Sí | No |
| Escritura | Sí | No | No |
| Certificación | No | No | No |

En este esquema 2PL multiversión, se pueden realizar varias operaciones de lectura simultáneamente a una operación de escritura (algo que los esquemas 2PL convencionales no permiten). Sin embargo, una transacción deberá esperar a confirmarse hasta obtener los bloqueos de certificación exclusivos de *todos los elementos* que ha modificado. Este esquema evita los abortos en cascada, ya que las transacciones sólo leen la versión de $X$ que fue escrita por una transacción confirmada. Sin embargo, si permitimos que un bloqueo de lectura se convierta en un bloqueo de escritura, se puede producir un interbloqueo, que debería tratarse con las variaciones de las técnicas que vimos en la Sección 18.1.3.

# 18.4 Técnicas de control de la concurrencia optimistas (validación)

En todas las técnicas de control de la concurrencia que hemos visto hasta ahora, se lleva a cabo cierto grado de comprobación *antes* de que pueda ejecutarse una operación de base de datos. Por ejemplo, en el bloqueo se realiza una comprobación para determinar si el elemento al que se va a acceder está bloqueado. En la ordenación de las marcas de tiempo, se compara la marca de tiempo de la transacción con las marcas de tiempo de lectura y escritura del elemento. Dicha comprobación representa un coste durante la ejecución de la transacción, que supone ralentizar las transacciones.

En las **técnicas de control de la concurrencia optimistas**, también conocidas como **técnicas de validación** o **certificación**, no se realiza comprobación alguna mientras la transacción se está ejecutando. Varios de los métodos de control de la concurrencia utilizan la técnica de la validación. Sólo describiremos un esquema, en el que las actualizaciones en la transacción no se aplican directamente a los elementos de la base de datos hasta que la transacción alcanza su final. Durante la ejecución de la transacción, todas las actualizaciones se aplican a *copias locales* de los elementos de datos que se conservan para la transacción.[6] Al término de la ejecución de la transacción, una **fase de validación** comprueba si alguna de las actualizaciones de la transacción viola la serialización. El sistema debe conservar cierta información necesaria para la fase de validación. Si la serialización no se viola, la transacción es confirmada y se actualiza la base de datos a partir de las copias locales; en caso contrario, la transacción es abortada y reiniciada posteriormente.

[6] ¡Esto puede asemejarse al almacenamiento de varias versiones de los elementos!

Hay tres fases para este protocolo de control de la concurrencia:

1. **Fase de lectura.** Una transacción puede leer valores de los elementos de datos confirmados de la base de datos. Sin embargo, las actualizaciones sólo se aplican a las copias locales (versiones) de los elementos de datos conservadas en el espacio de trabajo de la transacción.
2. **Fase de validación.** La comprobación se realiza para garantizar que la serialización no será violada si las actualizaciones de la transacción se aplican a la base de datos.
3. **Fase de escritura.** Si la fase de validación es satisfactoria, las actualizaciones de la transacción son aplicadas a la base de datos; en caso contrario, las actualizaciones se descartan y se reinicia la transacción.

La idea tras el control optimista de la concurrencia es hacer todas las comprobaciones de inmediato; por tanto, la ejecución de la transacción procede con un coste mínimo hasta alcanzar la fase de validación. Si hay una ligera interferencia entre las transacciones, la mayoría serán validadas satisfactoriamente. Sin embargo, si la interferencia es demasiada, muchas transacciones que se ejecutan hasta su terminación verán descartados sus resultados y deberán reiniciarse más tarde. Bajo estas circunstancias, las técnicas optimistas no funcionan bien. Se llaman así porque asumen que se producirá una pequeña interferencia y, por tanto, que no hay necesidad de hacer una comprobación durante la ejecución de la transacción.

El protocolo optimista que describimos utiliza las marcas de tiempo de las transacciones y requiere que el sistema mantenga el control de los conjuntos de escritura y lectura de las transacciones. Además, de cada transacción hay que guardar las horas de *inicio* y *terminación* de alguna de las tres fases. Recuerde que el conjunto de escritura de una transacción es el conjunto de elementos que escribe, y el conjunto de lectura es el conjunto de elementos que lee. En la fase de validación de la transacción $T_i$, el protocolo comprueba que $T_i$ no interfiere con ninguna transacción confirmada ni con otras transacciones que actualmente estén en su fase de validación. La fase de validación para $T_i$ comprueba que, para *cada* transacción $T_j$ que se confirma o está en su fase de validación, se cumple *una* de las siguientes condiciones:

1. La transacción $T_j$ completa su fase de escritura antes de que $T_i$ inicie su fase de lectura.
2. $T_i$ inicia su fase de escritura después de que $T_j$ complete su fase de escritura, y el conjunto de lectura de $T_i$ no tiene elementos en común con el conjunto de escritura de $T_j$.
3. Ni el conjunto de lectura ni el conjunto de escritura de $T_i$ tienen elementos en común con el conjunto de escritura de $T_j$, y $T_j$ completa su fase de lectura antes de que $T_i$ complete su fase de lectura.

Al validar la transacción $T_i$, primero se comprueba la primera condición para cada transacción $T_j$, puesto que la condición (1) es la más sencilla de comprobar. Sólo si la condición (1) es falsa se comprueba la condición (2), y sólo si la condición (2) es falsa se comprueba la condición (3) (la más compleja de evaluar). Si se cumple cualquiera de estas tres condiciones, no hay interferencia y $T_i$ se valida satisfactoriamente. Si *ninguna* de estas tres condiciones se cumple, la validación de la transacción $T_i$ falla y es abortada y reiniciada más tarde debido a que puede haberse producido alguna interferencia.

## 18.5 Granularidad de los elementos de datos y bloqueo de la granularidad múltiple

Todas las técnicas de control de la concurrencia asumen que la base de datos está formada por un cierto número de elementos de datos. Un elemento de base de datos puede ser alguna de estas cosas:

- Un registro de la base de datos.
- El valor de un campo de un registro de la base de datos.
- Un bloque de disco.

- Un fichero entero.
- La base de datos entera.

La granularidad puede afectar al rendimiento del control de la concurrencia y de la recuperación. En la Sección 18.5.1 explicamos algunas de las contrapartidas con respecto a elegir el nivel de granularidad utilizado para el bloqueo, mientras que en la Sección 18.5.2 explicamos un esquema de bloqueo de granularidad múltiple, en el que es posible cambiar dinámicamente el nivel de granularidad (tamaño del elemento de datos).

### 18.5.1 Consideraciones sobre el nivel de granularidad para el bloqueo

El tamaño de los elementos de datos se denomina con frecuencia **granularidad del elemento de datos**. La *granularidad fina* se refiere a los tamaños de elemento pequeños, mientras que la *granularidad gruesa* se refiere a los tamaños de elemento más grandes. Al elegir el tamaño del elemento de datos hay que tener en consideración varias contrapartidas. Explicaremos el tamaño del elemento de datos en el contexto del bloqueo, aunque pueden desarrollarse unos argumentos parecidos para otras técnicas de control de la concurrencia.

En primer lugar, observe que cuanto mayor es el tamaño del elemento de datos, más bajo es el grado de concurrencia permitido. Por ejemplo, si el tamaño del elemento de datos es un bloque de disco, una transacción $T$ que necesita bloquear un registro $B$ debe bloquear el bloque de disco entero $X$ que contiene $B$ porque un bloqueo está asociado con el elemento de datos entero (bloque). Ahora, si otra transacción $S$ quiere bloquear un registro $C$ diferente que resulta residir en el mismo bloque $X$ en un modo de bloqueo en conflicto, se ve obligada a esperar. Si el tamaño del elemento de datos fuera un registro, la transacción $S$ podría proceder, porque estaría bloqueando un elemento de datos diferente (registro).

Por otro lado, cuanto más pequeño es el tamaño del elemento de datos, mayor número de elementos hay en la base de datos. Como cada elemento está asociado con un bloqueo, el sistema tendrá una cantidad mayor de bloqueos activos que el gestor de bloqueos deberá controlar. Se realizarán más operaciones de bloqueo y desbloqueo, lo que provocará una sobrecarga más alta. Además, se necesitará más espacio de almacenamiento para la tabla de bloqueo. En el caso de las marcas de tiempo, se necesita almacenamiento para las marcas de lectura y de escritura de cada elemento de datos, y habrá una sobrecarga parecida para manipular una gran cantidad de elementos.

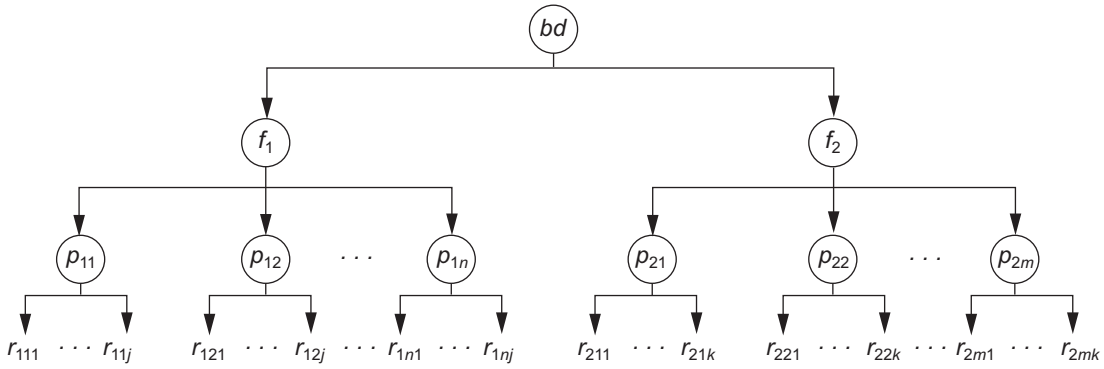
Dado lo anterior, una cuestión obvia es la siguiente: ¿cuál es el mejor tamaño de elemento? La respuesta es que *depende de los tipos de transacciones implicadas*. Si una transacción típica accede a pocos registros, lo mejor es tener una granularidad equivalente a un registro. Por otro lado, si una transacción accede normalmente a muchos registros del mismo fichero, lo mejor es tener una granularidad equivalente a un bloque o un fichero, de modo que la transacción considerará todos los registros como uno (o unos cuantos) elementos de datos.

### 18.5.2 Bloqueo por nivel de granularidad múltiple

Ya que el mejor tamaño de granularidad depende de la transacción dada, parece adecuado que un sistema de bases de datos deba soportar varios niveles de granularidad, donde el nivel de granularidad puede ser diferente para distintas combinaciones de transacciones. La Figura 18.7 muestra una sencilla jerarquía de granularidad con una base de datos que contiene dos ficheros, cada uno de ellos con varias páginas de disco, y cada página con varios registros. Esto puede utilizarse para ilustrar un protocolo 2PL de **nivel de granularidad múltiple**, donde es posible que se solicite un bloqueo a cualquier nivel. Sin embargo, se necesitarán tipos de bloqueos adicionales para soportar eficazmente un protocolo semejante.

Considere el siguiente escenario, únicamente con los tipos de bloqueo compartido y exclusivo, que se refiere al ejemplo de la Figura 18.7. Suponga que la transacción $T_1$ quiere actualizar *todos los registros* del fichero

**Figura 18.7.** Jerarquía de granularidad para ilustrar el bloqueo por nivel de granularidad múltiple.




$f_1$, y $T_1$ solicita y se le concede un bloqueo exclusivo para $f_1$. Después, todas las páginas de $f_1$ ($p_{11}$ a $p_{1n}$) (y los registros contenidos en esas páginas) son bloqueadas en modo exclusivo. Esto es beneficioso para $T_1$ porque la configuración de un solo bloqueo a nivel de fichero es más eficaz que la configuración de $n$ bloqueos a nivel de página o tener que bloquear independientemente cada registro. Ahora, piense en otra transacción $T_2$ que sólo quiere leer el registro $r_{1nj}$ de la página $p_{1n}$ del fichero $f_1$; entonces $T_2$ solicitaría un bloqueo a nivel de registro compartido sobre $r_{1nj}$. Sin embargo, el sistema de bases de datos (es decir, el gestor de transacciones o, más concretamente, el gestor de bloqueos) debe verificar la compatibilidad del bloqueo solicitado con los bloqueos ya existentes. Una forma de verificar esto es atravesando el árbol desde la hoja $r_{1nj}$ hasta $p_{1n}$ hasta $f_1$ hasta *bd*. Si en cualquier momento surge un bloqueo conflictivo en cualquiera de estos elementos, entonces la solicitud de bloqueo para $r_{1nj}$ es denegada y $T_2$ es bloqueada y deberá esperar. Esta travesía sería bastante eficaz.

Sin embargo, ¿qué pasaría si la solicitud de la transacción $T_2$ llegara *antes* que la solicitud de la transacción $T_1$? En este caso, el bloqueo de registro compartido es concedido a $T_2$ para $r_{1nj}$, pero cuando es solicitado el bloqueo a nivel de fichero de $T_1$, al gestor de bloqueos le resulta muy difícil comprobar todos los nodos (páginas y registros) que son descendientes del nodo $f_1$ en busca de un conflicto por bloqueo. Esto sería muy ineficaz y acabaría con el propósito de tener bloqueos de nivel de granularidad múltiple. Para que resulte práctico el bloqueo por nivel de granularidad múltiple, son necesarios otros tipos de bloqueos, denominados **bloqueos de intención**. La idea es que una transacción indique, junto con la ruta desde la raíz hasta el nodo deseado, el tipo de bloqueo (compartido o exclusivo) que requerirá de uno de los descendientes del nodo. Hay tres tipos de bloqueos de intención:

1. Bloqueo de intención compartida (IS) indica que se solicitará un(os) bloqueo(s) compartido(s) en algún(os) (de los) nodo(s) descendiente(s).
2. Bloqueo de intención exclusiva (IX) indica que se solicitará un(os) bloqueo(s) exclusivo(s) en algún(os) (de los) nodo(s) descendiente(s).
3. Bloqueo compartido con intención exclusiva (SIX) indica que el nodo actual está bloqueado en modo compartido pero se solicitará un(os) bloqueo(s) exclusivo(s) en algún(os) (de los) nodo(s) descendiente(s).

La tabla de compatibilidad de los tres bloqueos de intención, y los bloqueos compartido y exclusivo, se muestran en la Figura 18.8. Además de la introducción de los tres tipos de bloqueos de intención, debe utilizarse un protocolo de bloqueo adecuado. El protocolo de **bloqueo por nivel de granularidad múltiple** (**MGL**, ***multiple granularity locking***) consiste en las siguientes reglas:

1. Debe adherirse a la compatibilidad de bloqueo (basándose en la Figura 18.8).
2. La raíz del árbol debe bloquearse primero, en cualquier modo.

**Figura 18.8.** Matriz de compatibilidad de bloqueos para el bloqueo de granularidad múltiple.

| | IS | IX | S | SIX | X |
|---|---|---|---|---|---|
| IS | Sí | Sí | Sí | Sí | No |
| IX | Sí | Sí | No | No | No |
| S | Sí | No | Sí | No | No |
| SIX | Sí | No | No | No | No |
| X | No | No | No | No | No |

3. Un nodo *N* puede ser bloqueado por una transacción *T* en modo S o IS sólo si el nodo padre de *N* ya está bloqueado por la transacción *T* en modo IS o IX.
4. Un nodo *N* puede ser bloqueado por una transacción *T* en modo X, IX o SIX sólo si el padre del nodo *N* ya está bloqueado por la transacción *T* en modo IX o SIX.
5. Una transacción *T* puede bloquear un nodo sólo si no ha desbloqueado ningún nodo (para implementar el protocolo 2PL).
6. Una transacción *T* puede desbloquear un nodo, *N*, sólo si ninguno de los hijos del nodo *N* está actualmente bloqueado por *T*.

La regla 1 simplemente dice que no pueden concederse los bloqueos en conflicto. Las reglas 2, 3 y 4 determinan las condiciones para que una transacción pueda bloquear un nodo dado en cualquiera de los modos de bloqueo. Las reglas 5 y 6 del protocolo MGL implementan las reglas 2PL para producir planificaciones serializables. Para ilustrar el protocolo MGL con la jerarquía de base de datos de la Figura 18.7, considere estas tres transacciones:

1. $T_1$ quiere actualizar los registros $r_{111}$ y $r_{211}$.
2. $T_2$ quiere actualizar todos los registros de la página $p_{12}$.
3. $T_3$ quiere leer el registro $r_{11j}$ y el fichero $f_2$ entero.

La Figura 18.9 muestra una posible planificación serializable para estas tres transacciones. Sólo mostramos las operaciones de bloqueo y desbloqueo. Utilizamos la notación <tipo_bloqueo>(<elemento>) para mostrar las operaciones de bloqueo de la transacción.

El protocolo de nivel de granularidad múltiple es especialmente apropiado para procesar una mezcla de transacciones que incluya lo siguiente: (1) transacciones cortas que sólo acceden a unos pocos elementos (registros o campos) y (2) transacciones largas que acceden a ficheros enteros. En este entorno, con un protocolo semejante se incurre en menos bloqueos de transacciones y menos sobrecarga por bloqueo, en comparación con un método de bloqueo de granularidad de un solo nivel.

## 18.6 Uso de bloqueos para controlar la concurrencia en los índices

El bloqueo en dos fases también se puede aplicar a los índices (consulte el Capítulo 14), donde los nodos de un índice son equiparables a las páginas de disco. Sin embargo, la posesión de bloqueos en las páginas de índice hasta la fase de reducción de 2PL puede provocar una cantidad indebida de bloqueos de transacciones

**Figura 18.9.** Operaciones de bloqueo para ilustrar una planificación serializable.

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| IX(*bd*) | | |
| IX($f_1$) | | |
| | IX(*bd*) | |
| | | IS(*bd*) |
| | | IS($f_1$) |
| | | IS($p_{11}$) |
| IX($p_{11}$) | | |
| X($r_{111}$) | | |
| | IX($f_1$) | |
| | X($p_{12}$) | |
| | | S($r_{11j}$) |
| IX($f_2$) | | |
| IX($p_{21}$) | | |
| X($p_{211}$) | | |
| desbloquear($r_{211}$) | | |
| desbloquear($p_{21}$) | | |
| desbloquear($f_2$) | | |
| | | S($f_2$) |
| | desbloquear($p_{12}$) | |
| | desbloquear($f_1$) | |
| | desbloquear(*bd*) | |
| desbloquear($r_{111}$) | | |
| desbloquear($p_{11}$) | | |
| desbloquear($f_1$) | | |
| desbloquear(*bd*) | | |
| | | desbloquear($r_{11j}$) |
| | | desbloquear($p_{11}$) |
| | | desbloquear($f_1$) |
| | | desbloquear($f_2$) |
| | | desbloquear(*bd*) |

porque la búsqueda en un índice siempre *empieza* por la *raíz*. Por consiguiente, si una transacción quiere insertar un registro (operación de escritura), se bloqueará la raíz en modo exclusivo, para que todas las demás solicitudes de bloqueo en conflicto para el índice deban esperar hasta que la transacción entre en la fase de reducción. Esto bloquea el acceso al índice a todas las demás transacciones, por lo que en la práctica debemos utilizar otras metodologías para bloquear un índice.

La estructura de árbol del índice puede beneficiarse de cuando se desarrolla un esquema de control de la concurrencia. Por ejemplo, cuando se está ejecutando una búsqueda en un índice (operación de lectura), se recorre el árbol desde la raíz hasta una hoja. Una vez que se ha accedido a un nodo del nivel inferior, los nodos del nivel superior de esa ruta no se utilizarán de nuevo. Así, una vez obtenido un bloqueo de lectura sobre un nodo hijo, se puede liberar el bloqueo sobre el padre. Cuando se está aplicando una inserción a un nodo hoja (es decir, cuando se insertan una clave y un puntero), entonces debe bloquearse un nodo hoja específico en modo exclusivo. No obstante, si ese nodo no está lleno, la inserción no provocará cambios en los nodos de índice del nivel superior, lo que implica que no deben bloquearse exclusivamente.

Un método más conservador para las inserciones sería bloquear el nodo raíz en modo exclusivo y después acceder al nodo hijo apropiado de la raíz. Si el nodo hijo no está lleno, entonces el bloqueo sobre el nodo raíz

puede liberarse. Este método puede aplicarse hasta llegar a la hoja, lo que normalmente suponen tres o cuatro niveles desde la raíz. Aunque se posean los bloqueos exclusivos, se liberan pronto. Una alternativa, el **método más optimista**, sería solicitar y mantener *bloqueos compartidos* sobre los nodos que llevan hasta el nodo hoja, con un *bloqueo exclusivo* sobre esa hoja. Si la inserción provoca la división de la hoja, la inserción se propagará al o los nodos del nivel superior. Después, los bloqueos sobre el o los nodos del nivel superior pueden actualizarse al modo exclusivo.

Otro método para bloquear un índice es utilizar una variante del árbol $B^+$, denominada **árbol de enlace B**. En este tipo de árbol, los nodos hermanos del mismo nivel se enlazan en cada nivel. Esto permite el uso de bloqueos compartidos al solicitar una página, y requiere que el bloqueo sea liberado antes de acceder al nodo hijo. En una operación de inserción, el bloqueo compartido sobre un nodo se actualizaría al modo exclusivo. Si se produce una división, hay que volver a bloquear el nodo padre en modo exclusivo. Hay una complicación en las operaciones de búsqueda ejecutadas concurrentemente con la actualización. Suponga que una operación de actualización concurrente sigue la misma ruta que la búsqueda, e inserta una entrada nueva en el nodo hoja. Además, suponga que la inserción provoca la división del nodo hoja. Cuando la inserción se ha realizado, se reanuda el proceso de búsqueda, siguiendo el puntero hasta la hoja deseada, sólo para ver que la clave que se está buscando no está presente porque la división la ha movido a un nodo hoja nuevo, que sería el *hermano derecho* del nodo hoja original. Sin embargo, el proceso de búsqueda todavía puede tener éxito si sigue el puntero (enlace) desde el nodo hoja original hasta su hermano derecho, adonde se habrá movido la clave deseada.

La manipulación de la eliminación, donde se combinan dos o más nodos del árbol de índice, también forma parte del protocolo de concurrencia de árbol de enlace B. En este caso, se mantienen los bloqueos sobre los nodos que se van a combinar, así como un bloqueo sobre el padre de los dos nodos a combinar.

# 18.7 Otros problemas del control de la concurrencia

En esta sección veremos algunos problemas más relacionados con el control de la concurrencia. En la Sección 18.7.1 explicamos los problemas asociados con la inserción y la eliminación de registros y lo que se conoce como *problema del fantasma*, que puede surgir cuando se insertan registros. Este problema se describió en la Sección 17.6 como un problema potencial que requiere una medida de control de la concurrencia. En la Sección 18.7.2 explicamos los problemas que se pueden dar cuando una transacción visualiza algunos datos en el monitor antes de confirmarse, y resulta que después se cancela.

## 18.7.1 Inserción, eliminación y registros fantasma

Cuando se **inserta** un nuevo elemento de datos en la base de datos, es obvio que no es posible acceder a él hasta haberse creado el elemento y completado la operación de inserción. En un entorno de bloqueo, se puede crear un bloqueo para el elemento y establecer el modo exclusivo (escritura); el bloqueo puede liberarse al mismo tiempo que se liberan otros bloqueos de escritura, en base al protocolo de control de la concurrencia que se esté utilizando. En el caso de un protocolo basado en marcas de tiempo, las marcas de tiempo de lectura y escritura del elemento nuevo se establecen a la marca de tiempo de la transacción que se está creando.

A continuación, piense en la **operación de eliminación** aplicada a un elemento de datos existente. En los protocolos de bloqueo, de nuevo hay que obtener un bloqueo exclusivo (escritura) antes de que la transacción pueda eliminar el elemento. Para la ordenación de las marcas de tiempo, el protocolo debe garantizar que ninguna transacción posterior ha leído o escrito el elemento antes de permitir la eliminación del elemento.

La situación conocida como **problema del fantasma** se da cuando un registro nuevo que una transacción $T$ está insertando satisface una condición que un conjunto de registros accedido por otra transacción $T'$ debe

satisfacer. Por ejemplo, suponga que la transacción *T* está insertando un registro EMPLEADO nuevo cuyo Dno = 5, mientras que la transacción *T'* está accediendo a todos los registros EMPLEADO cuyo Dno = 5 (por ejemplo, para sumar todos los sueldos a fin de calcular el presupuesto en personal del departamento 5). Si el orden en serie equivalente es *T* seguida por *T'*, entonces *T'* debe leer el registro EMPLEADO nuevo e incluir su Sueldo en la suma. Si el orden es *T'* seguida por *T*, no se incluiría el sueldo nuevo. Aunque las transacciones entran lógicamente en conflicto, en el último caso no hay realmente ningún registro (elemento de datos) en común entre las dos transacciones, puesto que *T'* puede haber bloqueado todos los registros con Dno = 5 *antes* de que *T* inserte el registro nuevo. Esto es porque el registro que provoca el conflicto es un **registro fantasma** que ha aparecido de repente en la base de datos en la que se ha insertado. Si otras operaciones con las dos transacciones entran en conflicto, es posible que el protocolo de control de la concurrencia no reconozca el conflicto debido al registro fantasma.

Una solución a este problema es utilizar el **bloqueo de índice**, que explicamos en la Sección 18.6. Como recordará del Capítulo 14, un índice incluye entradas que tienen el valor de un atributo, más un conjunto de punteros a todos los registros del fichero que tienen ese valor. Por ejemplo, un índice con el campo Dno de EMPLEADO incluiría una entrada para cada valor de Dno distinto, más un conjunto de punteros a todos los registros de EMPLEADO que tienen ese valor. Si la entrada de índice se bloquea antes de que el registro pueda ser accedido, entonces el conflicto con el registro fantasma puede detectarse, porque la transacción *T'* solicitaría un bloqueo de lectura sobre la entrada de índice correspondiente a Dno = 5, y *T* solicitaría un bloqueo de escritura sobre la misma entrada *antes* de que ellas colocaran los bloqueos sobre los registros actuales. Como los bloqueos de índice entrarían en conflicto, se detectaría el conflicto del fantasma.

Una técnica más genérica, denominada **bloqueo de predicado**, bloquearía de un modo parecido el acceso a todos los registros que satisfacen un *predicado* (condición) arbitrario; sin embargo, los bloqueos de predicado han demostrado ser difíciles de implementar con eficacia.

### 18.7.2 Transacciones interactivas

Otro problema surge cuando las transacciones interactivas leen la entrada y escriben la salida en un dispositivo interactivo, como el monitor, antes de ser confirmadas. El problema es que un usuario puede introducir un valor para un elemento de datos en una transacción *T* basándose en el valor escrito en pantalla por la transacción *T'*, que puede que no esté confirmada. Esta dependencia entre *T* y *T'* no puede ser modelada por el método de control de la concurrencia del sistema, ya que sólo está basado en el usuario que interactúa con dos transacciones.

Una forma de tratar este problema consiste en posponer la salida de las transacciones a pantalla hasta que han sido confirmadas.

### 18.7.3 Cerrojos

Los bloqueos que se mantienen durante poco tiempo se conocen normalmente como **cerrojos**. Los cerrojos no obedecen el protocolo de control de la concurrencia usual, como el bloqueo en dos fases. Por ejemplo, puede utilizarse un cerrojo para garantizar la integridad física de una página cuando se está escribiendo ésta desde el búfer al disco; se adquiriría un cerrojo para la página, se escribiría la página en el disco y, por último, se liberaría el cerrojo.

## 18.8 Resumen

En este capítulo hemos explicado las técnicas DBMS para controlar la concurrencia. Empezamos con los protocolos basados en los bloqueos, que son, de lejos, los más utilizados en la práctica. Explicamos el protocolo de bloqueo en dos fases (2PL) y algunas de sus variantes: 2PL básico, 2PL estricto, 2PL conservador y 2PL

riguroso. Las variantes estricta y rigurosa son más comunes debido a sus mejores propiedades de recuperabilidad. Hicimos una introducción de los conceptos de bloqueos compartidos (lectura) y exclusivos (escritura), y vimos cómo el bloqueo puede garantizar la serialización al utilizarse en combinación con la regla de bloqueo en dos fases. También presentamos varias técnicas para tratar con el problema del interbloqueo, que puede surgir con los bloqueos. En la práctica, es normal utilizar la detección del interbloqueo (gráficos de espera) y tiempos limitados.

Presentamos otros protocolos de control de la concurrencia que no se utilizan tan a menudo en la práctica, pero que son importantes como alternativas teóricas que muestran una solución para este problema. Entre ellos encontramos el protocolo de ordenación de marcas de tiempo, que garantiza la serialización basada en la ordenación de las marcas de tiempo de las transacciones. Las marcas de tiempo son identificadores de transacción únicos generados por el sistema. Explicamos la regla de escritura de Thomas, que mejora el rendimiento pero no garantiza la serialización por conflicto. También hemos visto el protocolo de ordenación de marcas de tiempo estricto. Explicamos dos protocolos multiversión, que asumen que en la base de datos se pueden guardar las versiones más antiguas de los elementos de datos. Una técnica, denominada bloqueo en dos fases multiversión (que se ha utilizado en la práctica), asume que pueden existir dos versiones de un elemento e intenta aumentar la concurrencia haciendo compatibles los bloqueos de escritura y lectura (a costa de introducir un modo de bloqueo de certificación adicional). También hemos presentado un protocolo multiversión basado en la ordenación de las marcas de tiempo, y un ejemplo de protocolo optimista, que también se conoce como protocolo de certificación o validación.

Después, centramos nuestra atención en la granularidad de los elementos de datos. Describimos un protocolo de bloqueo multigranularidad que permite modificar la granularidad (tamaño del elemento) en base a la mezcla de transacciones del momento, con el objetivo de mejorar el rendimiento del control de la concurrencia. Asimismo, hemos hablado del problema práctico que supone desarrollar protocolos de boqueo para los índices de modo que éstos no se conviertan en un estorbo para el acceso concurrente. Por último, hablamos del problema del fantasma y de los problemas relacionados con las transacciones interactivas, y describimos brevemente el concepto de cerrojos y de sus diferencias con los bloqueos.

El siguiente capítulo está dedicado a las técnicas de recuperación.

## Preguntas de repaso

**18.1.** ¿Qué es el protocolo de bloqueo en dos fases? ¿Cómo garantiza la serialización?

**18.2.** Detalle algunas de las variaciones del protocolo de bloqueo en dos fases. ¿Por qué es preferible utilizar casi siempre el bloqueo en dos fases estricto o riguroso?

**18.3.** Explique los problemas del interbloqueo y la espera indefinida, y los diferentes métodos de tratar con ellos.

**18.4.** Compare los bloqueos binarios con los bloqueos exclusivos/compartidos. ¿Por qué son preferibles estos últimos?

**18.5.** Describa los protocolos esperar-morir y herir-esperar para prevenir el interbloqueo.

**18.6.** Describa los protocolos de espera cautelosa, no espera y tiempo limitado para la prevención del interbloqueo.

**18.7.** ¿Qué es una marca de tiempo? ¿Cómo genera el sistema las marcas de tiempo?

**18.8.** Explique el protocolo de ordenación de marcas de tiempo para controlar la concurrencia. ¿En qué se diferencia la ordenación estricta de marcas de tiempo de la ordenación básica de marcas de tiempo?

**18.9.** Explique dos técnicas multiversión para controlar la concurrencia.

**18.10.** ¿Qué es un bloqueo de certificación? ¿Cuáles son las ventajas y los inconvenientes de utilizar los bloqueos de certificación?

**18.11.** ¿En qué se diferencian las técnicas de control de la concurrencia optimistas de otras técnicas de control de la concurrencia? ¿Por qué se llaman también técnicas de validación o certificación? Explique las fases típicas de un método optimista de control de la concurrencia.

**18.12.** ¿Cómo afecta la granularidad de los elementos de datos al rendimiento del control de la concurrencia? ¿Qué factores afectan a la selección del tamaño de granularidad para los elementos de datos?

**18.13.** ¿Qué tipo de bloqueos son necesarios para las operaciones de inserción y eliminación?

**18.14.** ¿Qué es el bloqueo de granularidad múltiple? ¿Bajo qué circunstancias se utiliza?

**18.15.** ¿Qué son los bloqueos de intención?

**18.16.** ¿Cuándo se utilizan los cerrojos?

**18.17.** ¿Qué es un registro fantasma? Explique el problema que un registro fantasma puede provocar en el control de la concurrencia.

**18.18.** ¿Cómo resuelve el bloqueo de índice el problema del fantasma?

**18.19.** ¿Qué es un bloqueo de predicado?

## Ejercicios

**18.20.** Demuestre que el protocolo de bloqueo en dos fases básico garantiza la serialización por conflicto de las planificaciones. (*Sugerencia:* Demuestre que, si un gráfico de serialización para una planificación tiene un ciclo, entonces al menos una de las transacciones participantes en la planificación no obedece el protocolo de bloqueo en dos fases).

**18.21.** Modifique las estructuras de datos para los bloqueos multimodo y los algoritmos para bloquear_lectura(*X*), bloquear_escritura(*X*) y desbloquear(*X*) de modo que sean posibles la degradación y la promoción de los bloqueos. (*Sugerencia:* El bloqueo necesita comprobar el o los id de transacción que conservan el bloqueo, si los hay).

**18.22.** Demuestre que el bloqueo en dos fases estricto garantiza las planificaciones estrictas.

**18.23.** Demuestre que los protocolos esperar-morir y herir-esperar evitan el interbloqueo y la inanición.

**18.24.** Demuestre que la espera cautelosa evita el interbloqueo.

**18.25.** Aplique el algoritmo de ordenación de marcas de tiempo a las planificaciones de la Figura 17.8(b) y (c), y determine si el algoritmo permitirá la ejecución de las planificaciones.

**18.26.** Repita el Ejercicio 18.25, pero utilice el método de ordenación de marcas de tiempo multiversión.

**18.27.** ¿Por qué el bloqueo en dos fases no se utiliza como método para controlar la concurrencia para índices como los árboles $B^+$?

**18.28.** La matriz de compatibilidad de la Figura 18.8 muestra que los bloqueos IS e IX son compatibles. Explique por qué es así.

**18.29.** El protocolo MGL dice que una transacción *T* puede desbloquear un nodo *N*, sólo si ninguno de los hijos del nodo *N* está bloqueado por la transacción *T*. Demuestra que, sin esta condición, el protocolo MGL sería incorrecto.

## Bibliografía seleccionada

El protocolo de bloqueo en dos fases y el concepto de bloqueos de predicado se propusieron por vez primera en Eswaran y otros (1976). Bernstein y otros (1987), Gray y Reuter (1993), y Papadimitriou (1986) se centran en el control de la concurrencia y la recuperabilidad. Kumar (1996) se centra en el rendimiento y los métodos de control de la concurrencia. El bloqueo se explica en Gray y otros (1975), Lien y Weinberger (1978), Kedem y Silbershatz (1980), y Korth (1983). Los interbloqueos y los gráficos de espera se formalizaron en Holt (1972), y los esquemas esperar-morir y herir-esperar se presentaron en Rosenkrantz y otros

(1978). La espera cautelosa se explicó en Hsu y otros (1992). Helal y otros (1993) compara varios métodos de bloqueo. Las técnicas de control de la concurrencia basadas en las marcas de tiempo se explican en Bernstein y Goodman (1980) y Reed (1983). El control optimista de la concurrencia se explica en Kung y Robinson (1981) y Bassiouni (1988). Papadimitriou y Kanellakis (1979) y Bernstein y Goodman (1983) explican las técnicas multiversión. La ordenación de marcas de tiempo multiversión se propuso en Reed (1978, 1983), y el bloqueo en dos fases multiversión se explica en Lai y Wilkinson (1984). En Gray y otros (1975) se propuso un método para el bloqueo con granularidades múltiples, y el efecto de bloquear las granularidades se analiza en Ries y Stonebraker (1977). Bhargava y Reidl (1988) presenta un método para seleccionar dinámicamente entre distintos métodos de control de la concurrencia y de recuperabilidad. Los métodos de control de la concurrencia para los índices se presentan en Lehman y Yao (1981) y en Shasha y Goodman (1988). En Srinivasan y Carey (1991) se presenta un estudio del rendimiento de varios algoritmos de control de la concurrencia con árboles $B^+$.

Otro trabajo reciente sobre el control de la concurrencia incluye el control de la misma basándose en la semántica (Badrinath y Ramamritham, 1992), los modelos de transacción para actividades de ejecución larga (Dayal y otros, 1991), y la administración de transacciones multinivel (Hasse y Weikum, 1991).

CAPÍTULO 19

# Técnicas de recuperación de bases de datos

En este capítulo explicamos algunas de las técnicas que podemos utilizar para que una base de datos se recupere ante los fallos. En la Sección 17.1.4 explicamos las diferentes causas de un fallo, como la caída del sistema y los errores de transacción. Además, en la Sección 17.2 vimos muchos de los conceptos que se utilizan para los procesos de recuperación, como el registro del sistema y los puntos de confirmación.

Empezamos en la Sección 19.1 con una descripción de un procedimiento de recuperación típico y una clasificación de los algoritmos de recuperación. Después explicamos varios conceptos de recuperación, como el registro antes de la escritura (*write-ahead logging*) y la actualización en el lugar (*in-place*) frente a las actualizaciones en la sombra, y el proceso de anulación (deshacer) del efecto de una transacción incompleta o fallida. En la Sección 19.2 presentamos las técnicas de recuperación basadas en la *actualización diferida*, también conocida como técnica NO-DESHACER/REHACER. En la Sección 19.3 explicamos las técnicas de recuperación basadas en la *actualización inmediata*, entre las que podemos citar los algoritmos DESHACER/REHACER y UNDO/NO-REDO. En la Sección 19.4 explicamos la técnica conocida como paginación en la sombra, que puede clasificarse como un algoritmo NO-DESHACER/NO-REHACER. En la Sección 19.5 presentamos un ejemplo práctico de esquema de recuperación DBMS, denominado ARIES. La recuperación en multibases de datos la explicamos brevemente en la Sección 19.6. Por último, en la Sección 19.7 explicamos las técnicas de recuperación ante fallos catastróficos.

Nuestro interés está en describir los distintos métodos de recuperación. Si desea descripciones de las características de recuperación en sistemas específicos, tendrá que consultar las notas bibliográficas y los manuales de usuario de dichos sistemas. Las técnicas de recuperación están a menudo entrelazadas con los mecanismos de control de la concurrencia. Ciertas técnicas de recuperación se utilizan mejor con determinados métodos de control de la concurrencia. Explicaremos los conceptos de recuperación independientemente de los mecanismos de control de la concurrencia, pero explicaremos las circunstancias bajo las que un mecanismo de recuperación en particular se utiliza mejor con un determinado protocolo de control de la concurrencia.

## 19.1 Conceptos de recuperación

### 19.1.1 Descripción de la recuperación y clasificación de los algoritmos de recuperación

Recuperarse del fallo de una transacción normalmente significa que la base de datos se *restaura* al estado coherente más reciente, inmediatamente anterior al momento del fallo. Para ello, el sistema debe guardar

información sobre los cambios que las distintas transacciones aplicaron a los elementos de datos. Esta información normalmente se guarda en el **registro del sistema**, como explicamos en la Sección 17.2.2. Una estrategia típica para recuperarse se puede resumir informalmente de este modo:

1. Si hay un daño extensivo a una amplia porción de la base de datos debido a un fallo catastrófico, como una caída del disco, el método de recuperación restaura una copia de seguridad antigua de la base de datos que normalmente se archiva en cinta y reconstruye un estado más actual volviendo a aplicar o *rehaciendo* las operaciones de las transacciones confirmadas a partir de la *copia de seguridad del registro*, hasta el momento del fallo.
2. Cuando la base de datos no está dañada físicamente, pero se ha vuelto inconsistente por fallos no catastróficos de los tipos 1 a 4 (consulte la Sección 17.1.4), la estrategia es invertir cualquier cambio que provocara la inconsistencia *deshaciendo* algunas operaciones. También puede ser necesario *rehacer* algunas operaciones a fin de restaurar un estado consistente de la base de datos, como veremos. En este caso, no necesitamos una copia archivada completa de la base de datos. En su lugar, durante la recuperación se consultan las entradas guardadas en el registro del sistema online.

Conceptualmente, podemos distinguir dos técnicas principales para recuperarse frente a fallos no catastróficos: actualización diferida y actualización inmediata. Las técnicas de **actualización diferida** no actualizan físicamente la base de datos en el disco hasta que la transacción haya alcanzado su punto de confirmación; es entonces cuando las actualizaciones se graban en la base de datos. Antes de alcanzar la confirmación, todas las actualizaciones de las transacciones se guardan en el espacio de trabajo de transacción local (o búferes). Durante la confirmación, las actualizaciones se guardan primero en el registro del sistema y, después, se escriben en la base de datos. Si una transacción falla antes de alcanzar su punto de confirmación, no habrá cambiado la base de datos en absoluto, por lo que no se necesita DESHACER. Puede ser necesario REHACER el efecto de las operaciones de una transacción confirmada a partir del registro, porque es posible que su efecto no se haya grabado todavía en la base de datos. Por tanto, la actualización diferida también se conoce como algoritmo NO-DESHACER/REHACER, técnica que explicamos en la Sección 19.2.

En las técnicas de **actualización inmediata**, la base de datos puede ser actualizada por algunas operaciones de una transacción *antes* de que esta última alcance su punto de confirmación. Sin embargo, esas operaciones se graban normalmente en el registro del sistema, *en disco*, *antes* de que se apliquen a la base de datos, lo que todavía hace posible la recuperación. Si una transacción falla después de grabar algunos cambios en la base de datos pero antes de alcanzar su punto de confirmación, el efecto de sus operaciones sobre la base de datos debe deshacerse; es decir, la transacción debe anularse. En el caso general de una actualización inmediata, durante la recuperación es posible tener que recurrir a *deshacer* y *rehacer*. Esta técnica, conocida como **algoritmo DESHACER/REHACER**, requiere las dos operaciones, y se utiliza con mucha frecuencia en la práctica. Una variante del algoritmo en la que todas las actualizaciones se graban en la base de datos antes de que la transacción se confirme sólo requiere la operación de *deshacer*, por lo que recibe el nombre de **algoritmo DESHACER/NO-REHACER**. En la Sección 19.3 explicamos estas técnicas.

### 19.1.2 Almacenamiento en caché (búfer) de los bloques de disco

El proceso de recuperación se entrelaza a menudo con funciones del sistema operativo (en particular, con el almacenamiento en caché o en búfer en la memoria principal de las páginas de disco). Normalmente, una o más de las páginas de disco que incluyen los elementos de datos que se van a actualizar se **almacenan en caché** en los búferes de la memoria principal, y después se actualizan en memoria antes de escribirse de nuevo en el disco. El almacenamiento en caché es tradicionalmente una función del sistema operativo, pero debido a su importancia en la eficacia de los procedimientos de recuperación, el DBMS se encarga de hacerlo llamando a rutinas de bajo nivel de los sistemas operativos.

En general, es conveniente considerar la recuperación en términos de páginas de disco de base de datos (bloques). Normalmente, se reserva una colección de búferes en memoria, denominados **caché DBMS**, bajo con-

trol del DBMS. Se utiliza un **directorio** para la caché a fin de rastrear los elementos de la base de datos que se encuentran en los búferes.[1] Puede ser una tabla con entradas de este tipo: <Dirección_página_disco, Ubicación_búfer>. Cuando el DBMS solicita una acción sobre algún elemento, primero comprueba el directorio de la caché para determinar si la página de disco que contiene ese elemento se encuentra en la caché. Si no es así, debe localizarse el elemento en el disco y copiar en la caché las páginas de disco apropiadas. Puede ser necesario **limpiar** algunos de los búferes de la caché para conseguir espacio para el elemento nuevo. Para seleccionar los búferes que se van a limpiar se puede utilizar alguna estrategia de sustitución de página propia de los sistemas operativos, como, por ejemplo, LRU (menos utilizado recientemente) o FIFO (primero en entrar, primero en salir), o alguna nueva estrategia específica del DBMS.

Asociado con cada búfer de la caché hay un **bit sucio**, que puede incluirse en la entrada de directorio, para indicar si se ha modificado o no el búfer. Cuando primero se lee una página de la base de datos en disco a un búfer de la caché, el directorio de caché se actualiza con la dirección de la nueva página de disco, y el bit sucio se establece a 0 (cero). En cuanto el búfer se modifica, el bit sucio de la entrada de directorio correspondiente se establece a 1 (uno). Cuando el contenido del búfer es reemplazado (limpiado) de la caché, el contenido primero debe escribirse en la página de disco correspondiente *sólo si su bit sucio es 1*. También necesitamos otro bit, denominado bit ***pin-unpin*** (se dice que a una página en caché se está accediendo actualmente [valor de bit a 1] si todavía no puede escribirse en el disco).

Al volcar a disco un búfer modificado podemos utilizar dos estrategias. La primera, conocida como **actualización en el lugar** (***in-place***), escribe el búfer en la *misma ubicación de disco original*, por lo que sobrescribe el valor antiguo de cualquier elemento de datos modificado en disco.[2] Por tanto, se conserva una sola copia de cada bloque de disco de la base de datos. La segunda estrategia, conocida como ***shadowing*** (en la sombra), escribe un búfer actualizado en una ubicación diferente del disco, por lo que pueden conservarse varias versiones de los elementos de datos.

En general, el valor antiguo de un elemento de datos antes de su actualización se denomina **BFIM** (imagen antes de la actualización, *before image*), y el valor nuevo después de la actualización **AFIM** (imagen después de la actualización, *after image*). En la técnica de *shadowing*, podemos conservar en disco las versiones BFIM y AFIM; por tanto, no es estrictamente necesario mantener un registro para la recuperación. En la Sección 19.4 explicaremos brevemente la recuperación basada en la técnica de *shadowing*.

### 19.1.3 Registro antes de la escritura, robar/no-robar y forzar/no-forzar

Cuando se utiliza la actualización en el lugar, es necesario utilizar un registro del sistema para la recuperación (consulte la Sección 17.2.2). En este caso, el mecanismo de recuperación debe garantizar la grabación de la BFIM del elemento de datos en la entrada apropiada del registro del sistema y que esa entrada se vuelque en disco antes de que la BFIM sea sobrescrita con la AFIM en la base de datos en disco. Este proceso se suele denominar **registro antes de la escritura**. Antes de poder describir un protocolo para el registro antes de la escritura, debemos distinguir entre dos tipos de información para un comando de escritura: la información necesaria para DESHACER y la información necesaria para REHACER. Una **entrada de registro del tipo REHACER** incluye el **valor nuevo** (AFIM) del elemento escrito por la operación, ya que lo necesitamos para *rehacer* el efecto de la operación a partir del registro del sistema (estableciendo el valor del elemento en la base de datos a su AFIM).

Las **entradas de registro del tipo DESHACER** incluyen el **valor antiguo** (BFIM) del elemento, ya que lo necesitamos para *deshacer* el efecto de la operación a partir del registro del sistema (estableciendo el valor del elemento en la base de datos de nuevo a su BFIM). En un algoritmo DESHACER/REHACER, se combi-

[1] Es algo parecido al concepto de tablas de páginas utilizado por el sistema operativo.

[2] La actualización en el lugar se utiliza en la práctica en la mayoría de los sistemas.

nan los dos tipos de entradas de registro. Además, cuando es posible una anulación en cascada, las entradas leer_elemento del registro del sistema se consideran como entradas del tipo DESHACER (consulte la Sección 19.1.5).

Como hemos mencionado, la caché del DBMS conserva bloques de disco de la base de datos, que no sólo incluyen *bloques de datos*, sino también *bloques de índice* y *bloques de registro* del disco. Al escribir un registro del registro del sistema, se almacena en el bloque de registro actual en la caché del DBMS. El registro del sistema es simplemente un fichero secuencial de disco (sólo de agregación) y la caché del DBMS puede contener varios bloques del registro del sistema (por ejemplo, los últimos *n* bloques de dicho registro) que se escribirán en disco. Cuando se realiza una actualización de un bloque de datos (almacenado en la caché del DBMS), una entrada del registro asociada se escribe en el último bloque de registro en la caché del DBMS. Con el método de registro antes de la escritura, los bloques de registro que contienen las entradas de registro asociadas a la actualización de un bloque de datos particular se escriben en disco antes de que el propio bloque de datos se pueda escribir de nuevo en el disco.

La terminología de recuperación DBMS estándar incluye los términos **robar/no-robar** y **forzar/no-forzar**, que especifican cuándo una página de la base de datos se puede escribir en disco desde la caché:

1. Si una página en caché actualizada por una transacción *no puede* escribirse en disco antes de que la transacción se confirme, se denomina **método no-robar**. El bit pin-unpin indica si una página no puede escribirse de nuevo en disco. En caso contrario, si el protocolo permite escribir un búfer actualizado *antes* de que la transacción se confirme, se denomina **robar**. El robo se utiliza cuando el gestor de caché del DBMS (búfer) necesita un marco búfer para otra transacción y el gestor de búferes reemplaza una página existente que se había actualizado pero cuya transacción no se había confirmado.
2. Si todas las páginas actualizadas por una transacción se escriben inmediatamente en disco cuando la transacción se confirma, se denomina **método forzar**. En caso contrario, se conoce como **no-forzar**.

El esquema de recuperación por actualización diferida de la Sección 19.2 obedece a un método *no-robar*. Sin embargo, los sistemas de bases de datos típicos emplean una estrategia *robar/no-forzar*. La ventaja de robar es que evita la necesidad de un espacio en búfer muy grande para almacenar en memoria todas las páginas actualizadas. La ventaja de no-forzar es que una página actualizada de una transacción confirmada todavía puede estar en el búfer cuando otra transacción necesita actualizarla, lo que elimina el coste de E/S que supone leer de nuevo la página del disco. Esto puede proporcionar un considerable ahorro de operaciones de E/S cuando una página específica se actualiza mucho por parte de varias transacciones.

Para permitir la recuperación cuando se utiliza la actualización en el lugar, las entradas adecuadas necesarias para la recuperación deben grabarse permanentemente en el registro del sistema en disco antes de que se apliquen los cambios a la base de datos. Por ejemplo, considere el siguiente protocolo **WAL** (**registro antes de la escritura**, *write-ahead logging*) para un algoritmo de recuperación que requiere DESHACER y REHACER:

1. La imagen anterior a la actualización de un elemento no puede ser sobrescrita por su imagen posterior a la actualización en la base de datos en disco hasta que todos los registros del registro del sistema del tipo DESHACER para la transacción de actualización (hasta este punto) se hayan escrito por la fuerza en el disco.
2. La operación de confirmación de una transacción no puede completarse hasta que todos los registros del registro del sistema del tipo REHACER y del tipo DESHACER para esa transacción se hayan escrito por la fuerza en disco.

Para facilitar el proceso de recuperación, es posible que el subsistema de recuperación del DBMS tenga que mantener varias listas relacionadas con las transacciones que se están procesando en el sistema. Entre ellas estaría una lista de **transacciones activas** que ya se han iniciado pero que todavía no se han confirmado, y puede que otras listas con todas las **transacciones confirmadas** y **abortadas** desde el último punto de control (consulte la siguiente sección). El mantenimiento de estas listas hace que el proceso de recuperación sea más eficaz.

### 19.1.4 Puntos de control en el registro del sistema y puntos de control difusos

Otro tipo de entrada en el registro es el denominado **punto de control**.[3] En el registro del sistema se escribe periódicamente un registro [checkpoint] (punto de control) en el punto en que el sistema escribe en la base de datos en disco todos los búferes del DBMS que se han modificado. En consecuencia, todas las transacciones que tienen sus entradas [commit, *T*] en el registro del sistema por delante de una entrada [checkpoint] no tienen que *rehacer* sus operaciones ESCRIBIR en caso de una caída del sistema porque todas sus actualizaciones se grabarán en la base de datos en disco durante el punto de control. El gestor de recuperaciones de un DBMS debe decidir a qué intervalos tomar un punto de control.

El intervalo puede medirse en tiempo (por ejemplo, cada *m* minutos) o como un número *t* de transacciones confirmadas desde el último punto de control (los valores de *m* o *t* son parámetros del sistema). La toma de un punto de control consiste en las siguientes acciones:

1. Suspender temporalmente la ejecución de las transacciones.
2. Forzar la escritura en disco de todos los búferes de memoria que se hayan modificado.
3. Escribir un registro [checkpoint] en el registro del sistema, y forzar la escritura del registro en disco.
4. Reanudar la ejecución de las transacciones.

Como consecuencia del paso 2, un registro de punto de control en el registro del sistema también puede incluir información adicional, como una lista con los identificadores de las transacciones activas y las ubicaciones (direcciones) de los registros primero y más reciente (último) del registro del sistema para cada transacción activa. Esto puede facilitar el proceso de deshacer las operaciones de las transacciones en el caso de que una transacción deba anularse.

El tiempo necesario para escribir todos los búferes de memoria modificados puede retrasar el procesamiento de una transacción debido al paso 1. Para reducir este retraso, se suele utilizar una técnica denominada de **puntos de control difusos**. En esta técnica, el sistema puede reanudar el procesamiento de transacciones después de haberse escrito el registro [checkpoint] en el registro del sistema sin tener que esperar a que termine el paso 2. Sin embargo, mientras no se complete el paso 2, el registro [checkpoint] seguirá siendo válido. Para lograr esto, el sistema mantiene un puntero a un punto de control válido, que continúa apuntando al registro [checkpoint] anterior del registro del sistema. Una vez concluido el paso 2, ese puntero cambia para apuntar al nuevo punto de control del registro del sistema.

### 19.1.5 Anulación de transacciones

Si una transacción falla por cualquier razón después de actualizar la base de datos, es posible tener que **anular** la transacción. Si la transacción ha modificado el valor de cualquier elemento de datos y lo ha escrito en la base de datos, debe restaurarse a su valor anterior (BFIM). Las entradas de registro de tipo deshacer se utilizan para restaurar los valores antiguos de los elementos de datos que deben anularse.

Si una transacción *T* es anulada, también debe anularse cualquier transacción *S* que, en el ínterin, lea el valor de algún elemento de datos *X* escrito por *T*. De forma parecida, como *S* se anula, cualquier transacción *R* que haya leído el valor de algún elemento de datos *Y* escrito por *S* también debe anularse; y así sucesivamente. Este fenómeno se conoce como **anulación en cascada**, y puede ocurrir cuando el protocolo de recuperación garantiza planificaciones *recuperables* pero no garantiza unas planificaciones *estrictas* o *libres de la anulación en cascada* (consulte la Sección 17.4.2). Con razón, la anulación en cascada puede ser muy compleja y

[3] El término *punto de control* se ha utilizado en algunos sistemas para describir situaciones más restrictivas, como en DB2. También se ha utilizado en la literatura para describir conceptos completamente diferentes.

**Figura 19.1.** Ilustración de la anulación en cascada (un proceso que nunca ocurre en las planificaciones estrictas o libres de la anulación en cascada). (a) Operaciones de lectura y escritura de tres transacciones. (b) Registro del sistema en el momento de la caída. (c) Operaciones antes de la caída.

**(a)**

| $T_1$ |
|---|
| leer_elemento($A$) |
| leer_elemento($D$) |
| escribir_elemento($D$) |

| $T_2$ |
|---|
| leer_elemento($B$) |
| escribir_elemento($B$) |
| leer_elemento($D$) |
| escribir_elemento($D$) |

| $T_3$ |
|---|
| leer_elemento($C$) |
| escribir_elemento($B$) |
| leer_elemento($A$) |
| escribir_elemento($A$) |

**(b)**

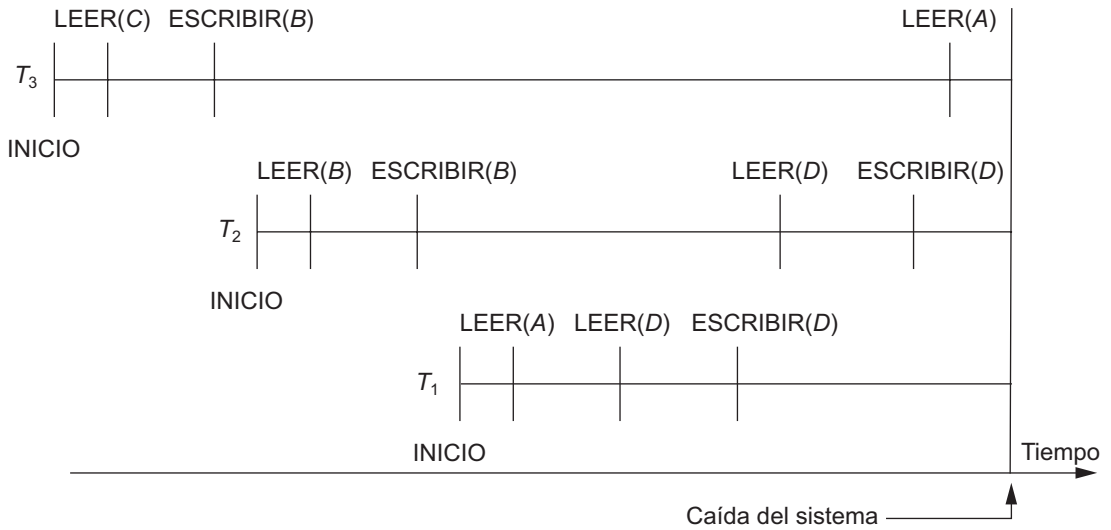
| | | *A* | *B* | *C* | *D* |
|---|---|---|---|---|---|
| | | 30 | 15 | 40 | 20 |
| | [iniciar transacción,$T_3$] | | | | |
| | [leer_elemento,$T_3$,$C$] | | | | |
| * | [escribir_elemento,$T_3$,$B$,15,12] | | 12 | | |
| | [iniciar transacción,$T_2$] | | | | |
| | [leer_elemento,$T_2$,$B$] | | | | |
| ** | [escribir_elemento,$T_2$,$B$,12,18] | | 18 | | |
| | [iniciar transacción,$T_1$] | | | | |
| | [leer_elemento,$T_1$,$A$] | | | | |
| | [leer_elemento,$T_1$,$D$] | | | | |
| | [escribir_elemento,$T_1$,$D$,20,25] | | | | 25 |
| | [leer_elemento,$T_2$,$D$] | | | | |
| ** | [escribir_elemento,$T_2$,$D$,25,26] | | | | 26 |
| | [leer_elemento,$T_3$,$A$] | | | | |

Caída del sistema

* $T_3$ se anula porque no alcanza su punto de confirmación.

** $T_2$ se anula porque lee el valor del elemento $B$ escrito por $T_3$.

**(c)**



consumir mucho tiempo. Por eso, casi todos los mecanismos de recuperación están diseñados de tal forma que nunca se necesite recurrir a la anulación en cascada.

La Figura 19.1 muestra un ejemplo que necesita la anulación en cascada. En la Figura 19.1(a) se muestran las operaciones de lectura y escritura de tres transacciones individuales. La Figura 19.1(b) muestra el registro del sistema en el momento de producirse una caída del sistema para una planificación concreta de la ejecución de esas transacciones. Los valores de los elementos de datos $A$, $B$, $C$ y $D$, que las transacciones utilizan, aparecen a la derecha de las entradas del registro del sistema. Asumimos que los valores originales de los elementos de datos, mostrados en la primera línea, son $A = 30$, $B = 15$, $C = 40$ y $D = 20$. En el momento del fallo del sistema, la transacción $T_3$ no ha alcanzado su conclusión y debe anularse. Las operaciones de escritura de $T_3$, marcadas con un solo asterisco en la Figura 19.1(b), son las operaciones de $T_3$ que se deshacen durante la anulación. La Figura 19.1(c) muestra gráficamente las operaciones de las diferentes transacciones a lo largo del eje temporal.

Ahora debemos comprobar la anulación en cascada. En la Figura 19.1(c) vemos que la transacción $T_2$ lee el valor del elemento $B$ que la transacción $T_3$ escribió; esto mismo puede determinarse examinando el registro del sistema. Como se anula la transacción $T_3$, $T_2$ también debe anularse ahora. Las operaciones de escritura de $T_2$, marcadas con ** en el registro del sistema, son las que se deshacen. Sólo las operaciones escribir_elemento deben deshacerse durante la anulación de la transacción; las operaciones leer_elemento se guardan en el registro del sistema sólo para determinar si es necesaria la anulación en cascada de otras transacciones.

En la práctica, la anulación en cascada de transacciones nunca es necesaria porque los métodos de recuperación prácticos garantizan *planificaciones libres de anulaciones en cascada* o *estrictas*. Por tanto, tampoco hay necesidad de grabar ninguna operación leer_elemento en el registro del sistema, porque esto sólo es necesario para determinar la anulación en cascada.

## 19.2 Técnicas de recuperación basadas en la actualización diferida

La idea que hay tras las técnicas de actualización diferida es diferir o posponer las actualizaciones de la base de datos hasta que la transacción complete su ejecución satisfactoriamente y alcance su punto de confirmación.[4]

Durante la ejecución de la transacción, las actualizaciones sólo se graban en el registro del sistema y en los búferes de la caché. Una vez que la transacción alcanza su punto de confirmación y se fuerza la escritura del registro del sistema en disco, las actualizaciones se graban en la base de datos. Si una transacción falla antes de alcanzar su punto de confirmación, no es necesario deshacer ninguna operación porque la transacción no ha modificado de forma alguna la base de datos en disco. Aunque esto puede simplificar la recuperación, no puede utilizarse en la práctica a menos que las transacciones sean cortas y cada transacción modifique pocos elementos. Para otros tipos de transacciones, hay posibilidad de agotar el espacio en búfer porque deben conservarse los cambios de la transacción en los búferes de la caché hasta alcanzarse el punto de confirmación.

Podemos enunciar un protocolo de actualización diferida típico de este modo:

1. Una transacción no puede modificar la base de datos en disco hasta haber alcanzado su punto de confirmación.
2. Una transacción no alcanza su punto de confirmación hasta que todas sus operaciones de actualización se han grabado en el registro del sistema *y* este último se ha escrito en el disco.

El paso 2 de este protocolo es una reafirmación del protocolo WAL (registro antes de la escritura). Como la base de datos nunca se actualiza en disco hasta que se confirma, nunca hay necesidad de deshacer operacio-

[4] Así, pues, la actualización diferida puede distinguirse generalmente como un *método no-robar*.

nes. Es lo que se conoce como **algoritmo de recuperación NO-DESHACER/REHACER**. La operación de rehacer sólo es necesaria si el sistema falla después de que la transacción se haya confirmado pero antes de que todos los cambios se hayan grabado en la base de datos en disco. En este caso, las operaciones de la transacción se rehacen a partir de las entradas del registro.

Normalmente, el método de recuperación ante fallos está estrechamente relacionado con el método de control de la concurrencia en los sistemas multiusuario. En primer lugar explicaremos la recuperación en los sistemas monousuario, en los que no se necesita controlar la concurrencia, de modo que pueda comprender el proceso de recuperación independientemente de cualquier método de control de la concurrencia. Después veremos cómo dicho control puede afectar al proceso de recuperación.

### 19.2.1 Recuperación mediante la actualización diferida en un entorno monousuario

En un entorno de este tipo, el algoritmo de recuperación puede ser bastante simple. El algoritmo RDU_S (Recuperación utilizando la actualización diferida en un entorno monousuario) utiliza un procedimiento REHACER, proporcionado con posterioridad, para rehacer determinadas operaciones escribir_elemento. Funciona del siguiente modo:

> **Procedimiento RDU_S.** Utiliza dos listas de transacciones: las transacciones confirmadas desde el último punto de control, y las transacciones activas (a lo sumo, sólo encajará una transacción en esta categoría porque el sistema es monousuario). Aplica la operación REHACER a todas las operaciones escribir_elemento de las transacciones confirmadas del registro del sistema en el orden en que se escribieron en el registro. Reinicia las transacciones activas.

El procedimiento REHACER se define como sigue:

> **Procedimiento REHACER (ESCRIBIR_OP).** Rehacer una operación escribir_elemento (ESCRIBIR_OP) consiste en examinar su entrada de registro [escribir_elemento, *T*, *X*, valor_nuevo] y establecer el valor del elemento *X* de la base de datos a valor_nuevo, que es la imagen después de la actualización (AFIM).

Es necesario que la operación REHACER sea ***idempotent***, es decir, ejecutarla una y otra vez es equivalente a ejecutarla una sola vez. De hecho, el proceso de recuperación entero debe ser *idempotent* porque si el sistema fallara durante el proceso de recuperación, el siguiente intento de recuperación podría REHACER determinadas operaciones escribir_elemento que ya habrían sido rehechas durante el primer proceso de recuperación. ¡El resultado de recuperarse frente a una caída del sistema *durante la recuperación* debe ser idéntico al resultado de recuperarse *si no hay ninguna caída durante la recuperación*!

Observe que la única transacción de la lista de transacciones activas no habrá tenido efecto en la base de datos debido al protocolo de actualización diferida, y es ignorada completamente por el proceso de recuperación porque ninguna de sus operaciones se reflejó en la base de datos en disco. Sin embargo, esta transacción debe restablecerse ahora, bien automáticamente por parte del proceso de recuperación, bien manualmente por parte del usuario.

La Figura 19.2 muestra un ejemplo de recuperación en un entorno monousuario, donde el primer fallo ocurre durante la ejecución de la transacción $T_2$, como se muestra en la Figura 19.2(b). El proceso de recuperación rehará la entrada [escribir_elemento, $T_1$, $D$, 20] del registro del sistema restableciendo el valor del elemento $D$ a 20 (su valor nuevo). Las entradas [escribir, $T_2$, ...] del registro del sistema son ignoradas por el proceso de recuperación porque $T_2$ no está confirmada. Si se produce un segundo fallo durante la recuperación de este primer fallo, se repite el mismo proceso de recuperación desde el principio hasta el final, con idénticos resultados.

**Figura 19.2.** Ejemplo de recuperación utilizando la actualización diferida en un entorno monousuario. (a) Operaciones de lectura y escritura de dos transacciones. (b) Registro del sistema en el momento de la caída.

**(a)**

| $T_1$ |
|---|
| leer_elemento($A$) |
| leer_elemento($D$) |
| escribir_elemento($D$) |

| $T_2$ |
|---|
| leer_elemento($B$) |
| escribir_elemento($B$) |
| leer_elemento($D$) |
| escribir_elemento($D$) |

**(b)**

| |
|---|
| [iniciar_transacción, $T_1$] |
| [escribir_elemento, $T_1$, $D$, 20] |
| [confirmar, $T_1$] |
| [iniciar_transacción, $T_2$] |
| [escribir_elemento, $T_2$, $B$, 10] |
| [escribir_elemento, $T_2$, $D$, 25] ← Caída del sistema |

Las operaciones [escribir_elemento...] de $T_1$ se rehacen.

Las entradas de registro $T_2$ son ignoradas por el proceso de recuperación.

## 19.2.2 Actualización diferida con ejecución concurrente en un entorno multiusuario

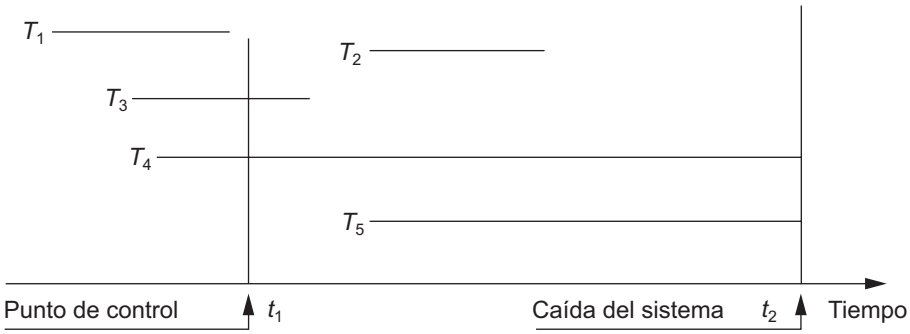
En los sistemas multiusuario con control de la concurrencia, el proceso de recuperación puede ser más complejo, en función de los protocolos utilizados para controlar la concurrencia. En muchos casos, los procesos de control de la concurrencia y de recuperación están interrelacionados. En general, cuanto mayor sea el grado de concurrencia que queramos lograr, más tiempo consumirá la tarea de recuperación.

Considere un sistema en el que el control de la concurrencia utiliza el bloqueo en dos fases estricto, por lo que los bloqueos sobre los elementos tienen efecto *hasta que la transacción alcanza su punto de confirmación*. Después de esto, los bloqueos pueden liberarse. Esto garantiza unas planificaciones estrictas y serializables. Asumiendo que las entradas [checkpoint] están incluidas en el registro del sistema, a continuación ofrecemos un posible algoritmo de recuperación para este caso, que podemos denominar RDU_M (Recuperación utilizando la actualización diferida en un entorno multiusuario). Este algoritmo utiliza el procedimiento REHACER definido anteriormente.

> **Procedimiento RDU_M (con puntos de control).** Utiliza dos listas de transacciones de las que el sistema se ocupa: las transacciones confirmadas $T$ desde el último punto de control (**lista de confirmación**), y las transacciones activas $T'$ (**lista activa**). Rehace todas las operaciones de escritura de las transacciones confirmadas a partir del registro del sistema, *en el orden en que fueron escritas en dicho registro*. Las transacciones que están activas y que no se confirmaron son efectivamente canceladas y deben reenviarse.

La Figura 19.3 muestra una posible planificación de la ejecución de las transacciones. Cuando se tomó el punto de control en el momento $t_1$, la transacción $T_1$ se había confirmado; no así las transacciones $T_3$ y $T_4$. Antes de que el sistema se cayera en el momento $t_2$, se confirmaron $T_3$ y $T_2$, pero no $T_4$ y $T_5$. Según el método RDU_M, no hay necesidad de rehacer las operaciones escribir_elemento de la transacción $T_1$ (o de cualquier transacción confirmada antes del último punto de control $t_1$). Sin embargo, las operaciones escribir_

**Figura 19.3.** Ejemplo de recuperación en un entorno multiusuario.




elemento de $T_2$ y $T_3$ deben rehacerse, ya que las dos alcanzaron sus puntos de confirmación después del último punto de control. Recuerde que el registro del sistema está obligado a escribirse antes de confirmarse una transacción. Las transacciones $T_4$ y $T_5$ son ignoradas: son canceladas o anuladas porque ninguna de sus operaciones escribir_elemento se grabó en la base de datos bajo el protocolo de actualización diferida. Nos referiremos más tarde a la Figura 19.3 para ilustrar otros protocolos de recuperación.

Podemos hacer que el algoritmo de recuperación NO-DESHACER/REHACER sea *más eficaz*: observe que, si las transacciones confirmadas desde el último punto de control han actualizado el elemento de datos $X$ (como se indica en las entradas del registro del sistema) más de una vez, sólo es necesario rehacer *la última actualización de X* a partir del registro durante la recuperación. Este último procedimiento REHACER sobrescribirá en cualquier caso las demás actualizaciones. En este caso, empezamos desde *el final del registro del sistema*; después, siempre que se rehace un elemento, se añade a una lista de elementos rehechos. Antes de aplicarse un procedimiento REHACER a un elemento, se comprueba la lista; si el elemento aparece en ella, no se rehace de nuevo, puesto que ya se ha recuperado su último valor.

Si una transacción aborta por cualquier razón (por ejemplo, debido a un método de detección de interbloqueos), simplemente se vuelve a enviar, puesto que no ha modificado la base de datos en disco. Un inconveniente del método aquí descrito es que limita la ejecución concurrente de transacciones porque *todos los elementos permanecen bloqueados hasta que la transacción alcanza su punto de confirmación*. Además, puede requerir demasiado espacio en búfer para conservar todos los elementos actualizados hasta que la transacción se confirma. La principal ventaja de este método es que las operaciones de la transacción *nunca tienen que deshacerse*, por dos razones:

1. Una transacción no graba ningún cambio en la base de datos en disco hasta después de alcanzar su punto de confirmación (es decir, hasta haber completado satisfactoriamente su ejecución). Por tanto, una transacción nunca se anula por un fallo durante su ejecución.
2. Una transacción nunca leerá el valor de un elemento escrito por una transacción que no ha sido confirmada, porque los elementos permanecen bloqueados hasta que la transacción alcanza su punto de confirmación. Por tanto, no se producirá la anulación en cascada.

La Figura 19.4 muestra un ejemplo de recuperación en un sistema multiusuario que utiliza el método de recuperación y control de la concurrencia que acabamos de describir.

### 19.2.3 Acciones de transacción que no afectan a la base de datos

En general, una transacción tendrá acciones que *no afectan* a la base de datos, como la generación e impresión de mensajes o informes a partir de la información recuperada de la base de datos. Si una transacción falla antes de completarse, no es recomendable que el usuario obtenga dichos informes, porque la transacción no se ha completado. Si se generan dichos informes erróneos, parte del proceso de recuperación tendría que infor-

**Figura 19.4.** Ejemplo de recuperación utilizando la actualización diferida con transacciones concurrentes. (a) Operaciones de lectura y escritura de cuatro transacciones. (b) Registro del sistema en el momento de la caída.

**(a)**

| $T_1$ | $T_2$ | $T_3$ | $T_4$ |
|---|---|---|---|
| leer_elemento($A$) | leer_elemento($B$) | leer_elemento($A$) | leer_elemento($B$) |
| leer_elemento($D$) | escribir_elemento($B$) | escribir_elemento($A$) | escribir_elemento($B$) |
| escribir_elemento($D$) | leer_elemento($D$) | leer_elemento($C$) | leer_elemento($A$) |
| | escribir_elemento($D$) | escribir_elemento($C$) | escribir_elemento($A$) |

**(b)**

| |
|---|
| [iniciar_transacción, $T_1$] |
| [escribir_elemento, $T_1$, $D$, 20] |
| [confirmar, $T_1$] |
| [punto_de_control] |
| [iniciar_transacción, $T_4$] |
| [escribir_elemento, $T_4$, $B$, 15] |
| [escribir_elemento, $T_4$, $A$, 20] |
| [confirmar, $T_4$] |
| [iniciar_transacción, $T_2$] |
| [escribir_elemento, $T_2$, $B$, 12] |
| [iniciar_transacción, $T_3$] |
| [escribir_elemento, $T_3$, $A$, 30] |
| [escribir_elemento, $T_2$, $D$, 25] ← Caída del sistema |

$T_2$ y $T_3$ son ignoradas porque no alcanzan sus puntos de confirmación.

$T_4$ se rehace porque su punto de confirmación está después del último punto de control del sistema.

mar al usuario de que esos informes son erróneos, puesto que el usuario puede tomar una decisión basada en ellos que afecte a la base de datos. Por tanto, dichos informes sólo deben generarse *una vez que la transacción ha alcanzado su punto de confirmación*. Un método común para tratar con dichas acciones consiste en emitir los comandos que generan los informes pero manteniéndolos como trabajos en lote, que se ejecutan una vez que la transacción ha alcanzado su punto de confirmación. Si la transacción falla, los trabajos en lote se cancelan.

## 19.3 Técnicas de recuperación basadas en la actualización inmediata

En estas técnicas, cuando una transacción emite un comando de actualización, la base de datos puede actualizarse *inmediatamente*, sin que haya que esperar a que la transacción alcance su punto de confirmación. No obstante, en estas técnicas una operación de actualización todavía tiene que grabarse en el registro del sistema (en disco) *antes* de que se aplique a la base de datos (utilizando el protocolo de registro antes de la escritura), de modo que podamos recuperarnos en caso de fallo.

Hay que prepararse para *deshacer* el efecto de las operaciones de actualización aplicadas a la base de datos por *transacciones fallidas*. Esto se consigue anulando la transacción y deshaciendo el efecto de sus operaciones escribir_elemento. En teoría, podemos distinguir dos categorías principales de algoritmos de actualización inmediata. Si la técnica de recuperación garantiza que todas las actualizaciones de una transacción se grabarán en la base de datos en disco *antes de que la transacción se confirme*, nunca habrá necesidad de REHACER ninguna operación de las transacciones confirmadas. Es lo que se conoce como **algoritmo de recuperación DESHACER/NO-REHACER**. Por otro lado, si la transacción tiene permitido confirmarse antes de que todos sus cambios se escriban en la base de datos, tenemos el caso más general, conocido como **algoritmo de recuperación DESHACER/REHACER**. También es la técnica más compleja. A continuación explicamos dos ejemplos de algoritmos DESHACER/REHACER y dejamos como ejercicio para el lector el desarrollo de una variante DESHACER/NO-REHACER. En la Sección 19.5 describimos un método más práctico conocido como técnica de recuperación ARIES.

### 19.3.1 Recuperación DESHACER/REHACER basada en la actualización inmediata en un entorno monousuario

En un sistema monousuario, si se produce un fallo, la transacción que se está ejecutando (activa) en el momento de producirse el fallo, puede haber grabado algunos cambios en la base de datos. El efecto de dichas operaciones debe deshacerse. El algoritmo de recuperación RIU_S (Recuperación utilizando la actualización inmediata en un entorno monousuario) utiliza el procedimiento REHACER definido anteriormente, así como el procedimiento DESHACER que definimos a continuación.

**Procedimiento RIU_S.**

1. Utiliza dos listas de transacciones que el sistema se encarga de mantener: las transacciones confirmadas desde el último punto de control y las transacciones activas (como máximo, sólo una transacción entrará en esta categoría porque el sistema es monousuario).
2. Deshace todas las operaciones escribir_elemento de la transacción *activa* a partir del registro del sistema, utilizando el procedimiento DESHACER descrito a continuación.
3. Rehace las operaciones escribir_elemento de las transacciones *confirmadas* a partir del registro del sistema, en el mismo orden en que se escribieron en dicho registro, utilizando el procedimiento REHACER descrito anteriormente.

El procedimiento DESHACER se define de este modo:

**Procedimiento DESHACER (ESCRIBIR_OP)**. Deshacer una operación escribir_elemento (escribir_op) consiste en examinar su entrada de registro [escribir_elemento, *T*, *X*, valor_antiguo, valor_nuevo] y establecer el valor del elemento *X* de la base de datos a valor_antiguo que es la imagen anterior a la actualización (BFIM). Para deshacer varias operaciones escribir_elemento de una o más transacciones a partir del registro del sistema hay que proceder en el *orden inverso* al orden en el que las operaciones se escribieron en el registro del sistema.

### 19.3.2 Recuperación DESHACER/REHACER basada en la actualización inmediata con ejecución concurrente

Cuando está permitida la ejecución concurrente, el proceso de recuperación depende una vez más de los protocolos utilizados para controlar la concurrencia. El procedimiento RIU_M (Recuperación utilizando la actualización inmediata para un entorno multiusuario) esboza un algoritmo de recuperación para las transacciones concurrentes con actualización inmediata. Asumiremos que el registro del sistema incluye puntos de control y que el protocolo de control de la concurrencia produce *planificaciones estrictas* (como, por ejemplo, el pro-

tocolo de bloqueo en dos fases estricto). Recuerde que una planificación estricta no permite que una transacción lea o escriba un elemento a menos que se haya confirmado (o abortado o anulado) la transacción que escribió ese elemento. Sin embargo, se pueden producir interbloqueos en el bloqueo en dos fases estricto, lo que requiere abortar y DESHACER transacciones. En el caso de una planificación estricta, DESHACER una operación requiere asignar al elemento su valor antiguo (BFIM).

**Procedimiento RIU_M.**

1. Utiliza dos listas de transacciones que el sistema se encarga de mantener: las transacciones confirmadas desde el último punto de control y las transacciones activas.
2. Deshace todas las operaciones escribir_elemento de las transacciones *activas* (no confirmadas), utilizando el procedimiento DESHACER. Las operaciones deben deshacerse en el orden inverso al orden en que se escribieron en el registro del sistema.
3. Rehace todas las operaciones escribir_elemento de las transacciones *confirmadas* a partir del registro del sistema, en el orden en que se escribieron en dicho registro.

Como explicamos en la Sección 19.2.2, el paso 3 es más eficaz empezando por el *final del registro del sistema* y rehaciendo únicamente *la última actualización de cada elemento X*. Siempre que se rehace un elemento, se añade a una lista de elementos rehechos y no se vuelve a rehacer de nuevo. Es posible idear un procedimiento parecido para mejorar la eficacia del paso 2.

## 19.4 Paginación en la sombra (*shadowing*)

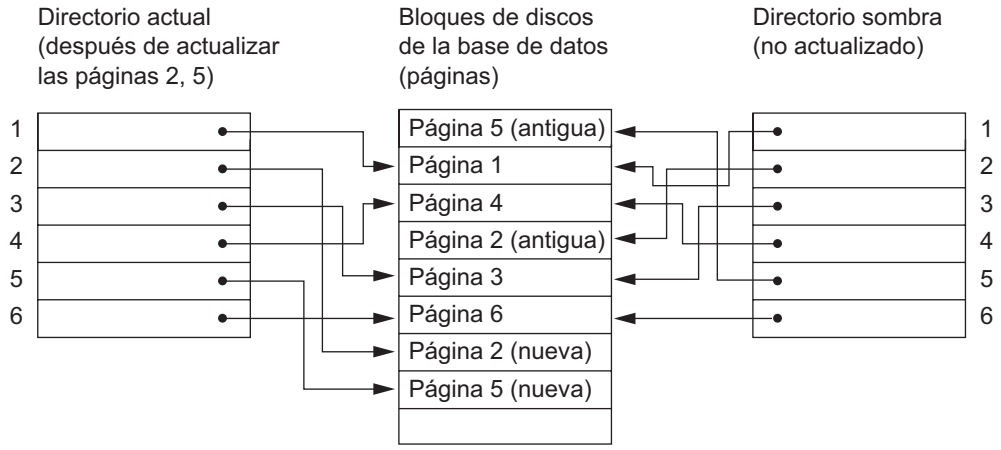
Este esquema de recuperación no requiere el uso de un registro del sistema en un entorno monousuario. En un entorno multiusuario, es posible que se necesite un registro del sistema para el método de control de la concurrencia. La paginación en la sombra considera que la base de datos está compuesta de un número de páginas de disco de tamaño fijo (o bloques de disco) (por ejemplo, *n*) para temas de recuperación. Se construye un **directorio** con *n* entradas[5], donde la entrada número *i* apunta a la página *i* de la base de datos en disco. El directorio se guarda en la memoria principal si no es demasiado grande, y todas las referencias (lecturas o escrituras) a las páginas de la base de datos en disco pasan por él. Cuando empieza la ejecución de una transacción, el **directorio actual** (cuyas entradas apuntan a las páginas más recientes o últimas de la base de datos en disco) se copia en un **directorio sombra**. El directorio sombra se guarda después en el disco, mientras que la transacción utiliza el directorio actual.

Durante la ejecución de la transacción, el directorio sombra *nunca* se modifica. Cuando se efectúa una operación escribir_elemento, se crea una nueva copia de la página de base de datos modificada, pero *no se sobrescribe* la copia antigua. En vez de ello, la nueva página se escribe en algún otro sitio (en algún bloque de disco que no se ha utilizado anteriormente). La entrada del directorio actual se modifica de modo que apunte al nuevo bloque de disco, mientras que el directorio sombra no se modifica y sigue apuntando al antiguo bloque de disco, no modificado. La Figura 19.5 ilustra los conceptos de directorio sombra y actual. Se guardan dos versiones de las páginas actualizadas por la transacción. La versión antigua es referenciada por el directorio sombra y la versión nueva por el directorio actual.

Para recuperarse ante un fallo durante la ejecución de una transacción, es suficiente con liberar las páginas modificadas de la base de datos y descartar el directorio actual. El estado de la base de datos anterior a la ejecución de la transacción está disponible a través del directorio sombra, y ese estado se recupera restableciendo el directorio sombra. La base de datos vuelve así a su estado anterior a la transacción que se estaba ejecutando cuando se produjo la caída, y se descarta cualquier página modificada. La confirmación de una transacción corresponde a descartar el directorio sombra anterior. Como la recuperación no implica deshacer

[5] El directorio es parecido a la tabla de páginas mantenida por el sistema operativo para cada proceso.

**Figura 19.5.** Ejemplo de paginación en la sombra.




ni rehacer elementos de datos, esta técnica se puede clasificar como técnica NO-DESHACER/NO-REHACER para la recuperación.

En un entorno multiusuario con transacciones concurrentes, es preciso incorporar los registros del sistema y los puntos de control a la técnica de paginación en la sombra. Un inconveniente de la paginación en la sombra es que las páginas actualizadas de la base de datos cambian de ubicación en el disco. Esto hace más difícil conseguir que las páginas relacionadas de la base de datos permanezcan juntas en el disco sin necesidad de recurrir a complejas estrategias de administración del almacenamiento. Además, si el directorio es grande, el coste de escribir directorios sombra en disco a medida que se confirman las transacciones es significativo. La manipulación de la **colección de basura** cuando una transacción se confirma es una complicación más. Las páginas antiguas referenciadas por el directorio sombra que se han actualizado deben liberarse y añadirse a una lista de páginas libres para su uso en el futuro. Estas páginas ya no se necesitan una vez confirmada la transacción. Otro problema es que la operación para migrar entre los directorios actual y sombra debe implementarse como una operación atómica.

# 19.5 Algoritmo de recuperación ARIES

Vamos a describir ahora el algoritmo ARIES como un ejemplo de algoritmo de recuperación que se utiliza en los sistemas de bases de datos. ARIES utiliza un método robar/no-forzar para escribir, y está basado en tres conceptos: registro antes de la escritura, repetición del histórico durante el procedimiento de rehacer, y registro de cambios durante el procedimiento deshacer. El registro antes de la escritura lo explicamos en la Sección 19.1.3. El segundo concepto, **repetición del histórico**, significa que ARIES volverá a trazar todas las acciones del sistema de bases de datos anteriores a la caída para reconstruir el estado de la base de datos *en el momento de producirse la caída*. Las transacciones que no estaban confirmadas al producirse la caída (transacciones activas) se deshacen. El tercer concepto, **registro durante el procedimiento deshacer**, impedirá que ARIES repita las operaciones de deshacer completas si durante la recuperación se produce un fallo, que provoca un reinicio del proceso de recuperación.

El procedimiento de recuperación ARIES consta de tres pasos principales: análisis, REHACER y DESHACER. El **análisis** identifica las páginas sucias (actualizadas) en el búfer[6] y el conjunto de transacciones activas en

[6] Los búferes actuales se pueden perder durante una caída, puesto que están en la memoria principal. Las tablas adicionales que se almacenan en el registro del sistema durante la creación de puntos de control (tabla de páginas sucias, tabla de transacciones) permiten a ARIES identificar esta información.

el momento de la caída. También se determina el punto correcto dentro del registro del sistema donde debe empezar la operación REHACER. La **fase REHACER** realmente vuelve a aplicar a la base de datos las actualizaciones del registro del sistema. Por regla general, la operación REHACER sólo se aplica a las transacciones confirmadas. Sin embargo, éste no es el caso de ARIES. Cierta información en el registro de ARIES proporcionará el punto inicial para REHACER, a partir del que se aplicarán las operaciones REHACER hasta que se alcance el final del registro. Además, la información almacenada por ARIES y en las páginas de datos permitirá a ARIES determinar si la operación que tiene que rehacerse se ha aplicado realmente a la base de datos y, por consiguiente, no es necesario que vuelva a aplicarse. De este modo, durante la recuperación *sólo se aplican las operaciones REHACER necesarias*. Por último, durante la **fase DESHACER** se explora el registro del sistema hacia atrás y se deshacen en orden inverso las operaciones de las transacciones que estaban activas en el momento de producirse la caída. La información necesaria para que ARIES lleve a cabo su procedimiento de recuperación incluye el registro, la tabla de transacciones y la tabla de páginas sucias. Además, se utilizan los puntos de control. Estas tablas son mantenidas por el gestor de transacciones y escritas en el registro durante la creación de puntos de control.

En ARIES, cada registro del registro del sistema tiene un **número de secuencia de registro (LSN)** asociado que se incrementa de uno en uno e indica la dirección del registro del registro del sistema en el disco. Cada LSN corresponde a un *cambio específico* (acción) de alguna transacción. Asimismo, cada página de datos almacenará el LSN de la *última entrada del registro del sistema correspondiente a un cambio para esa página*. En el registro del sistema se escribe un registro para cualquiera de las siguientes acciones: actualización de una página (escritura), confirmación de una transacción, aborto de una transacción, deshacer una actualización y finalizar una transacción. Ya hemos explicado la necesidad de incluir las tres primeras acciones en el registro, pero las dos últimas necesitan alguna explicación. Cuando se deshace una actualización, en el registro del sistema se escribe *un registro de compensación*. Cuando una transacción termina, bien porque se confirma, bien porque aborta, en el registro del sistema se escribe un *registro de finalización*.

Los campos comunes de todas las entradas del registro de sistema son los siguientes: el LSN anterior para esa transacción, el ID de la transacción, y el tipo de entrada del registro. El LSN anterior es importante porque enlaza las entradas del registro (en orden inverso) para cada transacción. En una acción de actualización (escritura), los campos adicionales que se incluyen en la entrada del registro son los siguientes: el ID de página de la página que incluye el elemento, la longitud del elemento actualizado, su desplazamiento desde el principio de la página y las imágenes anterior y posterior a la actualización del elemento.

Además del registro del sistema, se necesitan dos tablas para una recuperación eficaz: la **tabla de transacciones** y la **tabla de páginas sucias**, de cuyo mantenimiento se encarga el gestor de transacciones. Cuando se produce una caída, estas tablas se reconstruyen en la fase de análisis de la recuperación. La tabla de transacciones contiene una entrada *por cada transacción activa*, con información como el ID de la transacción, el estado de la transacción y el LSN de la entrada más reciente del registro del sistema para esa transacción. La tabla de páginas sucias contiene una entrada por cada página sucia del búfer, con el ID de página y el LSN correspondiente a la actualización más temprana de esa página.

**Los puntos de control** en ARIES constan de lo siguiente: escribir un registro inicio_punto_control en el registro del sistema, escribir un registro fin_punto_control en el registro del sistema, y escribir *el LSN* del registro inicio_punto_control en un fichero especial. A este fichero especial se accede durante la recuperación para localizar la información del último punto de control. Con el registro fin_punto_control, el contenido de la tabla de transacciones y de la tabla de páginas sucias se añade al final del registro del sistema. Para reducir el coste, se utilizan los **puntos de control difusos** para que el DBMS pueda seguir ejecutando transacciones durante la creación de puntos de control (consulte la Sección 19.1.4). Además, el contenido de la caché del DBMS no tiene que copiarse en disco durante el punto de control, ya que la tabla de transacciones y la tabla de páginas sucias (que se agregan al registro del sistema en disco) contienen la información necesaria para la recuperación. Si durante la creación del punto de control se produce una caída, el fichero especial se referirá al punto de control anterior, que se utiliza para la recuperación.

Después de una caída, el encargado es el gestor de recuperación de ARIES. En primer lugar se accede a la información del último punto de control a través del fichero especial. La **fase de análisis** empieza en el registro inicio_punto_control y procede hasta el final del registro del sistema. Cuando se encuentra el registro fin_punto_control, se accede a la tabla de transacciones y a la tabla de páginas sucias (recuerde que estas tablas se escribieron en el registro durante la creación del punto de control). Durante el análisis, los registros del registro del sistema que se están analizando pueden provocar modificaciones en estas dos tablas. Por ejemplo, si se encuentra un registro de finalización para una transacción $T$ en la tabla de transacciones, entonces la entrada correspondiente a $T$ se borra de esa tabla. Si se encuentra cualquier otro tipo de registro para una transacción $T'$, entonces se inserta una entrada para $T'$ en la tabla de transacciones, si no existe todavía, y se modifica el último campo LSN. Si el registro corresponde a un cambio para la página $P$, entonces se crearía una entrada para dicha página (si no existe todavía en la tabla) y se modificaría el campo LSN asociado. Cuando la fase de análisis está completa, la información necesaria para REHACER y DESHACER se ha compilado en las tablas.

A continuación va la **fase REHACER**. Para reducir la cantidad de trabajo innecesario, ARIES empieza a rehacer en un punto del registro donde sabe (con seguridad) que los cambios anteriores en las páginas sucias *ya se han aplicado en la base de datos en disco*. Esto lo puede determinar localizando el LSN más pequeño, $M$, de todas las páginas sucias de la tabla de páginas sucias, que indica la posición del registro donde ARIES tiene que empezar la fase REHACER. Cualquier cambio correspondiente a un LSN $< M$, para las transacciones que se pueden rehacer, ya se debe de haber propagado al disco o haberse sobrescrito en el búfer; en caso contrario, esas páginas sucias con ese LSN estarían en el búfer (y en la tabla de páginas sucias). Así, REHACER empieza en la entrada del registro con LSN $= M$ y explora hacia adelante hasta el final del registro del sistema. Para cada cambio grabado en el registro del sistema, el algoritmo REHACER verificaría si habría que volver a aplicar el cambio. Por ejemplo, si un cambio grabado en el registro pertenece a la página $P$ que no está en la tabla de páginas sucias, entonces ese cambio ya está en disco y no hay necesidad de volver a aplicarlo. O bien, si un cambio grabado en el registro (con LSN $= N$, por ejemplo) pertenece a la página $P$ y la tabla de páginas sucias contiene una entrada para $P$ con un LSN mayor que $N$, entonces el cambio ya está presente. Si no se cumple ninguna de estas dos condiciones, la página $P$ es leída desde el disco y el LSN almacenado en esa página, LSN($P$), es comparado con $N$. Si $N <$ LSN($P$), entonces el cambio ha sido aplicado y no hay que reescribir la página en el disco.

Una vez que ha finalizado la fase REHACER, la base de datos se encuentra en el estado exacto en el que estaba cuando se produjo la caída. El conjunto de transacciones activas (denominado **conjunto_deshacer**) quedó identificado en la tabla de transacciones durante la fase de análisis. Ahora, la **fase DESHACER** comienza la exploración hacia atrás desde el final del registro del sistema y deshace las acciones adecuadas. Por cada acción que se deshace se escribe un registro de compensación en el registro del sistema. El procedimiento DESHACER lee hacia atrás en el registro hasta que todas las acciones del conjunto de transacciones de conjunto_deshacer se han deshecho. Cuando esto se ha completado, se da por terminado el proceso de recuperación y puede empezar de nuevo el procesamiento normal.

Considere el ejemplo de recuperación de la Figura 19.6. Hay tres transacciones: $T_1$, $T_2$ y $T_3$. $T_1$ actualiza la página $C$, $T_2$ actualiza las páginas $B$ y $C$, y $T_3$ actualiza la página A. La Figura 19.6(a) muestra el contenido parcial del registro del sistema y (b) muestra el contenido de la tabla de transacciones y de la tabla de páginas sucias. Ahora, suponga que se produce una caída en este punto. Dado que ha tenido lugar un punto de control, se recupera la dirección del registro inicio_punto_control asociado, que es la ubicación 4. La fase de análisis comienza en la ubicación 4 hasta alcanzar el final. El registro fin_punto_control contendría la tabla de transacciones y la tabla de páginas sucias de la Figura 19.6(b), y la fase de análisis reconstruiría estas tablas. Cuando la fase de análisis encuentra el registro 6, en la tabla de transacciones se crea una nueva entrada para la transacción $T_3$ y en la tabla de páginas sucias se crea una entrada nueva para la página $A$. Una vez analizado el registro 8, el estado de la transacción $T_2$ cambia a confirmada en la tabla de transacciones. La Figura 19.6(c) muestra las dos tablas después de la fase de análisis.

**Figura 19.6.** Ejemplo de recuperación en ARIES. (a) El registro del sistema en el momento de la caída. (b) Las tablas de transacciones y de páginas sucias en el momento del punto de control. (c) Las tablas de transacciones y de páginas sucias después de la fase de análisis.

**(a)**

| Lsn | Último_lsn | Id_tran | Tipo | Id_página | Otra_información |
|---|---|---|---|---|---|
| 1 | 0 | $T_1$ | actualizada | *C* | . . . |
| 2 | 0 | $T_2$ | actualizada | *B* | . . . |
| 3 | 1 | $T_1$ | confirmada | | . . . |
| 4 | inicio_punto_control | | | | |
| 5 | fin_punto_control | | | | |
| 6 | 0 | $T_3$ | actualizada | *A* | . . . |
| 7 | 2 | $T_2$ | actualizada | *C* | . . . |
| 8 | 7 | $T_2$ | confirmada | | . . . |

**(b)**

TABLA DE TRANSACCIONES

| Id_transacción | Último_lsn | Estado |
|---|---|---|
| $T_1$ | 3 | confirmada |
| $T_2$ | 2 | en curso |

TABLA DE PÁGINAS SUCIAS

| Id_página | Lsn |
|---|---|
| *C* | 1 |
| *B* | 2 |

**(c)**

TABLA DE TRANSACCIONES

| Id_transacción | Último_lsn | Estado |
|---|---|---|
| $T_1$ | 3 | confirmada |
| $T_2$ | 8 | confirmada |
| $T_3$ | 6 | en curso |

TABLA DE PÁGINAS SUCIAS

| Id_página | Lsn |
|---|---|
| *C* | 1 |
| *B* | 2 |
| *A* | 6 |

Para la fase REHACER el LSN más pequeño en la tabla de páginas sucias es 1. Por tanto, REHACER empezará en el registro 1 y empezará a rehacer las actualizaciones. Los LSNs {1, 2, 6, 7} correspondientes a las actualizaciones de las páginas C, B, A y C, respectivamente, no son inferiores a los LSNs de esas páginas (como se muestra en la tabla de páginas sucias). Así que esas páginas de datos se leerán de nuevo y las actualizaciones volverán a aplicarse desde el registro del sistema (asumiendo que los LSNs actuales almacenados en esas páginas de datos son inferiores a la entrada de registro correspondiente). En este punto, la fase REHACER ha finalizado y empieza la fase DESHACER. A partir de la tabla de transacciones (véase la Figura 19.6[c]), DESHACER sólo se aplica a la transacción activa $T_3$. La fase DESHACER empieza en la entrada 6 del registro (la última actualización para $T_3$) y procede hacia atrás en el registro. La cadena regresiva de actualizaciones para la transacción $T_3$ (sólo el registro 6 del registro del sistema en este ejemplo) es seguida y se deshace.

## 19.6 Recuperación en sistemas multibase de datos

Hasta ahora hemos asumido implícitamente que una transacción accede a una sola base de datos. En algunos casos, es posible que una sola transacción, denominada **transacción multibase de datos**, tenga que acceder

a varias bases de datos. Incluso, estas bases de datos pueden estar almacenadas en diferentes tipos de DBMSs; por ejemplo, algunos DBMSs pueden ser relacionales, mientras otros pueden ser orientados a objetos, jerárquicos o de red. En tal caso, cada DBMS implicado en la transacción multibase de datos puede tener su propia técnica de recuperación y su propio gestor de transacciones separados de los de los demás DBMSs. Esta situación es algo parecida al caso de un sistema de administración de bases de datos distribuido (consulte el Capítulo 25), donde distintas partes de la base de datos residen en diferentes sitios que están conectados por una red de comunicación.

Para mantener la atomicidad de una transacción multibase de datos, es necesario tener un mecanismo de recuperación de dos niveles. Necesitamos un **gestor de recuperación global**, o **coordinador**, para conservar la información necesaria para la recuperación, además de los gestores de recuperación locales y la información que mantienen (registro del sistema, tablas). El coordinador normalmente obedece un protocolo denominado **protocolo de confirmación en dos fases**, que son las siguientes:

- **Fase 1.** Cuando todas las bases de datos participantes señalan al coordinador que la parte de la transacción multibase de datos que implica a cada una ha concluido, el coordinador envía un mensaje *preparado para la confirmación* a cada participante para que se prepare para la confirmación de la transacción. Cada base de datos participante que recibe este mensaje escribirá todos los registros del registro del sistema y la información necesarios para la recuperación local en el disco, y después enviará una señal *listo para la confirmación* u *OK* al coordinador. Si la escritura en disco falla o la transacción local no puede confirmarse por alguna razón, la base de datos participante envía una señal del tipo *no es posible la confirmación* o *no OK* al coordinador. Si este último no recibe una respuesta de la base de datos en un intervalo de tiempo dado, asume que la respuesta es *no OK*.
- **Fase 2.** Si *todas* las bases de datos participantes responden con un *OK*, y el voto del coordinador también es *OK*, la transacción es satisfactoria y el coordinador envía una señal de *confirmación* para la transacción a las bases de datos participantes. Como todos los efectos locales de la transacción y la información necesarios para la recuperación local se han grabado en los registros del sistema de las bases de datos participantes, la recuperación ante un fallo es ahora posible. Cada base de datos participante completa la confirmación de la transacción escribiendo una entrada [commit] para esa transacción en el registro del sistema y actualiza permanentemente la base de datos si es necesario. Por el contrario, si una o más de las bases de datos participantes o el coordinador tiene una respuesta *no OK*, la transacción ha fallado, y el coordinador envía un mensaje para *anular* o DESHACER el efecto local de la transacción a cada base de datos participante. Esto se hace deshaciendo las operaciones de la transacción, utilizando el registro del sistema.

El efecto neto del protocolo de confirmación en dos fases es que o todas las bases de datos participantes confirman el efecto de la transacción, o ninguna de ellas lo hace. En caso de que cualquiera de los participantes (o el coordinador) falle, siempre es posible volver a un estado en el que la transacción se confirma o se anula. Un fallo durante o antes de la fase 1 normalmente requiere que la transacción se anule, mientras que un fallo durante la fase 2 significa que una transacción satisfactoria puede recuperarse y confirmarse.

## 19.7 Copia de seguridad de la base de datos y recuperación ante fallos catastróficos

Hasta ahora, todas las técnicas que hemos explicado son aplicables a fallos no catastróficos. Una suposición importante ha sido que el registro del sistema se conserva en el disco y que no se pierde como consecuencia del fallo. De forma parecida, el directorio sombra debe almacenarse en el disco para permitir la recuperación cuando se utiliza la paginación en la sombra. Las técnicas de recuperación que hemos explicado utilizan las entradas del registro del sistema o del directorio sombra para recuperarse ante un fallo, llevando la base de datos de nuevo a un estado consistente.

El gestor de recuperación de un DBMS también debe estar equipado para encargarse de fallos más catastróficos, como la caída de los discos. La principal técnica que se utiliza para manipular dichas caídas es un copia de seguridad de la base de datos, en la que esta última y el registro se copian periódicamente en un medio de almacenamiento de bajo coste, como las cintas magnéticas. En caso de un fallo catastrófico del sistema, se puede volver a cargar la última copia de seguridad de la cinta al disco, de modo que el sistema se reinicia.

Los datos de aplicaciones críticas, como las que se utilizan en banca, aseguradoras, mercados de valores y otras bases de datos, se vuelcan periódicamente en copias de seguridad y se guardan en ubicaciones seguras físicamente separadas. Se han utilizado cajas fuertes subterráneas para proteger estos datos de inundaciones, tormentas, terremotos o incendios. Eventos recientes como el ataque terrorista del 11/S en Nueva York (2001) o el huracán Katrina en Nueva Orleans (2005) han creado una mayor conciencia sobre la *recuperación ante desastres de bases de datos críticas*.

Para evitar la pérdida de todos los efectos de las transacciones que se han ejecutado desde la última copia de seguridad, es costumbre hacer una copia de seguridad del registro del sistema a intervalos más frecuentes que la copia de seguridad de la base de datos completa, copiándolo periódicamente en la cinta magnética. Normalmente, el registro del sistema es considerablemente más pequeño que la propia base de datos y, por tanto, se puede hacer una copia de seguridad de él con más frecuencia. Por consiguiente, los usuarios no pierden todas las transacciones que han ejecutado desde la última copia de seguridad de la base de datos. Es posible rehacer el efecto sobre la base de datos de todas las transacciones confirmadas grabadas en la porción del registro del sistema que se ha copiado en la cinta. Después de cada copia de seguridad de la base de datos se inicia un nuevo registro del sistema. Por tanto, para recuperarse ante un fallo del disco, primero se vuelve a crear la base de datos en el disco a partir de su última copia de seguridad en cinta. A continuación, se reconstruyen los efectos de todas las transacciones confirmadas cuyas operaciones se han grabado en las copias de seguridad del registro del sistema.

## 19.8 Resumen

En este capítulo hemos explicado las técnicas de recuperación ante fallos en las transacciones. El objetivo principal de la recuperación es garantizar la propiedad de la atomicidad de una transacción. Si una transacción falla antes de completar su ejecución, el mecanismo de recuperación debe asegurarse de que la transacción no tiene efectos perdurables en la base de datos. Primero ofrecimos una explicación informal de un proceso de recuperación y, después, explicamos los conceptos del sistema relacionados con la recuperación: almacenamiento en caché, actualización en el lugar frente a actualización en la sombra, imágenes anterior y posterior a la actualización de un elemento de datos, operaciones de recuperación DESHACER frente a REHACER, políticas robar/no-robar y forzar/no-forzar, puntos de control del sistema, y protocolo de registro antes de la escritura.

A continuación, explicamos dos métodos de recuperación diferentes: actualización diferida y actualización inmediata. Las técnicas de actualización diferida posponen cualquier actualización real de la base de datos en disco hasta que la transacción alcanza su punto de confirmación. La transacción fuerza la escritura del registro del sistema en disco antes de grabar las actualizaciones en la base de datos. Este método, cuando se utiliza con determinados métodos de control de la concurrencia, nunca está diseñado para requerir la anulación de la transacción, y la recuperación consiste simplemente en rehacer las operaciones de las transacciones confirmadas después del último punto de control a partir del registro del sistema. El inconveniente es que puede necesitarse demasiado espacio en búfer, ya que las actualizaciones se conservan en los búferes y no se aplican en disco hasta que la transacción se confirma. La actualización diferida puede conducir a un algoritmo de recuperación conocido como NO-DESHACER/REHACER. Las técnicas de actualización inmediata pueden aplicar cambios en la base de datos en disco antes de que la transacción termine satisfactoriamente. Cualquier cambio aplicado a la base de datos debe grabarse primero en el registro del sistema y escribirse en disco para que estas operaciones puedan deshacerse en caso necesario. También hemos ofrecido una visión general de

un algoritmo de recuperación para actualización inmediata, conocido como DESHACER/REHACER. También se puede desarrollar otro algoritmo, DESHACER/NO-REHACER, para la recuperación inmediata en caso de que todas las acciones de una transacción se graben en la base de datos antes de confirmarse.

Además, hemos explicado la técnica de la paginación en la sombra para la recuperación, que hace un seguimiento de las páginas antiguas de la base de datos utilizando un directorio sombra. Esta técnica, que está clasificada como NO-DESHACER/NO-REHACER, no requiere un registro del sistema en los sistemas monousuario, pero sí en los sistemas multiusuario. También hemos presentado ARIES, un esquema de recuperación específico que se utiliza en muchos de los productos de bases de datos relacionales de IBM. Después vimos el protocolo de confirmación en dos fases, que se utiliza para recuperarse ante fallos que implican transacciones multibase de datos. Por último, explicamos la recuperación ante fallos catastróficos, que normalmente se lleva a cabo haciendo copias de seguridad en cinta de la base de datos y del registro del sistema. La copia de seguridad de este último se puede hacer más frecuentemente que la de la base de datos, y se puede utilizar para rehacer operaciones empezando por la última copia de seguridad de la base de datos.

## Preguntas de repaso

**19.1.** Explique los diferentes tipos de fallos en las transacciones. ¿Qué se entiende por fallo catastrófico?

**19.2.** Explique las acciones llevadas a cabo por las operaciones leer_elemento y escribir_elemento en una base de datos.

**19.3.** ¿Para qué se utiliza el registro del sistema? ¿Cuáles son las típicas clases de entradas en un registro del sistema? ¿Qué son los puntos de control, y por qué son importantes? ¿Qué son los puntos de confirmación, y por qué son importantes?

**19.4.** ¿Cómo utiliza el subsistema de recuperación las técnicas de almacenamiento en búfer y en caché?

**19.5.** ¿Qué son las imágenes anterior a la actualización (BFIM) y posterior a la actualización (AFIM) de un elemento de datos? ¿Cuál es la diferencia entre la actualización en el lugar y la actualización en la sombra, respecto a cómo manipulan la BFIM y la AFIM?

**19.6.** ¿Qué son las entradas de registro de tipo DESHACER y de tipo REHACER?

**19.7.** Describa el protocolo de registro antes de la escritura.

**19.8.** Identifique tres listas típicas de transacciones de cuyo mantenimiento se encargue el subsistema de recuperación.

**19.9.** ¿Qué significa la anulación de una transacción? ¿Qué se entiende por anulación en cascada? ¿Por qué los métodos de recuperación prácticos utilizan protocolos que no permiten la anulación en cascada? ¿Qué técnicas de recuperación no requieren ninguna anulación?

**19.10.** Explique las operaciones DESHACER y REHACER y las técnicas de recuperación utilizadas por cada una de ellas.

**19.11.** Explique la técnica de recuperación por actualización diferida. ¿Cuáles son sus ventajas e inconvenientes? ¿Por qué se la conoce como método NO-DESHACER/REHACER?

**19.12.** ¿Cómo manipula la recuperación las operaciones de transacción que no afectan a la base de datos, como la impresión de informes por parte de una transacción?

**19.13.** Explique la técnica de recuperación de actualización inmediata en entornos monousuario y multiusuario. ¿Cuáles son las ventajas y los inconvenientes de la actualización inmediata?

**19.14.** ¿Cuál es la diferencia entre los algoritmos DESHACER/REHACER y DESHACER/NO-REHACER para la recuperación con actualización inmediata? Desarrolle un boceto para un algoritmo DESHACER/NO-REHACER.

**19.15.** Describa la técnica de recuperación de paginación en la sombra. ¿Bajo qué circunstancias no se requiere un registro del sistema?

**19.16.** Describa las tres fases del método de recuperación ARIES.

**19.17.** ¿Cuáles son los números de secuencia de registro (LSNs) en ARIES? ¿Cómo se utilizan? ¿Qué información contienen las tablas de páginas sucias y de transacciones? Describa cómo se utilizan los puntos de control difusos en ARIES.

**19.18.** ¿Qué significan los términos robar/no-robar y forzar/no-forzar respecto a la administración de búferes por parte del procesamiento de transacciones?

**19.19.** Describa el protocolo de confirmación en dos fases para las transacciones multibase de datos.

**19.20.** Explique cómo se trata la recuperación ante fallos catastróficos.

## Ejercicios

**19.21.** Suponga que el sistema se cae antes de que la entrada [leer_elemento, $T_3$, $A$] se escriba en el registro de la Figura 19.1(b). ¿Esto representará alguna diferencia en el proceso de recuperación?

**19.22.** Suponga que el sistema se cae antes de que la entrada [escribir_elemento, $T_2$, $D$, 25, 26] se escriba en el registro del sistema de la Figura 19.1(b). ¿Esto representará alguna diferencia en el proceso de recuperación?

**19.23.** La Figura 19.7 muestra el registro del sistema correspondiente a una planificación particular para cuatro transacciones, $T_1$, $T_2$, $T_3$ y $T_4$, en el momento de una caída. Suponga que utilizamos el *protocolo de actualización inmediata* con puntos de control. Describa el proceso de recuperación ante la caída del sistema. Especifique las transacciones que se anulan, las operaciones del registro del sistema que se rehacen y cuáles (si las hay) se deshacen, y si tiene lugar alguna anulación en cascada.

**19.24.** Suponga que utilizamos el protocolo de actualización diferida para el ejemplo de la Figura 19.7. Muestre lo diferente que sería el registro del sistema en el caso de la actualización diferida por eliminación de las entradas innecesarias del registro; después, describa el proceso de recuperación utilizando su registro del sistema modificado. Asuma que sólo se han aplicado las operaciones REHACER, y especifique qué operaciones del registro se rehacen y cuáles se ignoran.

**19.25.** ¿En qué se diferencian los puntos de control en ARIES de los descritos en la Sección 19.1.4?

**19.26.** ¿Cómo utiliza ARIES los números de secuencia de registro para reducir la cantidad de trabajo REHACER necesario para la recuperación? Ilústrelo con un ejemplo utilizando la información de la Figura 19.6. Puede hacer sus propias suposiciones acerca de cuándo se escribe una página en disco.

**19.27.** ¿Qué implicaciones tendría una política de administración de búferes no-robar/forzar en la creación de puntos de control y la recuperación?

*Seleccione la respuesta correcta para cada una de las siguientes preguntas multiopción:*

**19.28.** El registro incremental con actualizaciones diferidas implica que el sistema de recuperación debe necesariamente:

- **a.** Almacenar en el registro del sistema el valor antiguo del elemento actualizado.
- **b.** Almacenar en el registro del sistema el valor nuevo del elemento actualizado.
- **c.** Almacenar en el registro del sistema tanto el valor antiguo como el valor nuevo del elemento actualizado.
- **d.** Almacenar en el registro del sistema sólo los registros de inicio de transacción y transacción confirmada.

**19.29.** El protocolo de registro antes de la escritura (WAL) simplemente significa que:

- **a.** La escritura de un elemento de datos debe realizarse antes que cualquier operación de registro.

**Figura 19.7.** Planificación de ejemplo y su registro correspondiente.

| |
|---|
| [iniciar_transacción, $T_1$] |
| [leer_elemento, $T_1$, $A$] |
| [leer_elemento, $T_1$, $D$] |
| [escribir_elemento, $T_1$, $D$, 20, 25] |
| [confirmar, $T_1$] |
| [punto_de_control] |
| [iniciar_transacción, $T_2$] |
| [leer_elemento, $T_2$, $B$] |
| [escribir_elemento, $T_2$, $B$, 12, 18] |
| [iniciar_transacción, $T_4$] |
| [leer_elemento, $T_4$, $D$] |
| [escribir_elemento, $T_4$, $D$, 25, 15] |
| [iniciar_transacción, $T_3$] |
| [escribir_elemento, $T_3$, $C$, 30, 40] |
| [leer_elemento, $T_4$, $A$] |
| [escribir_elemento, $T_4$, $A$, 30, 20] |
| [confirmar, $T_4$] |
| [leer_elemento, $T_2$, $D$] |
| [escribir_elemento, $T_2$, $D$, 15, 25] ← Caída del sistema |

**b.** El registro del registro del sistema para una operación debe escribirse antes de que se escriban los datos reales.

**c.** Todos los registros del registro del sistema deben escribirse antes de que empiece la ejecución de una nueva transacción.

**d.** Nunca es necesario escribir el registro del sistema en el disco.

**19.30.** En caso de un fallo de la transacción bajo un esquema de registro incremental de actualización diferida, ¿que necesitaríamos?

**a.** Una operación de deshacer.

**b.** Una operación de rehacer.

**c.** Una operación de deshacer y rehacer.

**d.** Ninguna de las anteriores.

**19.31.** Para el registro incremental con actualizaciones inmediatas, un registro del registro del sistema para una transacción contendría:

**a.** El nombre de la transacción, el nombre del elemento de datos, el valor antiguo del elemento y el valor nuevo del elemento.

**b.** El nombre de la transacción, el nombre del elemento de datos y el valor antiguo del elemento.

**c.** El nombre de la transacción, el nombre del elemento de datos y el valor nuevo del elemento.

**d.** El nombre de la transacción y el nombre del elemento de datos.

**19.32.** Para un correcto funcionamiento durante la recuperación, las operaciones de deshacer y rehacer deben ser:

**a.** Conmutativas.

**b.** Asociativas.

**c.** *Idempotent*.

**d.** Distributivas.

**19.33.** Cuando se produce un fallo, se consulta el registro del sistema y se deshace o rehace cada operación. Esto es un problema porque:

**a.** consultar el registro del sistema entero consume mucho tiempo.

**b.** Muchas operaciones rehacer son innecesarias.

**c.** (a) y (b).

**d.** Ninguna de las anteriores.

**19.34.** Al utilizar un registro del sistema basado en el esquema de recuperación, podría mejorarse el rendimiento así como proporcionarse un mecanismo de recuperación:

**a.** Escribiendo en disco los registros del registro del sistema al confirmarse cada transacción.

**b.** Escribiendo en disco los registros apropiados del registro del sistema durante la ejecución de la transacción.

**c.** Esperando a escribir los registros del registro del sistema hasta que se confirmen varias transacciones y se escriban como un lote.

**d.** Sin escribir en disco los registros del registro del sistema.

**19.35.** Hay una posibilidad de que se produzca una anulación en cascada cuando

**a.** Una transacción escribe elementos que sólo han sido escritos por una transacción confirmada.

**b.** Una transacción escribe un elemento que fue previamente escrito por una transacción sin confirmar.

**c.** Una transacción lee un elemento que fue previamente escrito por una transacción sin confirmar.

**d.** (b) y (c).

## Bibliografía seleccionada

Los libros Bernstein y otros (1987) y Papadimitriou (1986) están dedicados a la teoría y los principios del control de la concurrencia y la recuperación. El libro Gray y Reuter (1993) es un trabajo enciclopédico sobre el control de la concurrencia, la recuperación y otros problemas relacionados con el procesamiento de transacciones.

Verhofstad (1978) presenta un tutorial y un estudio sobre las técnicas de recuperación en los sistemas de bases de datos. La clasificación de los algoritmos basándose en sus características DESHACER/REHACER se explica en Haerder y Reuter (1983) y en Bernstein y otros (1983). Gray (1978) explica la recuperación, junto con otros aspectos del sistema sobre la implementación de sistemas operativos para las bases de datos. La técnica de la paginación en la sombra se explica en Lorie (1977), Verhofstad (1978) y Reuter (1980). Gray y otros (1981) explica el mecanismo de recuperación en SYSTEM R. Lockeman y Knutsen (1968), Davies (1972) y Bjork (1973) son antiguos ensayos que explican la recuperación. Chandy y otros (1975) explica la anulación de transacciones. Lilien y Bhargava (1985) explica el concepto de bloque de integridad y su uso para mejorar la eficacia de la recuperación.

La recuperación utilizando el registro antes de la escritura se analiza en Jhingran y Khedkar (1992) y se utiliza en el sistema ARIES (Mohan y otros, 1992a). Trabajos más recientes sobre la recuperación incluyen la compensación de las transacciones (Korth y otros, 1990) y la recuperación de la base de datos de la memoria principal (Kumar, 1991). Los algoritmos de recuperación ARIES (Mohan y otros, 1992) han tenido bastante éxito en la práctica. Franklin y otros (1992) explica la recuperación en el sistema EXODUS. Dos libros recientes, Kumar y Hsu (1998) y Kumar y Son (1998), explican la recuperación en detalle y contienen descripcio-

nes de los métodos de recuperación utilizados en distintos productos de bases de datos relacionales existentes.

PARTE 6

# Bases de datos de objetos y relacionales de objetos

CAPÍTULO 20

# Conceptos de las bases de datos de objetos

En este capítulo y en el siguiente explicamos los modelos de datos y los sistemas de bases de datos orientados a objetos.[1] Los modelos de datos y los sistemas tradicionales, como los relacionales, los de red y los jerárquicos, han tenido mucho éxito en el desarrollo de la tecnología de bases de datos, necesaria en muchas aplicaciones de bases de datos comerciales populares. Sin embargo, tienen sus inconvenientes cuando deben diseñarse e implementarse aplicaciones de bases de datos más complejas; por ejemplo, bases de datos para el diseño y la fabricación asistidos por computador (CAD/CAM y CIM[2]), experimentos científicos, telecomunicaciones, sistemas de información geográfica y multimedia.[3] Estas aplicaciones más modernas tienen unos requisitos y unas características que difieren de los de las aplicaciones comerciales tradicionales, como estructuras más complejas para los objetos, transacciones de mayor duración, nuevos tipos de datos para almacenar imágenes o elementos de texto más grandes, y la necesidad de definir operaciones específicas de la aplicación que no son estándar. Las bases de datos orientadas a objetos se propusieron para satisfacer las necesidades de estas aplicaciones más complejas. La metodología de orientación a objetos ofrece la flexibilidad de manipular algunos de los requisitos sin la limitación impuesta por los tipos de datos y los lenguajes de consulta disponibles en los sistemas de bases de datos tradicionales. Una característica fundamental de las bases de datos orientadas a objetos es la potencia que otorgan al diseñador para especificar tanto la *estructura* de los objetos complejos como las *operaciones* que pueden aplicarse a esos objetos.

Otra razón para la creación de bases de datos orientadas a objetos es el incremento del uso de lenguajes de programación orientados a objetos en el desarrollo de aplicaciones de software.

Las bases de datos son componentes fundamentales de muchos sistemas de software, y las bases de datos tradicionales son difíciles de utilizar con las aplicaciones orientadas a objetos que están desarrolladas con un lenguaje de programación orientado a objetos, como C++, Smalltalk o Java. Las bases de datos orientadas a objetos están diseñadas para que se integren directamente y sin problemas con las aplicaciones que están desarrolladas en dichos lenguajes.

---

[1] Estas bases de datos se denominan con frecuencia **bases de datos de objetos** y los sistemas se denominan **sistemas de administración de bases de datos de objetos** (**ODBMS**). Sin embargo, como este capítulo explica muchos conceptos generales sobre la orientación a objetos, utilizaremos el término *orientado a objetos* en lugar de únicamente *objetos*.

[2] Diseño asistido por computador/Fabricación asistida por computador y Fabricación integrada por computador.

[3] Las bases de datos multimedia deben almacenar varios tipos de objetos multimedia, como vídeo, audio, imágenes, gráficos y documentos (consulte el Capítulo 24).

La necesidad de características adicionales de modelado de datos también ha sido reconocida por los desarrolladores de DBMS, y las nuevas versiones de los sistemas relacionales están incorporando muchas de las características que se propusieron para las bases de datos orientadas a objetos. Esto ha llevado a sistemas que pueden clasificarse como *DBMSs relacionales de objetos o relacionales extendidos* (consulte el Capítulo 22). La última versión del estándar SQL para los DBMSs relacionales incluye algunas de estas características.

Aunque se han creado muchos prototipos experimentales y sistemas de bases de datos orientados a objetos comerciales, no se ha extendido su uso debido a la popularidad de los sistemas relacionales y relacionales de objetos. Entre los prototipos experimentales podemos citar el sistema ORION desarrollado en MCC,[4] OpenOODB de Texas Instruments, el sistema IRIS de los laboratorios Hewlett-Packard, el sistema ODE de AT&T Bell Labs,[5] y el proyecto ENCORE/ObServer de la Brown University. Los sistemas comerciales disponibles incluyen GEMSTONE/S Object Server de GemStone Systems, ONTOS DB de Ontos, Objectivity/DB de Objectivity Inc., Versant Object Database y FastObjects de Versant Corporation (y Poet), ObjectStore de Object Design, y Ardent Database de Ardent.[6] Todos estos productos representan sólo una lista parcial de los prototipos experimentales y de los sistemas de bases de datos orientados a objetos que se han creado.

En cuanto aparecieron los DBMSs orientados a objetos se reconoció la necesidad de un modelo y lenguaje estándares. Como el procedimiento formal de aprobación de estándares normalmente dura varios años, un consorcio formado por desarrolladores y usuarios de DBMSs orientados a objetos, denominado ODMG,[7] propuso un estándar que se conoce como estándar ODMG-93, que se ha ido revisando con el tiempo. En el Capítulo 21 describimos algunas de las características de este estándar.

Las bases de datos orientadas a objetos han adoptado muchos de los conceptos que se desarrollaron originalmente para los lenguajes de programación orientados a objetos.[8] En la Sección 20.1 examinamos los orígenes de la metodología de orientación a objetos y explicamos cómo se aplica a los sistemas de bases de datos. Después, en las Secciones 20.2 a 20.6, describimos los conceptos clave que se utilizan en muchos sistemas de bases de datos orientados a objetos. En la Sección 20.2 explicamos la *identidad objeto*, la *estructura de un objeto* y los *constructores de tipo*. En la Sección 20.3 presentamos los conceptos de *encapsulamiento de operaciones* y la definición de *métodos* como parte de las declaraciones de clase, y además explicamos los mecanismos para almacenar objetos en una base de datos haciéndolos *persistentes*. La Sección 20.4 describe las *jerarquías de tipos y clases* y la *herencia* en las bases de datos orientadas a objetos. La Sección 20.5 proporciona una visión general de los problemas que surgen cuando es preciso representar y almacenar *objetos complejos*. La Sección 20.6 explica conceptos adicionales, como el *polimorfismo*, la *sobrecarga del operador*, la *vinculación dinámica*, la *herencia múltiple y selectiva*, y el *versionado* y la *configuración de objetos*.

Este capítulo presenta los conceptos generales de las bases de datos orientadas a objetos, mientras que el Capítulo 21 presentará el estándar ODMG. El lector puede omitir las Secciones 20.5 y 20.6 de este capítulo si desea una introducción menos detallada de este tema.

# 20.1 Panorámica de los conceptos de orientación a objetos

Esta sección ofrece una visión general de la historia y los conceptos principales de las bases de datos orientadas a objetos (OODBs). Los conceptos OODB se explican más en detalle en las Secciones 20.2 a 20.6. El

[4] Microelectronics and Computer Technology Corporation, Austin, Texas.

[5] Ahora denominados Lucent Technologies.

[6] Anteriormente, O2 de O2 Technology.

[7] *Object Database Management Group* (Grupo de administración de bases de datos de objetos).

[8] También se desarrollaron conceptos parecidos en los campos del modelado semántico de datos y la representación del conocimiento.

término *orientación a objetos* (abreviado como *OO* u *O-O*) tiene sus orígenes en los lenguajes de programación OO, u OOPLs. Actualmente, los conceptos OO se aplican en las áreas de las bases de datos, la ingeniería del software, las bases de conocimiento, la inteligencia artificial y los sistemas de computación en general. Los OOPLs tienen sus raíces en el lenguaje SIMULA, que se propuso a finales de la década de 1960. En SIMULA, el concepto *clase* agrupa la estructura de datos interna de un objeto en una declaración de clase. Posteriormente, los investigadores propusieron el concepto de *tipos de datos abstractos*, que oculta las estructuras de datos internas y especifica todas las operaciones posibles que pueden aplicarse a un objeto, lo que derivó en el concepto de *encapsulamiento*. El lenguaje de programación Smalltalk, desarrollado en Xerox PARC[9] en la década de 1970, fue uno de los primeros lenguajes en incorporar explícitamente los conceptos de OO, como el envío de mensajes y la herencia. Se le conoce como lenguaje de programación OO *puro*, porque se diseñó explícitamente para que fuera orientado a objetos. Esto contrasta con los lenguajes de programación OO *híbridos*, que incorporan conceptos de OO dentro de un lenguaje ya existente. Un ejemplo de lenguaje híbrido es C++, que incorpora conceptos de OO en el conocido lenguaje de programación C.

Un **objeto** normalmente tiene dos componentes: un estado (valor) y un comportamiento (operaciones). Por tanto, tiene un cierto parecido con una *variable de programa* en un lenguaje de programación, sólo que tendrá normalmente una *estructura de datos compleja* y unas *operaciones específicas* definidas por el programador.[10] Los objetos en un OOPL sólo existen durante la ejecución del programa; por consiguiente, se denominan *objetos transitorios*. Una base de datos OO puede extender la existencia de los objetos guardándolos permanentemente, de modo que los objetos *persisten* más allá de la terminación del programa y pueden recuperarse y compartirse con posterioridad con otros programas. En otras palabras, las bases de datos OO almacenan *objetos persistentes* permanentemente en almacenamiento secundario, y permiten que se puedan compartir entre varios programas y aplicaciones. Esto requiere la incorporación de otras características bien conocidas de los sistemas de administración de bases de datos, como los mecanismos de indexación, el control de la concurrencia y la recuperación. Un sistema de bases de datos OO interactúa con uno o más lenguajes de programación OO a fin de proporcionar la funcionalidad de objeto persistente y compartido.

Un objetivo de las bases de datos OO es mantener una correspondencia directa entre los objetos del mundo real y de la base de datos, para que los objetos no pierdan su integridad e identidad, puedan identificarse fácilmente y pueda trabajarse con ellos. Por tanto, las bases de datos OO proporcionan un **identificador de objeto** (OID, *object identifier*) generado por el sistema para cada objeto. Podemos comparar esto con el modelo relacional, en el cual cada relación debe tener un atributo de clave principal cuyo valor identifique de forma única a cada tupla. En el modelo relacional, si el valor de la clave principal cambia, la tupla tendrá una identidad nueva, aunque todavía podría representar el mismo objeto del mundo real. De forma alternativa, un objeto del mundo real puede tener diferentes nombres para los atributos clave en diferentes relaciones, lo que complica la verificación de que las claves representan el mismo objeto (por ejemplo, el identificador de objeto puede representarse como IdEmp en una relación y como Dni en otra).

Otra característica de las bases de datos OO es que los objetos pueden tener una *estructura de objeto* de *complejidad arbitraria* a fin de contener toda la información necesaria que describe el objeto. En contraste, en los sistemas de bases de datos tradicionales, la información sobre un objeto complejo a menudo se *dispersa* por muchas relaciones o registros, lo que lleva a una pérdida de correspondencia directa entre un objeto del mundo real y su representación en la base de datos. La estructura interna de un objeto en los OOPLs incluye la especificación de **variables de instancia**, que almacenan los valores que definen el estado interno del objeto. Por tanto, una variable de instancia se parece al concepto de *atributo* en el modelo relacional, excepto que las variables de instancia pueden encapsularse dentro del objeto y, por tanto, no son necesariamente visibles a los usuarios externos. Las variables de instancia también pueden ser de tipos de datos arbitrariamente complejos. Los sistemas orientados a objetos permiten la definición de las operaciones o funciones (comportamiento) que pueden aplicarse a los objetos de un tipo concreto. De hecho, algunos modelos OO insisten en que todas las

[9] Palo Alto Research Center, Palo Alto, California.

[10] Los objetos tienen muchas otras características, como explicamos en el resto de este capítulo.

operaciones que un usuario puede aplicar a un objeto deben predefinirse. Esto obliga a un *encapsulamiento completo* de los objetos. Esta rígida metodología se ha relajado en la mayoría de los modelos de datos OO por varias razones. En primer lugar, el usuario de una base de datos a menudo necesita conocer los nombres de atributo a fin de poder especificar las condiciones de selección en los atributos para recuperar objetos específicos. En segundo lugar, un encapsulamiento completo implica que cualquier recuperación sencilla requiere una operación predefinida, lo que dificulta la especificación de consultas específicas sobre la marcha.

Para el encapsulamiento, una operación se define en dos partes. La primera parte, denominada *firma (signatura)* o *interfaz* de la operación, especifica el nombre de la operación y los argumentos (o parámetros). La segunda parte, denominada *método* o *cuerpo*, especifica la *implementación* de la operación. Las operaciones se pueden invocar pasando un *mensaje* a un objeto, que incluye el nombre de la operación y los parámetros. El objeto ejecuta después el método para esa operación. Este encapsulamiento permite modificar la estructura interna de un objeto, así como la implementación de sus operaciones, sin la necesidad de trastocar los programas externos que invocan esas operaciones. Por tanto, el encapsulamiento proporciona una forma de independencia de datos y operación (consulte el Capítulo 2).

Otros conceptos importantes en los sistemas OO son la *herencia* y las jerarquías de tipos y clases. Esto permite especificar nuevos tipos o clases que heredan gran parte de su estructura y/u operaciones de los tipos o clases anteriormente definidos. Por tanto, la especificación de tipos de objetos puede proceder sistemáticamente. Esto facilita el desarrollo incremental de tipos de datos de un sistema, y la *reutilización* de definiciones de tipo existentes a la hora de crear nuevos tipos de objetos.

Un problema de los primeros sistemas de bases de datos OO era la representación de las *relaciones* entre los objetos. La insistencia de un encapsulamiento completo en los primeros modelos de datos OO llevó al argumento de que las relaciones no debían representarse explícitamente, sino que debían describirse definiendo los métodos apropiados que localizaran los objetos relacionados. No obstante, este método no funciona muy bien en las bases de datos complejas con muchas relaciones, porque es útil identificar esas relaciones y hacerlas visibles a los usuarios. El estándar ODMG ha reconocido esta necesidad y representa explícitamente las relaciones binarias a través de un par de *referencias inversas* (es decir, colocando los OIDs de los objetos relacionados dentro de los propios objetos, y manteniendo la integridad referencial, como veremos en el Capítulo 21).

Algunos sistemas OO ofrecen la posibilidad de tratar con *varias versiones* del mismo objeto (una característica esencial en el diseño y la ingeniería de aplicaciones). Por ejemplo, una versión antigua de un objeto que representa un diseño probado y verificado debe conservarse hasta que el objeto nuevo haya sido probado y verificado. Una nueva versión de un objeto complejo puede incluir sólo unas cuantas versiones nuevas de sus objetos constituyentes, mientras que otros componentes permanecen igual. Además de permitir el versionado, las bases de datos OO también deben permitir la evolución del esquema, lo que ocurre cuando las declaraciones de tipo cambian o cuando se crean tipos o relaciones nuevos. Estas dos características no son específicas de las bases de datos OO y sería recomendable que se incluyeran en todos los tipos de DBMSs.[11]

Otro concepto OO es la *sobrecarga del operador*, que se refiere a la posibilidad de que una operación pueda aplicarse a diferentes tipos de objetos; en una situación semejante, un *nombre de operación* puede referirse a varias *implementaciones* distintas, en función del tipo de objetos a los que se aplique. Esta característica también se conoce como *polimorfismo del operador*. Por ejemplo, una operación para calcular el área de un objeto geométrico puede diferir en su método (implementación), dependiendo de si el objeto es de tipo triángulo, círculo o rectángulo. Esto puede requerir el uso de una *vinculación posterior* del nombre de la operación al método apropiado en tiempo de ejecución, cuando se conoce el tipo de objeto al que se aplica la operación.

Esta sección ofrece una visión general de los principales conceptos de las bases de datos OO. En las Secciones 20.2 a 20.6 explicamos estos conceptos más en profundidad.

---

[11] En el estándar SQL relacional (consulte la Sección 8.3) ya definimos varias operaciones de evolución del esquema, como ALTER TABLE.

# 20.2 Identidad del objeto, estructura del objeto y constructores de tipos

En esta sección explicamos en primer lugar el concepto de identidad de objeto, y después presentamos las operaciones de estructuración típicas para definir la estructura del estado de un objeto. Dichas operaciones se conocen a menudo como **constructores de tipos**. Definen las operaciones de estructuración básica de los datos que pueden combinarse para formar estructuras de objetos complejas.

## 20.2.1 Identidad del objeto

Un sistema de bases de datos OO proporciona una **identidad única** a cada objeto independiente almacenado en la base de datos. Esta identidad única suele implementarse mediante un **identificador de objeto** único, generado por el sistema, u OID. El valor de un OID no es visible para el usuario externo, sino que el sistema lo utiliza a nivel interno para identificar cada objeto de manera única, y para crear y administrar las referencias entre objetos. Si es necesario, el OID puede asignarse a las variables de programa del tipo apropiado.

La principal propiedad que debe tener un OID es la de ser **inmutable**, es decir, el valor de éste para un objeto particular no cambia. Esto preserva la identidad del objeto del mundo real que se está representando. Por tanto, un sistema de bases de datos OO debe disponer de algún mecanismo de generación de OIDs y conservación de la propiedad de inmutabilidad. También es preferible que cada OID se utilice sólo una vez; esto es, aunque un objeto se elimine de la base de datos, su OID no se deberá asignar a otro objeto. Estas dos propiedades implican que el OID no debe depender de cualquier valor de atributo del objeto, puesto que el valor de un atributo puede cambiar o corregirse. Igualmente, suele considerarse inapropiado basar este identificador en el valor de algún atributo o en la dirección física del objeto en el almacenamiento, puesto que esta dirección puede cambiar después de una reorganización física de la base de datos. No obstante, algunos sistemas emplean esta técnica para aumentar la eficacia de la recuperación de objetos. Si la dirección física de un objeto cambia, en la dirección antigua puede colocarse un *puntero indirecto* que proporcione la nueva ubicación física del objeto. Es más normal utilizar enteros largos como OIDs y después utilizar alguna forma de tabla de dispersión para asignar el valor de OID a la dirección física actual del objeto en el almacenamiento.

Algunos de los primeros modelos de datos OO requerían que todo (desde un simple valor hasta un objeto complejo) se representara como un objeto; por tanto, todo valor básico, como un entero, una cadena o un valor booleano, tiene un OID. Esto permite que dos valores básicos tengan OIDs diferentes, lo que en algunos casos puede resultar de utilidad. Por ejemplo, el valor entero 50 puede utilizarse para que unas veces represente un peso en kilogramos y otras para que represente la edad de una persona. Después, pueden crearse dos objetos básicos con OIDs distintos, pero estos dos objetos representarían el valor entero 50. Aunque resulta útil como modelo teórico, no es muy práctico, porque puede llevar a la generación de demasiados OIDs. Por tanto, la mayoría de los sistemas de bases de datos OO permiten la representación tanto de objetos como de **valores**. Cada objeto debe tener un OID inmutable, mientras que un valor no tiene OID y se presenta como tal. Así, un valor normalmente se almacena dentro de un objeto y otros objetos *no pueden referirse a él*. En algunos sistemas también es posible crear valores con estructura compleja sin que se necesite el OID correspondiente.

## 20.2.2 Estructura del objeto

En las bases de datos OO, el estado (valor actual) de un objeto complejo puede construirse a partir de otros objetos (u otros valores) utilizando ciertos **constructores de tipos**. Una forma de representar dichos objetos es ver cada objeto como un trío ($i$, $c$, $v$), donde $i$ es el *identificador de objeto* (OID) único, $c$ es un *constructor de tipo*[12] (es decir, una identificación de cómo se construye el estado del objeto) y $v$ es el estado del

[12] Es diferente de la operación constructor que se utiliza en C++ y otros OOPLs para crear objetos nuevos.

objeto (o *valor actual*). El modelo de datos normalmente incluirá varios constructores de tipos. Los tres constructores más básicos son **atom**, **tuple** y **set**. Otros constructores que también se utilizan mucho son **list, bag** y **array**. El constructor atom se utiliza para representar todos los valores atómicos básicos, como enteros, números reales, cadenas de caracteres, booleanos y cualquier tipo de datos básico que el sistema soporte directamente.

El estado $v$ de un objeto ($i$, $c$, $v$) se interpreta basándose en el constructor $c$. Si $c$ = atom, el estado (valor) $v$ es un valor atómico del dominio de valores básicos soportado por el sistema. Si $c$ = set, el estado $v$ es un *conjunto de identificadores de objetos* $\{i_1, i_2, ..., i_n\}$, que son los OIDs de un conjunto de objetos que normalmente son de algún tipo. Si $c$ = tuple, el estado $v$ es una tupla de la forma $<a_1:i_1, a_2:i_2, ..., a_n:i_n>$, donde cada $a_j$ es un nombre de atributo[13] y cada $i_j$ es un OID. Si $c$ = list, el valor $v$ es una *lista ordenada* $[i_1, i_2, ..., i_n]$ de OIDs de objetos del mismo tipo. Una lista se parece a un conjunto, excepto que los OIDs de una lista están *ordenados* y, por tanto, podemos referirnos al primer, segundo o $j$-ésimo objeto de una lista. Para $c$ = array, el estado del objeto es un array unidimensional de identificadores de objetos. La principal diferencia entre un array y una lista es que esta última puede tener una cantidad arbitraria de elementos, mientras que un array normalmente tiene un tamaño máximo. La diferencia entre *set* y *bag*[14] es que todos los elementos de un conjunto deben ser únicos, mientras que una bolsa puede tener elementos duplicados.

Este modelo de objetos permite el anidamiento arbitrario de conjuntos, listas, tuplas y otros constructores. El estado de un objeto que no es de tipo atómico se referirá a otros objetos por sus identificadores de objeto. Por tanto, el único caso donde aparece el valor real es en *el estado de un objeto de tipo atómico*.[15]

Los constructores de tipo set, list, array y bag se denominan **tipos colección** (o **tipos *bulk***), para distinguirlos de los tipos básicos y de los tipos de tupla. La principal característica de un tipo colección es que el estado del objeto será una *colección de objetos* que pueden estar desordenados (como un conjunto o una bolsa) u ordenados (como una lista o un array). El tipo de constructor **tupla** se denomina a veces **tipo estructurado**, puesto que es equivalente a la construcción **struct** de los lenguajes de programación C y C++.

**Ejemplo 1: Un objeto complejo.** Vamos a representar algunos objetos de la base de datos relacional de la Figura 5.6, utilizando el modelo anterior, donde un objeto queda definido mediante un trío (OID, constructor de tipo, estado) y los constructores de tipo disponibles son atom, set y tuple. Utilizamos $i_1$, $i_2$, $i_3$, ... para representar los identificadores de objeto únicos generados por el sistema. Vamos a considerar estos objetos:

$o_1$ = ($i_1$, atom, 'Madrid')
$o_2$ = ($i_2$, atom, 'Valencia')
$o_3$ = ($i_3$, atom, 'Sevilla')
$o_4$ = ($i_4$, atom, 5)
$o_5$ = ($i_5$, atom, 'Investigación')
$o_6$ = ($i_6$, atom, '22-05-1988')
$o_7$ = ($i_7$, set, $\{i_1, i_2, i_3\}$)
$o_8$ = ($i_8$, tuple, <NombreDpto:$i_5$, NúmeroDpto:$i_4$, Dire:$i_9$, Ubicaciones:$i_7$, Empleados:$i_{10}$, Proyectos:$i_{11}$>)
$o_9$ = ($i_9$, tuple, <Dire:$i_{12}$, FechaIngresoDirector:$i_6$>)
$o_{10}$ = ($i_{10}$, set, $\{i_{12}, i_{13}, i_{14}\}$)

[13] También denominado *nombre de variable de instancia* en la terminología OO.

[14] También denominado multiconjunto.

[15] Como mencionamos anteriormente, no es práctico generar un identificador de sistema único por cada valor, por lo que los sistemas reales permiten tanto OIDs como *valores estructurados*, que pueden estructurarse utilizando los mismos constructores de tipos como objetos, sólo que un valor *no tiene* un OID.

$o_{11}$ = ($i_{11}$, set {$i_{15}$, $i_{16}$, $i_{17}$})

$o_{12}$ = ($i_{12}$, tuple, <Nombre:$i_{18}$, Apellido1:$i_{19}$, Apellido2:$i_{20}$, Dni:$i_{21}$, ..., Sueldo:$i_{26}$, Supervisor:$i_{27}$, Dept:$i_8$>)

...

Los primeros seis objetos ($o_1$–$o_6$) de la lista representan valores atómicos. Habrá muchos objetos similares, uno por cada valor atómico constante distinto de la base de datos.[16] El objeto $o_7$ es un objeto de valor conjunto que representa el conjunto de ubicaciones para el departamento 5; el conjunto {$i_1$, $i_2$, $i_3$} se refiere a los objetos atómicos con los valores {'Madrid', 'Valencia', 'Sevilla'}. El objeto $o_8$ es un objeto de valor tupla que representa el propio departamento 5, y tiene los atributos NombreDpto, NúmeroDpto, Dire, Ubicaciones, etcétera. Los dos primeros atributos, NombreDpto y NúmeroDpto, tienen los objetos atómicos $o_5$ y $o_4$ como sus valores. El atributo Dire tiene como valor un objeto tupla $o_9$, que a su vez tiene dos atributos. El valor del atributo Dire es el objeto cuyo OID es $i_{12}$, que representa al empleado 'José Pérez Pérez' que dirige el departamento; mientras que el valor de FechaIngresoDirector es otro objeto atómico cuyo valor es una fecha. El valor del atributo Empleados de $o_8$ es un objeto conjunto con el OID = $i_{10}$, cuyo valor es el conjunto de identificadores de objeto para los empleados que trabajan para el DEPARTAMENTO (objetos $i_{12}$, más $i_{13}$ e $i_{14}$, que no se muestran). De forma parecida, el valor del atributo Proyectos de $o_8$ es un objeto conjunto con OID = $i_{11}$, cuyo valor es el conjunto de identificadores de objeto para los proyectos que están controlados por el departamento 5 (objetos $i_{15}$, $i_{16}$ e $i_{17}$, que no se muestran). El objeto cuyo OID = $i_{12}$ representa al empleado 'José Pérez Pérez' con todos sus atributos atómicos (Nombre, Apellido1, Apellido2, Dni, ..., Sueldo, que están haciendo referencia a los objetos atómicos $i_{18}$, $i_{19}$, $i_{20}$, $i_{21}$, ..., $i_{26}$, respectivamente [no se muestran]) más Supervisor, que hace referencia al objeto EMPLEADO con el OID = $i_{27}$ (representa a 'Eduardo Ochoa Paredes' que supervisa a 'José Pérez Pérez', pero no se muestra) y Dept que hace referencia al objeto departamento con OID = $i_8$ (representa al departamento 5 donde trabaja 'José Pérez Pérez').

En este modelo, un objeto se puede representar como una estructura de grafo que puede construirse aplicando recursivamente los constructores de tipos. El grafo que representa a un objeto $o_i$ puede construirse creando primero un nodo para el propio objeto $o_i$. El nodo para $o_i$ se etiqueta con el OID y el constructor del objeto $c$. También creamos un nodo en el grafo para cada valor atómico básico. Si un objeto $o_i$ tiene un valor atómico, dibujamos un arco tendente desde el nodo que representa $o_i$ hasta el nodo que representa su valor básico. Si construimos el valor del objeto, dibujamos arcos tendentes desde el nodo del objeto hasta el nodo que representa el valor construido. La Figura 20.1 muestra el grafo para el objeto DEPARTAMENTO $o_8$ dado anteriormente.

El modelo anterior permite dos tipos de definiciones en una comparación por igualdad de los *estados de dos objetos*. Se dice que dos objetos tienen **estados idénticos** (igualdad profunda) si los gráficos que representan sus estados son idénticos en cada respecto, incluyendo los OIDs de cada nivel. Por otro lado, la definición débil de igualdad es cuando dos objetos tienen **estados iguales** (igualdad poco profunda o superficial). En este caso, las estructuras gráficas deben ser iguales, y todos los valores atómicos correspondientes de los gráficos también deben ser iguales. Sin embargo, algunos nodos internos correspondientes en los dos gráficos pueden tener objetos con *OIDs diferentes*.

**Ejemplo 2: Objetos idénticos frente a iguales.** Un ejemplo puede ilustrar la diferencia entre las dos definiciones de la comparación por igualdad de los estados de los objetos. Supongamos que tenemos los objetos $o_1$, $o_2$, $o_3$, $o_4$, $o_5$ y $o_6$:

$o_1$ = ($i_1$, tuple, <$a_1$:$i_4$, $a_2$:$i_6$>)

$o_2$ = ($i_2$, tuple, <$a_1$:$i_5$, $a_2$:$i_6$>)

[16] Estos objetos atómicos son los que pueden provocar un problema, debido al uso de demasiados identificadores de objeto, si este modelo se implementa directamente.

$o_3$ = ($i_3$, tuple, <$a_1$:$i_4$, $a_2$:$i_6$>)
$o_4$ = ($i_4$, atom, 10)
$o_5$ = ($i_5$, atom, 10)
$o_6$ = ($i_6$, atom, 20)

Los objetos $o_1$ y $o_2$ tienen estados de *igualdad*, puesto que sus estados a nivel atómico son iguales, pero los valores se alcanzan a través de los objetos $o_4$ y $o_5$ distintos. No obstante, los estados de los objetos $o_1$ y $o_3$ son *idénticos*, aunque los propios objetos no lo son porque tienen OIDs distintos. De forma parecida, aunque los estados de $o_4$ y $o_5$ son idénticos, los objetos reales $o_4$ y $o_5$ son iguales pero no idénticos, porque tienen OIDs distintos.

### 20.2.3 Constructores de tipos

Un **lenguaje de definición de objetos (ODL)**[17] que incorpora los constructores de tipos anteriores se puede utilizar para definir los tipos de objetos para una aplicación de base de datos particular. En el Capítulo 21 describiremos el ODL estándar de ODMG, pero primero vamos a introducir gradualmente los conceptos en esta sección haciendo uso de una notación sencilla. Los constructores de tipo pueden utilizarse para definir las *estructuras de datos* para un *esquema de base de datos OO*. En la Sección 20.3 veremos cómo incorporar la definición de *operaciones* (o métodos) dentro de un esquema OO. La Figura 20.2 muestra la declaración de los tipos EMPLEADO y DEPARTAMENTO correspondientes a las instancias de objeto mostradas en la Figura 20.1. En la Figura 20.2, el tipo FECHA se define como una tupla, más que como un valor atómico, como en la Figura 20.1. También utilizamos las palabras clave tuple, set y list para los constructores de tipo, y los tipos de datos estándar disponibles (integer, string, float, etcétera) para los tipos atómicos.

Los atributos que se refieren a otros objetos (como Dept de EMPLEADO o Proyectos de DEPARTAMENTO) son básicamente **referencias** a otros objetos y sirven para representar *relaciones* entre los tipos de objeto. Por ejemplo, el atributo Dept de EMPLEADO es un tipo DEPARTAMENTO, y por tanto se utiliza para referirse a un objeto DEPARTAMENTO específico (donde trabaja el EMPLEADO). El valor de un atributo semejante sería un OID para un objeto DEPARTAMENTO específico. Una relación binaria se puede representar en una dirección, o puede tener una *referencia inversa*. La última representación facilita atravesar la relación en ambas direcciones. Por ejemplo, el atributo Empleados de DEPARTAMENTO tiene como valor un *conjunto de referencias* (es decir, un conjunto de OIDs) a objetos de tipo EMPLEADO; se trata de los empleados que trabajan para el DEPARTAMENTO. Lo inverso es el atributo de referencia Dept de EMPLEADO. En el Capítulo 21 veremos cómo el estándar ODMG permite declarar explícitamente lo inverso como atributos de relación para garantizar que las referencias inversas sean coherentes.

## 20.3 Encapsulamiento de operaciones, métodos y persistencia

El concepto de *encapsulamiento* es una de las principales características de los lenguajes y sistemas OO. También está relacionado con los conceptos de *tipos de datos abstractos* y *ocultación de información* en los lenguajes de programación. En los modelos y sistemas de bases de datos tradicionales, este concepto no se aplicaba porque lo habitual era que la estructura de los objetos de la base de datos fuera visible para los usuarios y programas externos. En estos modelos tradicionales, se aplicaban ciertas operaciones de bases de datos a los objetos de todos los tipos. Por ejemplo, en el modelo relacional, las operaciones para seleccionar, insertar, eliminar y modificar tuplas son genéricas y pueden aplicarse a *cualquier relación* de la base de datos. La

[17] Esto corresponde al DDL (Lenguaje de definición de datos) del sistema de bases de datos (consulte el Capítulo 2).

relación y sus atributos son visibles a los usuarios y a los programas externos que acceden a la relación utilizando estas operaciones.

**Figura 20.1.** Representación gráfica de un objeto complejo DEPARTAMENTO.

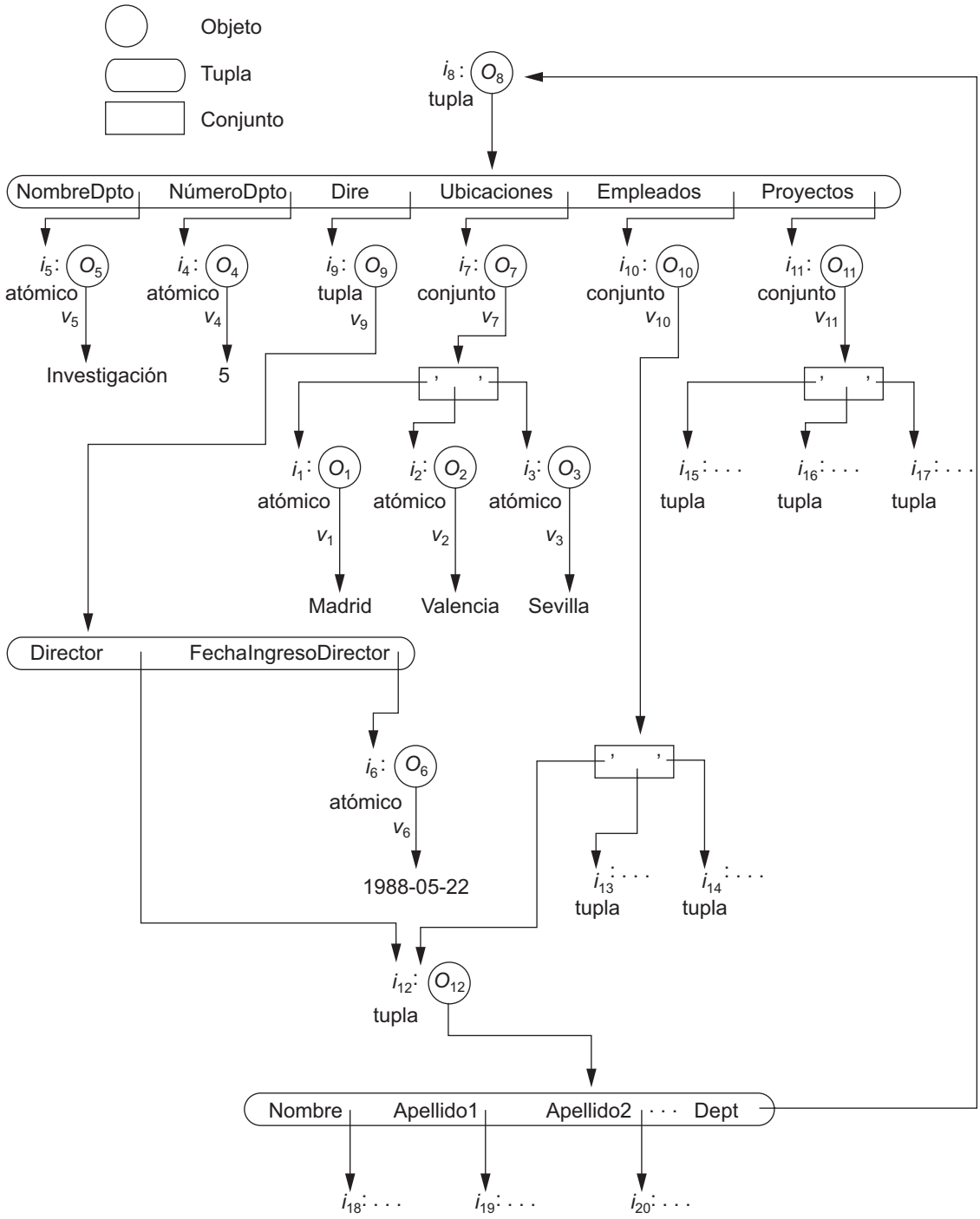


**Figura 20.2.** Especificación de los tipos de objeto EMPLEADO, FECHA y DEPARTAMENTO utilizando los constructores de tipo.

```
define type   EMPLEADO
        tuple (Nombre:       string;
               Apellido1:    string;
               Apellido2:    string;
               Dni:          string;
               FechaNac:     DATE;
               Dirección:    string;
               Sexo:         char;
               Sueldo:       float;
               Supervisor:   EMPLEADO;
               Dept:         DEPARTAMENTO;

define type FECHA
        tuple (Año: integer;
               Mes: integer;
               Día: integer; );

define type DEPARTAMENTO
        tuple (NombreDpto:  string;
               NúmeroDpto:  integer;
               Dire: tuple (  Director: EMPLEADO;
                              FechaIngreso: FECHA; );
               Ubicaciones: set(string);
               Empleados:   set(EMPLEADO);
               Proyectos    set(PROYECTO); );
```

## 20.3.1 Especificación del comportamiento de los objetos a través de operaciones de clase

Los conceptos de ocultación de información y encapsulamiento pueden aplicarse a los objetos de bases de datos. La idea principal es definir el **comportamiento** de un tipo de objeto basándose en las **operaciones** que pueden aplicarse externamente a los objetos de ese tipo. La estructura interna del objeto está oculta, y el objeto sólo es accesible a través de determinadas operaciones predefinidas. Algunas operaciones se pueden utilizar para crear (insertar) o destruir (eliminar) objetos; otras operaciones pueden actualizar el estado del objeto; y otras pueden utilizarse para recuperar partes del estado del objeto o para aplicar algunos cálculos. Y otras pueden llevar a cabo una combinación de recuperación, cálculo y actualización. En general, la **implementación** de una operación puede especificarse en un *lenguaje de programación de propósito general* que proporciona flexibilidad y potencia en la definición de operaciones.

Los usuarios externos del objeto sólo son conscientes de la **interfaz** del tipo objeto, que define el nombre y los argumentos (parámetros) de cada operación. La implementación se oculta a los usuarios externos; esto incluye la definición de las estructuras de datos internas del objeto y la implementación de las operaciones que acceden a esas estructuras. En la terminología OO, la parte de interfaz de cada operación se denomina **firma**, y la implementación de la operación se denomina **método**. Normalmente, un método es invocado enviando un **mensaje** al objeto para ejecutar el método correspondiente. Como parte de la ejecución de un

método, es posible enviar un mensaje subsiguiente a otro objeto y este mecanismo puede utilizarse para devolver valores de los objetos al entorno exterior o a otros objetos.

Para las aplicaciones de bases de datos, el requisito de que todos los objetos deben estar completamente encapsulados es demasiado exigente. Una forma de relajar este requisito es dividir la estructura de un objeto en atributos **visible** y **hidden** (oculto) (variables de instancia). Es posible acceder directamente a los atributos visibles para lectura mediante operadores externos o con un lenguaje de consulta de alto nivel. Los atributos ocultos de un objeto están completamente encapsulados y sólo se puede acceder a ellos mediante operaciones predefinidas. La mayoría de los OODBMSs emplean lenguajes de consulta de alto nivel para acceder a los atributos visibles. En el Capítulo 21 explicaremos el lenguaje de consulta SQL que se propuso como lenguaje de consulta estándar para los OODBs.

En la mayoría de los casos, las operaciones que *actualizan* el estado de un objeto se encapsulan. Es una forma de definir la semántica de actualización de los objetos, dado que en muchos modelos de datos OO se predefinen pocas restricciones de integridad en el esquema. Cada tipo de objeto tiene sus restricciones de integridad *programadas en los métodos* que crean, eliminan y actualizan los objetos escribiendo explícitamente el código para comprobar las violaciones de restricción y para manipular las excepciones. En estos casos, todas las operaciones de actualización son implementadas por operaciones encapsuladas. Más recientemente, el ODL para el estándar ODMG permite especificar algunas restricciones comunes como las claves y las relaciones inversas (integridad referencial) para que el sistema pueda implantar automáticamente dichas restricciones (consulte el Capítulo 21).

El término **clase** se utiliza a menudo para referirse a la definición de un tipo de objeto, junto con las definiciones de las operaciones para ese objeto.[18] La Figura 20.3 muestra cómo las definiciones de tipo de la Figura 20.2 se pueden extender con operaciones para definir clases. Para cada clase se declaran varias operaciones, y en la definición de clase se incluye la firma (interfaz) de cada operación. Por otra parte, con un lenguaje de programación hay que definir un método (implementación) para cada operación. Las operaciones típicas incluyen la operación **constructor de objeto**, que se utiliza para crear un objeto nuevo, y la operación **destructor**, que se utiliza para destruir un objeto. También podemos declarar varias operaciones **modificador de objeto** para modificar los estados (valores) de varios atributos de un objeto. Operaciones adicionales pueden **recuperar** información sobre el objeto.

Una operación se aplica normalmente a un objeto utilizando la **notación de punto**. Por ejemplo, si d es una referencia a un objeto DEPARTAMENTO, podemos invocar una operación como num_empleados escribiendo d.num_empleados. De forma parecida, si escribimos d.destruir_dept, el objeto referenciado por d se destruye (se elimina). La única excepción es la operación constructor, que devuelve una referencia a un objeto DEPARTAMENTO nuevo. Por tanto, es normal tener un nombre predeterminado para la operación constructor que es el nombre de la propia clase, aunque esto no se utilizó en la Figura 20.3.[19] La notación de punto también se utiliza para referirnos a los atributos de un objeto; por ejemplo, escribiendo d.NúmeroDpto o d.Dire.FechaIngreso.

### 20.3.2 Cómo especificar la persistencia de objeto a través de la denominación y la noción de alcance

Un OODBMS se acopla estrechamente con un Lenguaje de programación orientado a objetos (OOPL). El OOPL se utiliza para especificar las implementaciones de método, así como otro código de aplicación. Un

[18] Esta definición de *clase* es parecida a la que se utiliza en el conocido lenguaje de programación C++. El estándar ODMG utiliza la palabra *interfaz* además de *clase* (consulte el Capítulo 21). En el modelo EER, el término *clase* se utilizaba para referirse a un tipo objeto, junto con el conjunto de todos los objetos de ese tipo (consulte el Capítuo 4).

[19] Los nombres predeterminados para las operaciones de constructor y destructor existen en el lenguaje de programación C++. Por ejemplo, para la clase EMPLEADO, el *nombre de constructor predeterminado* es EMPLEADO y el *nombre de destructor predeterminado* es ~EMPLEADO. También es frecuente utilizar la operación *nuevo* para crear *objetos nuevos*.

**Figura 20.3.** Adición de operaciones a las definiciones de EMPLEADO y DEPARTAMENTO.

```
define class EMPLEADO
        type tuple ( Nombre:        string;
                     Apellido1:     string;
                     Apellido2:     string;
                     Dni:           string;
                     FechaNac:      FECHA;
                     Dirección:     string;
                     Sexo:          char;
                     Sueldo:        float;
                     Supervisor:    EMPLEADO;
                     Dept:          DEPARTAMENTO; );
        operations   edad:          integer;
                                    crear_emp:          EMPLEADO;
                                    destruir_emp:       boolean;
end EMPLEADO;

define class DEPARTAMENTO
        type tuple ( NombreDpto:    string;
                     NúmeroDpto:    integer;
                     Dire: tuple (  Director: EMPLEADO;
                                    FechaIngreso: DATE; );
                     Ubicaciones:   set(string);
                     Empleados:     set(EMPLEADO);
                     Proyectos      set(PROYECTO); );
operations           num_empleados: integer;
                     crear_dept:    DEPARTAMENTO;
                     destruir_dept: boolean;
                     asignar_emp(e: EMPLEADO): boolean;
                     (* añade un empleado al departamento *)
                     eliminar_emp(e: EMPLEADO): boolean;
                     (* elimina un empleado del departamento *)
end DEPARTAMENTO;
```

objeto se crea normalmente ejecutando algún programa de aplicación, invocando la operación del constructor de objeto. No todos los objetos se guardan permanentemente en la base de datos. Los **objetos transitorios** existen en el programa que se está ejecutando y desaparecen una vez que ese programa termina. Los **objetos persistentes** se almacenan en la base de datos y persisten después de la terminación del programa. Los mecanismos típicos para hacer que un objeto sea persistente son la *denominación* y la *noción de alcance* (*reachability*).

El **mecanismo de denominación** implica asignar a un objeto un nombre persistente único con el que el programa actual y otros programas pueden recuperarlo. Este nombre de objeto persistente puede suministrarse mediante una sentencia u operación específica del programa (véase la Figura 20.4). Los nombres que se suministran a los objetos han de ser únicos dentro de una base de datos particular. Por tanto, los objetos persistentes denominados se utilizan como **puntos de entrada** a la base de datos a través de los cuales los usuarios y las aplicaciones pueden iniciar su acceso a la base de datos. Obviamente, no resulta práctico asignar un nombre a todos los objetos de una base de datos grande que incluye miles de objetos, por lo que la mayoría de los

**Figura 20.4.** Creación de objetos persistentes mediante la denominación y la noción de alcance.

```
define class CONJUNTO_DPTO:
    type set (DEPARTAMENTO);
    operations    agregar_dept(d: DEPARTAMENTO): boolean;
        (* añade un departamento al objeto CONJUNTO_DPTO *)
            eliminar_dept(d: DEPARTAMENTO): boolean;
        (* elimina un departamento del objeto CONJUNTO_DPTO *)
            crear_conjunto_dept: CONJUNTO_DPTO;
            destruir_conjunto_dept: boolean;
end ConjuntoDept;

...

persistent name TODOS_DEPARTAMENTOS: CONJUNTO_DPTO;
(* TODOS_DEPARTAMENTOS es un objeto con nombre persistente de tipo CONJUNTO_DPTO *)

...

d:= crear_dept;
(* crea un objeto DEPARTAMENTO nuevo en la variable d *)

...

b:= TODOS_DEPARTAMENTOS.agregar_dept(d);
(* convierte a d en persistente añadiéndolo al conjunto persistente TODOS_DEPARTAMENTOS *)
```

objetos se hacen persistentes utilizando el segundo mecanismo, denominado **noción de alcance**. Este otro mecanismo funciona haciendo que el objeto sea alcanzable desde algún objeto persistente. Se dice que un objeto *B* es **alcanzable** desde un objeto *A* si una secuencia de referencias en el gráfico del objeto conduce desde el objeto *A* al objeto *B*. Por ejemplo, todos los objetos de la Figura 20.1 son alcanzables desde el objeto $o_8$; por tanto, si $o_8$ se hace persistente, todos los demás objetos de la Figura 20.1 también se hacen persistentes.

Si primero creamos un objeto persistente denominado *N*, cuyo estado es un *conjunto* o una *lista* de objetos de alguna clase *C*, podemos conseguir que los objetos de *C* sean persistentes *añadiéndolos* al conjunto o la lista, y haciéndolos así alcanzables o accesibles desde *N*. Por tanto, *N* define una **colección persistente** de objetos de clase *C*. Por ejemplo, podemos definir una clase CONJUNTO_DPTO (véase la Figura 20.4) cuyos objetos sean de tipo **set**(DEPARTAMENTO).[20]

Supongamos que creamos un objeto de tipo CONJUNTO_DPTO y que le asignamos el nombre TODOS_DEPARTAMENTOS, convirtiéndolo así en persistente (véase la Figura 20.4). Cualquier objeto DEPARTAMENTO que añadamos al conjunto de TODOS_DEPARTAMENTOS utilizando la operación agregar_dept se convertirá en persistente por ser alcanzable desde TODOS_DEPARTAMENTOS. El objeto TODOS_DEPARTAMENTOS se denomina con frecuencia **extensión** de la clase DEPARTAMENTO, pues albergará todos los objetos persistentes de tipo DEPARTAMENTO. Como veremos en el Capítulo 21, el estándar ODL ODMG le ofrece al diseñador del esquema la opción de designar una extensión como parte de la definición de una clase.

Observe la diferencia entre los modelos de bases de datos tradicionales y las bases de datos OO en este asunto. En los modelos de bases de datos tradicionales, como el modelo relacional o el modelo EER, se asume que *todos* los objetos son persistentes. Así pues, cuando en el modelo EER se define un tipo o clase de entidad

[20] Como veremos en el Capítulo 21, la sintaxis del ODL ODMG utiliza **set** <DEPARTAMENTO> en lugar de **set** (DEPARTAMENTO).

como EMPLEADO, representa tanto la *declaración de tipo* para EMPLEADO como el *conjunto persistente de todos los objetos* EMPLEADO. En el método OO, una declaración de clase de EMPLEADO sólo especifica el tipo y las operaciones para una clase de objetos. El usuario, si lo desea, debe definir por separado un objeto persistente de tipo set(EMPLEADO) o list(EMPLEADO) cuyo valor es la *colección de referencias* a todos los objetos EMPLEADO persistentes (véase la Figura 20.4).[21] Esto permite que los objetos transitorios y persistentes obedezcan las mismas declaraciones de tipo y clase del ODL y el OOPL. En general, es posible definir varias colecciones persistentes para la misma definición de clase, si es lo deseado.

# 20.4 Herencia y jerarquías de tipos y clases

Otra característica importante de los sistemas de bases de datos OO es que permiten las jerarquías de tipos y la herencia. Las jerarquías de tipos en las bases de datos normalmente implican una restricción en las extensiones correspondientes a los tipos de la jerarquía. En primer lugar, en la Sección 20.4.1 explicamos las jerarquías de tipos, y después explicamos en la Sección 20.4.2 las restricciones en las extensiones. En esta sección utilizamos un modelo OO diferente (un modelo en el que los atributos y las operaciones se tratan uniformemente), ya que los atributos y las operaciones pueden heredarse. En el Capítulo 21 explicamos el modelo de herencia del estándar ODMG, que difiere del modelo aquí explicado.

## 20.4.1 Jerarquías de tipos y herencia

En la mayoría de las aplicaciones de bases de datos, hay numerosos objetos del mismo tipo o clase. Por tanto, las bases de datos OO deben proporcionar la posibilidad de clasificar los objetos por su tipo, como hacen otros sistemas de bases de datos. Pero en las bases de datos OO, un requisito añadido es que el sistema permita la definición de nuevos tipos basándose en otros tipos predefinidos, cargándolos en una **jerarquía de tipos** (o **clases**). Algunos sistemas conservan por separado una jerarquía de tipos definida por el usuario.

Normalmente, un tipo se define asignándole un nombre de tipo y después definiendo varios atributos (variables de instancia) y operaciones (métodos) para ese tipo.[22]

En algunos casos, los atributos y las operaciones se denominan en conjunto *funciones*, ya que los atributos se parecen a funciones sin argumentos. El nombre de una función lo podemos utilizar para referirnos al valor de un atributo o al valor resultante de una operación (método). En esta sección utilizamos el término **función** para referirnos indistintamente a los atributos y las operaciones de un tipo objeto, ya que se tratan de una forma parecida en una introducción básica a la herencia.[23]

Un tipo en su forma más simple puede definirse asignándole un **nombre de tipo** y, después, enumerando los nombres de sus **funciones** visibles (públicas). Al especificar un tipo en esta sección, utilizamos el siguiente formato, que no especifica argumentos de funciones para simplificar la explicación:

NOMBRE_TIPO: función, función, ..., función

Por ejemplo, un tipo que describe las características de una PERSONA puede definirse de este modo:

PERSONA: Nombre, Direcc, FechaNac, Edad, Dni

En el tipo PERSONA, las funciones Nombre, Direcc, Dni y FechaNac pueden implementarse como atributos almacenados, mientras que la función Edad puede implementarse como un método que calcula la Edad a partir del valor del atributo FechaNac y la fecha actual.

---

[21] Algunos sistemas, como POET, crean automáticamente la extensión para una clase.

[22] En esta sección utilizaremos los términos *tipo* y *clase* con el mismo significado (a saber, los atributos y las operaciones de algún tipo de objeto).

[23] En el Capítulo 21 veremos que los tipos con funciones se parecen al concepto de interfaces tal como se utilizan en el ODL ODMG.

El concepto de **subtipo** resulta de utilidad cuando el diseñador o el usuario debe crear un tipo nuevo parecido pero no idéntico a un tipo definido ya existente. El subtipo hereda entonces todas las funciones del tipo predefinido, al que denominaremos **supertipo**. Por ejemplo, supongamos que queremos definir dos tipos nuevos, EMPLEADO y ESTUDIANTE, de este modo:

EMPLEADO: Nombre, Direcc, FechaNac, Edad, Dni, Sueldo, FechaContrato, TiempoEnLaEmpresa

ESTUDIANTE: Nombre, Direcc, FechaNac, Edad, Dni, Especialidad, NotaMedia

Como ESTUDIANTE y EMPLEADO incluyen todas las funciones definidas para PERSONA, más algunas funciones adicionales propias, podemos declararlas como **subtipos** de PERSONA. Cada una heredará las funciones definidas anteriormente de PERSONA (Nombre, Direcc, FechaNac, Edad y Dni). Para ESTUDIANTE, sólo tenemos que definir las funciones nuevas (locales) Especialidad y NotaMedia, que no se heredan. Probablemente, Especialidad puede definirse como un atributo almacenado, mientras que NotaMedia puede implementarse como un método que calcula la nota media del estudiante accediendo a los valores Nota que están almacenados internamente (ocultos) dentro de cada objeto ESTUDIANTE a modo de *atributos privados*. En el caso de EMPLEADO, las funciones Sueldo y FechaContrato pueden almacenarse como atributos, mientras que TiempoEnLaEmpresa puede ser un método que calcula la antigüedad a partir de FechaContrato.

La idea de definir un tipo implica definir todas sus funciones e implementarlas como atributos o como métodos. Al definir un subtipo, puede heredar todas estas funciones y sus implementaciones. Sólo las funciones que son específicas o **locales** al subtipo, y por tanto no están especificadas en el supertipo, tienen que definirse e implementarse. Por consiguiente, podemos declarar EMPLEADO y ESTUDIANTE de este modo:

EMPLEADO **subtipo-de** PERSONA: Sueldo, FechaContrato, TiempoEnLaEmpresa

ESTUDIANTE **subtipo-de** PERSONA: Especialidad, NotaMedia

En general, un subtipo incluye *todas* las funciones que están definidas para su supertipo, además de algunas funciones adicionales que son específicas del subtipo. Así pues, es posible generar una **jerarquía de tipos** para mostrar las relaciones supertipo/subtipo junto con todos los tipos declarados en el sistema.

A modo de ejemplo, consideremos un tipo que describe los objetos en la geometría plana, que puede definirse de este modo:

OBJETO_GEOMÉTRICO: Forma, Área, PuntoReferencia

Para el tipo OBJETO_GEOMÉTRICO, Forma se implementa como un atributo (su dominio puede ser un tipo enumerado con los valores ‘triángulo, ‘rectángulo, ‘círculo, etcétera), y Área es un método que se aplica para calcular el área. PuntoReferencia especifica las coordenadas que determinan la ubicación del objeto. Ahora supongamos que queremos definir varios subtipos para el tipo OBJETO_GEOMÉTRICO, así:

RECTÁNGULO **subtipo-de** OBJETO_GEOMÉTRICO: Anchura, Altura

TRIÁNGULO **subtipo-de** OBJETO_GEOMÉTRICO: Lado1, Lado2, Ángulo

CÍRCULO **subtipo-de** OBJETO_GEOMÉTRICO: Radio

La operación Área puede implementarse con un método diferente para cada subtipo, puesto que el procedimiento para calcular el área es diferente para los rectángulos, los triángulos y los círculos. De forma parecida, el atributo PuntoReferencia puede tener un significado diferente para cada subtipo; puede ser el punto central de los objetos RECTÁNGULO y CÍRCULO, y el vértice entre dos lados dados de un objeto TRIÁNGULO. Algunos sistemas de bases de datos OO permiten **renombrar** las funciones heredadas en subtipos diferentes para reflejar el significado de un modo más exacto.

Un modo alternativo de declarar estos subtipos consiste en especificar el valor del atributo Forma como una condición que debe cumplirse para los objetos de cada subtipo:

RECTÁNGULO **subtipo-de**

OBJETO_GEOMÉTRICO (Forma=‘rectángulo’): Anchura, Altura

TRIÁNGULO **subtipo-de**
OBJETO_GEOMÉTRICO (Forma=‘triángulo’): Lado1, Lado2, Ángulo

CÍRCULO **subtipo-de** OBJETO_GEOMÉTRICO (Forma=‘círculo’): Radio

Aquí, sólo los objetos OBJETO_GEOMÉTRICO que cumplen la condición Forma=‘rectángulo’ son del subtipo RECTÁNGULO, algo parecido a lo que ocurre con los otros dos subtipos. En este caso, todas las funciones del supertipo OBJETO_GEOMÉTRICO son heredadas por cada uno de los tres subtipos, pero el valor del atributo Forma queda restringido a un valor específico para cada uno.

Las definiciones de tipo describen objetos, pero no los generan. Simplemente son declaraciones de ciertos tipos; y como parte de esa declaración, se especifica la implementación de las funciones de cada tipo. En una aplicación de bases de datos hay muchos objetos de cada tipo. Al crear un objeto, normalmente pertenece a uno o más de esos tipos que se han declarado. Por ejemplo, un objeto círculo es de tipo CÍRCULO y OBJETO_GEOMÉTRICO (por herencia). Cada objeto también se convierte en miembro de una o más colecciones de objetos persistentes (o extensiones), que se utilizan para agrupar colecciones de objetos que son significativos para la aplicación de bases de datos.

### 20.4.2 Restricciones en las extensiones correspondientes a una jerarquía de tipos[24]

En la mayoría de las bases de datos OO, la colección de datos de una extensión (*extent*) tiene el mismo tipo o clase. Sin embargo, no es una condición necesaria. Por ejemplo, Smalltalk, un lenguaje denominado lenguaje OO *typeless* (sin tipo), permite una colección de objetos que contiene objetos de diferentes tipos. También es el caso de otros lenguajes sin tipo no orientado a objetos, como LISP, cuando se amplían con conceptos de OO. No obstante, como la mayoría de las bases de datos OO soportan tipos, asumiremos en el resto de esta sección que las extensiones son colecciones de objetos del mismo tipo.

Es común en las aplicaciones de bases de datos que cada tipo o subtipo tenga una extensión asociada, que alberga la colección de todos los objetos persistentes de ese tipo o subtipo. En este caso, la restricción es que cada objeto de una extensión que corresponde a un subtipo también debe ser miembro de la *extensión* que corresponde a su subtipo. Algunos sistemas de bases de datos OO tienen un tipo de sistema predefinido (denominado clase ROOT [raíz] o clase OBJECT [objeto]) cuya extensión contiene todos los objetos del sistema.[25]

La clasificación procede entonces asignando objetos en los subtipos adicionales que son significativos para la aplicación, creando una jerarquía de tipos o de clases para el sistema. Todas las extensiones para el sistema y las clases definidas por el usuario son subconjuntos de la extensión que corresponde a la clase OBJECT, directa o indirectamente. En el modelo ODMG (consulte el Capítulo 21), el usuario puede o no especificar una extensión para cada clase (tipo), dependiendo de la aplicación.

En la mayoría de los sistemas OO se hace una distinción entre los objetos persistentes y transitorios y las colecciones. Una colección persistente alberga una colección de objetos que se almacenan permanentemente en la base de datos y, por tanto, varios programas pueden acceder a ellos y compartirlos. Una colección transitoria existe temporalmente durante la ejecución de un programa, pero no se conserva más allá de la terminación del mismo. Por ejemplo, en un programa puede crearse una colección transitoria para conservar el resultado de una consulta que selecciona algunos objetos de una colección persistente y los copia en una

[24] En la segunda edición de este libro utilizábamos el término *jerarquías de clases* para describir estas restricciones de extensión. Como la palabra *clase* tiene muchos significados diferentes, en esta sección utilizamos *extensión*. Esto también es más coherente con la terminología ODMG (consulte el Capítulo 21).

[25] Esto se denomina OBJECT en el modelo ODMG (consulte el Capítulo 21).

colección transitoria. La colección transitoria alberga el mismo tipo de objetos que la colección persistente. El programa puede entonces manipular los objetos de la colección transitoria, y una vez que el programa termina, la colección transitoria deja de existir. En general, numerosas colecciones (transitorias o persistentes) pueden contener objetos del mismo tipo.

Los constructores de tipos explicados en la Sección 20.2 permiten que el estado de un objeto sea una colección de objetos. Por tanto, los objetos colección cuyos tipos están basados en el *constructor conjunto* pueden definir varias colecciones (una por cada objeto). Los objetos con valoración de conjunto son miembros de otra colección. Esto permite esquemas de clasificación multinivel, en los que un objeto de una colección tiene como estado una colección de objetos de una clase diferente.

Como veremos en el Capítulo 21, el modelo ODMG distingue entre herencia de tipo (denominada herencia de interfaz e indicada mediante dos puntos [:]) y la restricción de herencia de extensión (indicada por la palabra clave EXTEND).

# 20.5 Objetos complejos

Una motivación importante que llevó al desarrollo de sistemas OO fue el deseo de representar objetos complejos. Hay dos tipos principales de objetos complejos: estructurados y no estructurados. Un objeto complejo estructurado está formado por componentes y se define aplicando los constructores de tipo disponibles recursivamente a varios niveles. Un objeto complejo no estructurado normalmente es un tipo de datos que requiere una gran cantidad de almacenamiento, como el tipo de datos que representa una imagen o un objeto de texto grande.

## 20.5.1 Objetos complejos no estructurados y extensibilidad de tipos

La característica de **objeto complejo no estructurado** proporcionada por un DBMS permite almacenar y recuperar los objetos grandes que la aplicación de bases de datos necesita. Ejemplos típicos de estos objetos son las *imágenes de mapas de bits* (bitmaps) y las *cadenas de texto largas* (documentos); también se conocen como **objetos binarios grandes** (**BLOBs**, *binary large objects*). Las cadenas de caracteres también se denominan **objetos grandes de caracteres** (**CLOBs**, *character large objects*). Estos objetos no están estructurados en el sentido de que el DBMS no conoce su estructura (sólo la aplicación que los utiliza puede interpretar su significado). Por ejemplo, la aplicación puede tener funciones para visualizar una imagen o para buscar por determinadas palabras clave o detectar frases o párrafos en una cadena de texto larga. Se considera que los objetos son complejos cuando requieren una gran área de almacenamiento y no forman parte de los tipos de datos estándar proporcionados por los DBMSs tradicionales. Debido a que el tamaño del objeto es muy grande, un DBMS puede recuperar una porción del objeto y suministrársela a la aplicación antes de que se recupere el objeto entero. El DBMS también utiliza las técnicas de almacenamiento en búfer y en caché para recopilar las porciones del objeto antes de que el programa de aplicación tenga necesidad de acceder a él.

El software DBMS no tiene la capacidad de procesar directamente las condiciones de selección y otras operaciones basadas en los valores de esos objetos, a menos que la aplicación proporcione el código para efectuar las operaciones de comparación de la selección. En un OODBMS, esto puede realizarse definiendo un nuevo tipo de datos abstractos y proporcionando los métodos para seleccionar, comparar y visualizar dichos objetos. Por ejemplo, vamos a suponer que los objetos son imágenes bitmap bidimensionales. Supongamos que la aplicación tiene que seleccionar, a partir de una colección de dichos objetos, aquellos que incluyen un determinado patrón. En este caso, el usuario debe proporcionar el programa de reconocimiento de patrones como un método de los objetos que son de tipo bitmap. El OODBMS recupera entonces un objeto de la base de datos y ejecuta el método para reconocimiento de patrones sobre ese objeto, a fin de determinar si incluye el patrón requerido.

Como un OODBMS permite a los usuarios crear tipos nuevos, y como un tipo incluye tanto estructura como operaciones, podemos ver un OODBMS como que tiene un **sistema de tipos extensible**. Podemos crear librerías de tipos nuevos definiendo su estructura y sus operaciones, incluyendo los tipos complejos. Las aplicaciones pueden utilizar o modificar entonces esos tipos, en el último caso creando subtipos de los tipos proporcionados en las librerías. No obstante, el DBMS debe ofrecer las capacidades subyacentes de almacenamiento y recuperación de los objetos que requieren grandes cantidades de almacenamiento para que las operaciones puedan aplicarse con eficacia. La mayoría de los OODBMSs ofrecen el almacenamiento y la recuperación de objetos no estructurados grandes, como cadenas de caracteres o cadenas de bits, que pueden pasarse al programa *tal como están* para su interpretación. La tendencia actual en los DBMSs relacionales extendidos es ofrecer estas características. También están desarrollándose técnicas especiales de indexación.

## 20.5.2 Objetos complejos estructurados

Un **objeto complejo estructurado** difiere de uno no estructurado en que su estructura está definida por la aplicación repetida de los constructores de tipos suministrados por el OODBMS. Por tanto, la estructura del objeto está definida y es conocida por el OODBMS. A modo de ejemplo, consideremos el objeto DEPARTAMENTO de la Figurea 20.1. En el primer nivel, el objeto tiene una estructura de tupla con seis atributos: NombreDpto, NúmeroDpto, Dire, Ubicaciones, Empleados y Proyectos. Sin embargo, sólo dos de estos atributos (NombreDpto y NúmeroDpto) tienen valores básicos; los otros cuatro tienen una estructura compleja y, por consiguiente, constituyen el segundo nivel de la estructura del objeto complejo. Uno de esos cuatro atributos (Dire) tiene una estructura de tupla, y los otros tres (Ubicaciones, Empleados, Proyectos) tienen estructuras de conjunto. En el tercer nivel, para un valor de tupla Dire, tenemos un atributo básico (FechaIngresoDirector) y un atributo (Director) que se refiere a un objeto empleado, que tiene una estructura de tupla. Para un conjunto Ubicaciones, tenemos un conjunto de valores básicos, pero para los conjuntos Empleados y Proyectos, tenemos conjuntos de objetos estructurados como tuplas.

Entre un objeto complejo y sus componentes en cada nivel, existen dos tipos de semántica de referencia. El primer tipo, que podemos denominar **semántica de propiedad**, se aplica cuando los subobjetos de un objeto complejo se encapsulan dentro del objeto complejo y se consideran, por tanto, parte de ese objeto complejo. El segundo tipo, que podemos denominar **semántica de referencia**, se aplica cuando los componentes del objeto complejo son objetos independientes pero es posible hacer referencia a ellos desde el objeto complejo. Por ejemplo, podemos considerar los atributos NombreDpto, NúmeroDpto, Dire y Ubicaciones como propiedad de un DEPARTAMENTO, mientras que Empleados y Proyectos son referencias porque hacen referencia a objetos independientes. El primer tipo también recibe el nombre de relación *es-parte-de* o *es-componente-de*; y el segundo tipo se denomina relación *está-asociado-con*, puesto que describe una asociación de igualdad entre dos objetos independientes. La relación es-parte-de (semántica de propiedad) para la construcción de objetos complejos tiene la propiedad de que los objetos constituyentes se encapsulan dentro del objeto complejo y se consideran como parte del estado del objeto interno. No necesitan tener identificadores de objeto y sólo los métodos de ese objeto pueden acceder a ellos. Desaparecen si el propio objeto se elimina. Por el contrario, los componentes referenciados son considerados como objetos independientes que pueden tener su propia identidad y métodos. Cuando un objeto complejo tiene que acceder a sus componentes referenciados, lo hace invocando los métodos apropiados de los componentes, ya que no están encapsulados dentro del objeto complejo. Por tanto, la semántica de referencia representa las *relaciones* entre objetos independientes. Además, un objeto componente referenciado puede ser referenciado por más de un objeto complejo y, por consiguiente, no se elimina automáticamente cuando se elimina el objeto complejo.

Un OODBMS debe proporcionar opciones de almacenamiento para **agrupar** juntos los objetos constituyentes de un objeto complejo en el almacenamiento secundario, a fin de aumentar la eficacia de las operaciones que acceden al objeto complejo. En muchos casos, la estructura del objeto se almacena en páginas de disco de una forma no interpretada. Cuando una página de disco que incluye un objeto se recupera en la memoria, el OODBMS puede construir el objeto complejo estructurado a partir de la información de las páginas de

disco, que puede referirse a páginas de disco adicionales que también deben recuperarse. Es lo que se conoce como **ensamblaje de objeto complejo**.

# 20.6 Otros conceptos de orientación a objetos

En esta sección ofrecemos una panorámica de algunos conceptos OO adicionales, como el polimorfismo (sobrecarga del operador), la herencia múltiple, la herencia selectiva, el versionado y las configuraciones.

## 20.6.1 Polimorfismo (sobrecarga del operador)

Otra característica de los sistemas OO es que proporcionan el **polimorfismo** de operaciones, que también se conoce como **sobrecarga del operador**. Este concepto permite que se vincule el mismo *nombre de operador* o *símbolo* a dos o más *implementaciones* diferentes del operador, dependiendo del tipo de objetos a los que se aplique ese operador. Un ejemplo sencillo de los lenguajes de programación puede ilustrar este concepto. En algunos lenguajes, el operador "+" puede significar cosas diferentes cuando se aplica a operandos (objetos) de distintos tipos. Si los operandos de "+" son de tipo *entero*, la operación invocada es una suma de enteros. Si los operandos de "+" son del tipo *coma flotante*, la operación invocada es una suma en coma flotante. Si los operandos de "+" son de tipo *conjunto*, la operación invocada es una unión de conjuntos. El compilador puede determinar la operación que ha de ejecutarse basándose en los tipos de los operandos suministrados.

En las bases de datos OO puede darse una situación parecida. Podemos utilizar el ejemplo OBJETO_GEOMÉTRICO de la Sección 20.4 para ilustrar el polimorfismo de operación[26] en las bases de datos OO. Supongamos que declaramos OBJETO_GEOMÉTRICO y sus subtipos de este modo:

OBJETO_GEOMÉTRICO: Forma, Área, PuntoReferencia

RECTÁNGULO **subtipo-de**
    OBJETO_GEOMÉTRICO (Forma='rectángulo'): Anchura, Altura

TRIÁNGULO **subtipo-de**
    OBJETO_GEOMÉTRICO (Forma='triángulo'): Lado1, Lado2, Ángulo

CÍRCULO **subtipo-de** OBJETO_GEOMÉTRICO (Forma='círculo'): Radio

Aquí, la función Área está declarada para todos los objetos de tipo OBJETO_GEOMÉTRICO. No obstante, la implementación del método para Área puede ser distinto para cada subtipo de OBJETO_GEOMÉTRICO. Una posibilidad es contar con una implementación general que calcule el área de un OBJETO_GEOMÉTRICO generalizado (por ejemplo, escribiendo un algoritmo general para calcular el área de un polígono) y después reescribir algoritmos más eficaces para calcular las áreas de objetos geométricos de tipos específicos, como círculos, rectángulos, triángulos, etcétera. En este caso, la función Área se *sobrecarga* a causa de las diferentes implementaciones.

El OODBMS debe seleccionar ahora el método apropiado para la función Área en base al tipo de objeto geométrico al que se aplica. En los sistemas fuertemente tipados, esto puede hacerse en tiempo de compilación, ya que los tipos de objeto deben conocerse. Es lo que se denomina **vinculación temprana** (o **estática**). Sin embargo, en los sistemas con tipado débil o sin tipado (como Smalltalk y LISP), es posible que no se conozca el tipo del objeto al que se aplica la función hasta el tiempo de ejecución. En este caso, la función debe comprobar en tiempo de ejecución el tipo de objeto y, después, invocar el método adecuado. Es lo que a menudo se denomina **vinculación posterior** (o **dinámica**).

---

[26] En los lenguajes de programación hay varias clases de polimorfismo. El lector interesado puede consultar la sección "Bibliografía seleccionada".

### 20.6.2 Herencia múltiple y herencia selectiva

La **herencia múltiple** en una jerarquía de tipos se da cuando un determinado subtipo *T* es un subtipo de dos (o más) tipos y, por tanto, hereda las funciones (atributos y métodos) de los dos supertipos. Por ejemplo, podemos crear un subtipo DIRECTOR_INGENIERÍA que es un subtipo de DIRECTOR e INGENIERO. Esto lleva a la creación de una **red de tipos** más que de una jerarquía de tipos. Un problema que puede surgir con la herencia múltiple es que los supertipos de los que el subtipo hereda pueden tener funciones distintas con el mismo nombre, creándose ambigüedad. Por ejemplo, tanto DIRECTOR como INGENIERO pueden tener una función denominada Sueldo. Si esta función está implementada por diferentes métodos en los supertipos DIRECTOR e INGENIERO, existe una ambigüedad en cuanto a cuál de los dos es heredado por el subtipo DIRECTOR_INGENIERÍA. No obstante, es posible que INGENIERO y DIRECTOR hereden Sueldo del mismo supertipo (como EMPLEADO) superior en la red. Como regla general, si una función es heredada de algún *supertipo común*, entonces sólo se hereda una vez. En tal caso, no hay ambigüedad; el problema sólo surge si las funciones son distintas en los dos supertipos.

Hay varias técnicas para tratar la ambigüedad en la herencia múltiple. Una solución es que el sistema compruebe si hay ambigüedad en el momento de crear el subtipo, y dejar que el usuario seleccione explícitamente en ese momento la función que se heredará. Una segunda solución es utilizar alguna selección predeterminada del sistema. La tercera solución es prohibir completamente la herencia múltiple si se produce una ambigüedad de nombre, en lugar de obligar al usuario a cambiar el nombre de una de las funciones en uno de los supertipos. En cambio, algunos sistemas OO no permiten en absoluto la herencia múltiple.

La **herencia selectiva** se produce cuando un subtipo sólo hereda alguna de las funciones de un supertipo. Otras funciones no se heredan. En este caso, podemos utilizar una cláusula EXCEPT para listar las funciones en un supertipo que *no* serán heredadas por el subtipo. Normalmente no se suministra el mecanismo de herencia selectiva en los sistemas de bases de datos OO, pero se utiliza con frecuencia en las aplicaciones de inteligencia artificial.[27]

### 20.6.3 Versiones y configuraciones

Muchas de las aplicaciones que utilizan sistemas OO requieren la existencia de varias **versiones** del mismo objeto.[28] Por ejemplo, consideremos una aplicación de bases de datos para un entorno de ingeniería de software que almacena varios módulos software (como *módulos de diseño*, *módulos de código fuente* e *información de configuración*) para describir los módulos que al unirse forman un programa más complejo, y *casos de prueba* para testar el sistema. Normalmente, se aplican *actividades de mantenimiento* a un sistema software a medida que sus requisitos evolucionan. El mantenimiento suele implicar la modificación de algunos de los módulos de diseño e implementación. Si el sistema ya está operativo, y si es preciso cambiar uno o más de sus módulos, el diseñador debe crear una **versión nueva** de cada uno de esos módulos para implementar los cambios. De forma parecida, también es posible que haya que generar nuevas versiones de los casos de prueba para probar las versiones nuevas de los módulos. Sin embargo, las versiones existentes no deben descartarse hasta haber probado a fondo y aprobado las versiones nuevas; sólo entonces, las versiones nuevas reemplazarán a las antiguas.

Observe que puede haber más de dos versiones de un objeto. Por ejemplo, imaginemos dos programadores trabajando simultáneamente en la actualización del mismo módulo software. En este caso, se necesitan dos versiones, además del módulo original. Los programadores pueden actualizar simultáneamente sus propias versiones del mismo módulo software. Es lo que se denomina con frecuencia **ingeniería concurrente**. Sin embargo, al final será necesario mezclar esas dos versiones para que la versión nueva (híbrida) pueda

[27] En el modelo ODMG, la herencia de tipo sólo se refiere a la herencia de operaciones, no de los atributos (consulte el Capítulo 21).

[28] El versionado no es un problema único de las OODBs y puede aplicarse a los DBMSs relacionales y de otros tipos.

incluir los cambios realizados por los dos programadores. Durante la mezcla también será necesario asegurarse de que sus cambios son compartibles, lo que obliga a crear otra versión más del objeto: una que es el resultado de mezclar las dos versiones que se actualizaron independientemente.

Como se desprende de la explicación anterior, un OODBMS debe poder guardar y manipular varias versiones del mismo objeto conceptual. Varios sistemas proporcionan esta capacidad, permitiendo que la aplicación mantenga varias versiones de un objeto y se refiera explícitamente a versiones particulares según las necesidades. No obstante, el problema de mezclar y reconciliar los cambios introducidos en dos versiones diferentes normalmente se deja a los desarrolladores de la aplicación, que conocen la semántica de la misma. Algunos DBMSs tienen ciertos servicios que pueden comparar las dos versiones con el objeto original y determinar si los cambios introducidos son incompatibles, como asistencia al proceso de mezclado. Otros sistemas mantienen un **gráfico de versiones** que muestra las relaciones entre las versiones. Siempre que una versión $v_1$ se origina copiando otra versión $v$, puede dibujarse un arco tendente desde $v$ a $v_1$. De forma parecida, si se mezclan dos versiones $v_2$ y $v_3$ para crear una nueva versión $v_4$, se trazan arcos tendentes desde $v_2$ y $v_3$ a $v_4$. El gráfico de versiones puede ayudar a los usuarios a entender las relaciones entre las distintas versiones, y el sistema puede utilizarlo internamente para administrar la creación y la eliminación de versiones.

Cuando se aplica el versionado a objetos complejos, surgen otros problemas que deben resolverse. Un objeto complejo, como lo es un sistema software, puede constar de muchos módulos. Cuando está permitido el versionado, cada uno de esos módulos puede tener varias versiones diferentes y un gráfico de versiones. Una **configuración** del objeto complejo es una colección consistente en una versión de cada módulo planificada de tal forma que las versiones del módulo de la configuración son compatibles y, juntas, forman una versión válida del objeto complejo. Una nueva versión o configuración del objeto complejo no tiene que incluir las versiones nuevas de cada módulo. Por tanto, determinadas versiones de módulos que no han cambiado pueden pertenecer a más de una configuración del objeto complejo. Una configuración es una colección de versiones de *diferentes* objetos que, juntas, constituyen un objeto complejo, mientras que el gráfico de versiones describe las versiones del *mismo* objeto. Una configuración debe obedecer la estructura de tipos de un objeto complejo; varias configuraciones del mismo objeto complejo son análogas a varias versiones de un objeto componente.

## 20.7 Resumen

En este capítulo hemos explicado los conceptos de la metodología de orientación de objetos aplicada a los sistemas de bases de datos, que se propuso para satisfacer las necesidades de las aplicaciones de bases de datos complejas y para añadir funcionalidad de bases de datos a los lenguajes de programación orientados a objetos, como C++. Hemos visto los conceptos más importantes que se utilizan en las bases de datos OO, entre los que podemos citar los siguientes:

- **Identidad del objeto.** Los objetos tienen identidades únicas que son independientes de sus valores de atributo.
- **Constructores de tipos.** Las estructuras de los objetos complejos se pueden construir aplicando recursivamente un conjunto de constructores básicos (tupla, conjunto, lista y bolsa).
- **Encapsulamiento de operaciones.** Tanto la estructura del objeto como las operaciones que pueden aplicarse a los objetos están incluidas en las definiciones de clase del objeto.
- **Compatibilidad con el lenguaje de programación.** Los objetos persistentes y transitorios se manipulan sin problemas. Los objetos se hacen persistentes adjuntándolos a una colección persistente o denominándolos explícitamente.
- **Jerarquía de tipos y herencia.** Los tipos de objetos pueden especificarse utilizando una jerarquía de tipos, que permite heredar los atributos y los métodos de los tipos definidos anteriormente. En algunos modelos se permite la herencia múltiple.

- **Extensiones.** Todos los objetos persistentes de un tipo particular se pueden almacenar en una extensión. Las extensiones correspondientes a una jerarquía de tipos tienen implementadas restricciones de conjunto/subconjunto.
- **Soporte de objetos complejos.** Los objetos complejos estructurados y no estructurados se pueden almacenar y manipular.
- **Polimorfismo y sobrecarga del operador.** Los nombres de los métodos y las operaciones se pueden sobrecargar para que puedan aplicarse a diferentes tipos de objetos con implementaciones distintas.
- **Versionado.** Algunos sistemas OO ofrecen soporte para conservar varias versiones del mismo objeto.

En el siguiente capítulo veremos cómo se materializan algunos de estos conceptos en el estándar ODMG.

## Preguntas de repaso

**20.1.** ¿Cuál es el origen de la metodología de la orientación a objetos?

**20.2.** ¿Cuáles son las principales características que debe poseer un OID?

**20.3.** Explique los distintos constructores de tipos. ¿Cómo se utilizan para crear estructuras de objeto complejas?

**20.4.** Explique el concepto de encapsulamiento y cómo se utiliza para crear tipos de datos abstractos.

**20.5.** Explique el significado de los siguientes términos en la terminología de las bases de datos orientadas a objetos: *método*, *firma*, *mensaje*, *colección*, *extensión*.

**20.6.** ¿Cuál es la relación entre un tipo y su subtipo en una jerarquía de tipos? ¿Cuál es la restricción que se implementa en las extensiones correspondientes a los tipos en la jerarquía de tipos?

**20.7.** ¿Cuál es la diferencia entre objetos persistentes y transitorios? ¿Cómo se manipula la persistencia en los sistemas de bases de datos OO típicos?

**20.8.** ¿En qué se diferencian la herencia normal, la herencia múltiple y la herencia selectiva?

**20.9.** Explique el concepto de polimorfismo/sobrecarga del operador.

**20.10.** ¿Cuál es la diferencia entre objetos complejos estructurados y objetos complejos no estructurados?

**20.11.** ¿Cuál es la diferencia entre la semántica de propiedad y la semántica de referencia en los objetos complejos estructurados?

**20.12.** ¿Qué es el versionado? ¿Por qué es importante? ¿Cuál es la diferencia entre versiones y configuraciones?

## Ejercicios

**20.13.** Convierta el ejemplo de OBJETO_GEOMÉTRICO ofrecido en la Sección 20.4.1 de la notación funcional a la notación que aparece en la Figura 20.3, que distingue entre atributos y operaciones. Utilice la palabra clave INHERIT para indicar que una clase hereda de otra clase.

**20.14.** Compare la herencia en el modelo EER (consulte el Capítulo 4) con la herencia en el modelo OO que se describe en la Sección 20.4.

**20.15.** Consideremos el esquema EER UNIVERSIDAD de la Figura 4.10. Piense en las operaciones que necesita para los tipos de entidad/clases del esquema. No tenga en cuenta las operaciones de constructor y destructor.

**20.16.** Considere el esquema ER EMPRESA de la Figura 3.2. Piense en las operaciones que necesita para los tipos de entidad/clases del esquema. No tenga en cuenta las operaciones de constructor y destructor.

## Bibliografía seleccionada

Los conceptos de las bases de datos orientadas a objetos son una amalgama de conceptos procedentes de los lenguajes de programación OO y de los sistemas de bases de datos y los modelos de datos conceptuales. Muchos libros describen los lenguajes de programación OO; por ejemplo, Stroustrup (1986) y Pohl (1991) están dedicados a C++, y Goldberg (1989) a Smalltalk. Libros más recientes, como Cattell (1994) y Lausen y Vossen (1997), describen los conceptos de las bases de datos OO. Otros libros sobre los modelos OO incluyen una descripción detallada del OODBMS experimental desarrollado en Microelectronic Computer Corporation, denominado ORION, y de los temas OO relacionados (Kim y Lochovsky [1989]). Bancilhon y otros (1992) describe la creación del OODBMS O2, explicando detalladamente las decisiones sobre diseño y la implementación del lenguaje. En Dogac y otros (1994), un grupo de expertos de la OTAN ofrecen una explicación minuciosa de los temas de bases de datos OO.

La bibliografía sobre las bases de datos OO es inmensa, por lo que sólo podemos ofrecer una muestra representativa. Una publicación del CACM de octubre de 1991 y otra del IEEE Cumputer de diciembre de 1990 describen los conceptos y los sistemas de bases de datos OO. Dittrich (1986) y Zaniolo y otros (1986) examinan los conceptos básicos de los modelos de datos OO. Una publicación más antigua sobre las bases de datos OO es Baroody y DeWitt (1981). Su y otros (1988) presenta un modelo de datos OO que se está utilizando en las aplicaciones CAD/CAM. Gupta y otros (1992) explica las aplicaciones OO para los entornos del CAD, la administración de redes y otras áreas. Mitschang (1989) extiende el álgebra relacional con el fin de abarcar los objetos complejos. Los lenguajes de consulta y las interfaces gráficas de usuario para OO se describen en Gyssens y otros (1990), Kim (1989), Alashqur y otros (1989), Bertino y otros (1992), Agrawal y otros (1990), y Cruz (1992).

El Manifiesto de la orientación a objetos de Atkinson y otros (1989) es un interesante artículo que informa de la posición de un grupo de expertos respecto a las características obligatorias y opcionales que debe tener la administración de bases de datos OO. El polimorfismo en las bases de datos y en los lenguajes de programación OO se explica en Osborn (1989), Atkinson y Buneman (1987), y Danforth y Tomlinson (1988). La identidad de objeto se explica en Abiteboul y Kanellakis (1989). Los lenguajes de programación OO para las bases de datos se explican en Kent (1991). Las restricciones de objeto se explican en Delcambre y otros (1991) y en Elmasri y otros (1993). La autorización y la seguridad en las bases de datos OO se examinan en Rabitti y otros (1991) y en Bertino (1992).

Al final del Capítulo 21 encontrará más referencias de utilidad.

CAPÍTULO 21

# Estándares, lenguajes y diseño de bases de datos de objetos

Como explicamos al principio del Capítulo 8, es muy importante tener un estándar para un determinado tipo de sistema de bases de datos, porque proporciona soporte para la portabilidad de aplicaciones de bases de datos. La **portabilidad** se define generalmente como la capacidad de ejecutar un programa de aplicación en particular en diferentes sistemas con unas modificaciones mínimas en el propio programa. En el campo de las bases de datos de objetos,[1] la portabilidad permitiría que un programa escrito para acceder a un sistema gestor de bases de datos de objetos (ODBMS) pudiera acceder a otro paquete ODBMS siempre y cuando los dos paquetes soportaran fielmente el estándar. Para los usuarios de bases de datos esto es muy importante, porque normalmente el usuario se muestra receloso a invertir en una tecnología nueva si los distintos fabricantes no se adhieren a un estándar. A fin de ilustrar por qué es importante la portabilidad, supongamos que un usuario en particular invierte miles de dólares en crear una aplicación que se ejecuta en el producto de un determinado fabricante y que después, por alguna razón, ese producto le decepciona (por ejemplo, porque el rendimiento no satisface sus requisitos). Si la aplicación se escribió utilizando las estructuras estándar del lenguaje, es posible que el usuario pueda convertir la aplicación al producto de otro fabricante (que obedece los mismos estándares de lenguaje pero quizá ofrece un mejor rendimiento para la aplicación de ese usuario) sin tener que hacer grandes modificaciones que requieran tiempo y una inversión monetaria importante.

Una segunda ventaja potencial de tener y obedecer unos estándares es que ayuda a conseguir la interoperabilidad, que generalmente se refiere a la posibilidad de que una aplicación acceda a varios sistemas distintos. En lo relativo a las bases de datos, esto significa que el mismo programa de aplicación puede acceder a algunos datos almacenados bajo un paquete ODBMS, y a otros datos almacenados bajo otro paquete. Hay diferentes niveles de interoperabilidad. Por ejemplo, los DBMSs pueden ser dos paquetes DBMS distintos del mismo tipo (por ejemplo, dos sistemas de bases de datos de objetos), o pueden ser dos paquetes DBMS de diferentes tipos (por ejemplo, un DBMS relacional y otro de objetos). Una tercera ventaja de los estándares es que permiten a los clientes *comparar productos comerciales* más fácilmente, determinando las partes de los estándares que son soportadas por cada producto.

Como explicamos en la introducción del Capítulo 8, una de las razones del éxito de los DBMSs relacionales comerciales es el estándar SQL. La ausencia de un estándar para los ODBMSs durante varios años puede

[1] En este capítulo utilizaremos *bases de datos de objetos* en lugar de *bases de datos orientadas a objetos*, que es el término que utilizamos en el Capítulo 20, ya que en la terminología actual es lo más aceptado.

haber provocado que usuarios potenciales hayan dejado de convertirse a esta nueva tecnología. En consecuencia, un consorcio de fabricantes de ODBMS, denominado ODMG (*Object Data Management Group*), propuso un estándar que se conoce como ODMG-93 o estándar ODMG 1.0. En este capítulo describiremos la revisión ODMG 2.0. El estándar consta de siete partes: el **modelo de objeto**, el **lenguaje de definición de objetos** (**ODL**), el **lenguaje de consulta de objetos** (**OQL**) y las **vinculaciones** a los lenguajes de programación orientados a objetos. Se han especificado vinculaciones a varios lenguajes de programación orientados a objetos, como C++, Smalltalk y Java. Algunos fabricantes sólo ofrecen vinculaciones de lenguaje específicas, sin ofrecer capacidades completas de ODL y OQL. Describiremos el modelo de objeto ODMG en la Sección 21.1, ODL en la Sección 21.2, OQL en la Sección 21.3, y la vinculación con el lenguaje C++ en la Sección 21.4. Los ejemplos de cómo utilizar ODL, OQL y la vinculación con el lenguaje C++ utilizarán la base de datos UNIVERSIDAD del Capítulo 4. En nuestra descripción acataremos el modelo de objeto ODMG 2.0 tal como se describió en Cattell y otros (1997).[2] Es importante reseñar que muchas de las ideas que el modelo de objeto ODMG incorpora están basadas en dos décadas de investigación en modelado conceptual y bases de datos de objetos.

Siguiendo la descripción del modelo ODMG, describiremos una técnica para el diseño conceptual de una base de datos de objetos en la Sección 21.5. Explicaremos en qué se diferencian las bases de datos de objetos de las bases de datos relacionales y veremos cómo asignar un diseño de base de datos conceptual en el modelo EER a las sentencias ODL del modelo ODMG.

El lector puede omitir las Secciones 21.3 a 21.7 si desea una introducción menos detallada del tema.

# 21.1 Visión general del modelo de objeto del ODMG

El **modelo de objeto ODMG** es el modelo de datos en el que están basados el lenguaje de definición de objetos (ODL) y el lenguaje de consulta de objetos (OQL). De hecho, este modelo de objeto proporciona los tipos de datos, constructores de tipos y otros conceptos que pueden utilizarse en el ODL para especificar los esquemas de la base de datos de objetos. Por tanto, se pretendía ofrecer un modelo de datos estándar para las bases de datos de objetos, al igual que SQL describe un modelo de datos estándar para las bases de datos relacionales. También ofrece una terminología estándar en un campo en el que los mismos términos a veces se utilizan para describir conceptos diferentes. En este capítulo intentaremos obedecer la terminología ODMG. Muchos de los conceptos del modelo ODMG ya se explicaron en el Capítulo 20, y asumiremos que el lector ha leído las Secciones 20.1 a 20.5. Cuando sea preciso indicaremos cuándo la terminología ODMG difiere de la utilizada en el Capítulo 20.

## 21.1.1 Objetos y literales

Los objetos y los literales son los bloques constructivos básicos del modelo de objeto. La principal diferencia entre los dos es que un objeto tiene un identificador de objeto y un **estado** (o valor actual), mientras que un literal tiene un valor pero *no un identificador de objeto*.[3] En todo caso, el valor puede tener una estructura compleja. El estado de un objeto puede cambiar con el transcurso del tiempo modificando el valor del objeto. Un literal es básicamente un valor constante, posiblemente con una estructura compleja que no cambia.

Un **objeto** queda descrito por cuatro características: identificador, nombre, tiempo de vida (ciclo de vida) y estructura. El **identificador de objeto** es un identificador único para todo el sistema (u **Object_id**).[4] Todo

[2] La primera versión del modelo de objeto se publicó en 1993.

[3] Aquí utilizaremos indistintamente los términos *valor* y *estado*.

[4] Es equivalente al OID que mencionábamos en el Capítulo 20.

objeto debe tener un identificador de objeto. Además del Object_id, opcionalmente podemos asignar un **nombre** único a algunos objetos dentro de una base de datos particular (este nombre se puede utilizar para hacer referencia al objeto en un programa, y el sistema debe poder localizar el objeto dado ese nombre).[5] Obviamente, no todos los objetos individuales tienen nombres únicos. Normalmente, unos cuantos objetos, principalmente los que albergan colecciones de objetos de un determinado tipo de objeto (como las extensiones) tendrán un nombre. Estos nombres se utilizan como **puntos de entrada** a la base de datos; es decir, localizando esos objetos por su nombre único, el usuario puede localizar después otros objetos a los que se hace referencia desde esos objetos. Otros objetos importantes de la aplicación también pueden tener nombres únicos. Todos esos nombres, dentro de una base de datos en particular, deben ser únicos. El **tiempo de vida** de un objeto especifica si es un *objeto persistente* (es decir, un objeto de la base de datos) o un *objeto transitorio* (es decir, un objeto en un programa que se está ejecutando y que desaparece una vez que el programa termina). Por último, la **estructura** de un objeto especifica cómo se construye el objeto utilizando los constructores de tipos. La estructura especifica si un objeto es *atómico* o un *objeto colección*.[6] El término *objeto atómico* es diferente a la definición que dimos para *constructor atómico* en la Sección 20.2.2, y es bastante diferente a un literal atómico (que explicamos a continuación). En el modelo ODMG, un objeto atómico es cualquier objeto que no es una colección; por consiguiente, esto abarca los *objetos estructurados* creados con el constructor struct.[7] En la Sección 21.1.2 explicaremos los objetos colección y en la Sección 21.1.3 veremos los objetos atómicos. En primer lugar, vamos a definir el concepto de literal.

En un modelo de objeto, un **literal** es un valor que *no tiene* un identificador de objeto. Sin embargo, el valor puede tener una estructura simple o compleja. Hay tres tipos de literales: atómico, colección y estructurado. Los **literales atómicos**[8] corresponden a los valores de los tipos de datos básicos y están predefinidos. Los tipos de datos básicos del modelo de objeto son los números enteros largos, cortos y sin signo (que se especifican mediante las palabras clave **long**, **short**, **unsigned long**, **unsigned short** en ODL), los números normales y de doble precisión en coma flotante (**float**, **double**), los valores booleanos (**boolean**), los caracteres individuales (**char**), las cadenas de caracteres (**string**) y los tipos de enumeración (**enum**), además de otros. Los **literales estructurados** se corresponden aproximadamente con los valores que se construyen utilizando el constructor de tupla descrito en la Sección 20.2.2. Incluyen la fecha, el intervalo, la hora y la marca de tiempo como estructuras integradas (véase la Figura 21.1[b]), así como cualquier estructura de tipo definida por el usuario adicional que cada aplicación necesite.[9] Las estructuras definidas por el usuario se crean utilizando la palabra clave **struct** de ODL, al igual que en los lenguajes de programación C y C++. Los **literales de colección** especifican un valor que es una colección de objetos o valores, pero la propia colección no tiene un identificador de objeto. Las colecciones del modelo de objeto son **set**<*T*>, **bag**<*T*>, **list**<*T*> y **array**<*T*>, donde *T* es el tipo de objetos o valores de la colección.[10] Otro tipo colección es **dictionary**<*K, V*>, que es una colección de asociaciones <*K, V*>, donde cada *K* es una clave (un valor de búsqueda único) asociada a un valor *V*; esto puede utilizarse para crear un índice en una colección de valores.

La Figura 21.1 ofrece una perspectiva simplificada de los componentes básicos del modelo objeto. La notación de ODMG utiliza la interfaz de palabras clave donde nosotros habíamos utilizado las palabras clave type

[5] Es equivalente al mecanismo de denominación para la persistencia que describíamos en la Sección 20.3.

[6] En el modelo ODMG, los *objetos atómicos* no se corresponden con los objetos cuyos valores son tipos de datos básicos. Todos los valores básicos (enteros, reales, etcétera) se consideran *literales*.

[7] La construcción *struct* es equivalente al *constructor de tupla* del Capítulo 20.

[8] El uso de la palabra *atómico* en *literal atómico* es equivalente a la forma en que utilizamos el constructor atómico en la Sección 20.2.2.

[9] Las estructuras para fechas, intervalos, horas y marcas de tiempo se pueden utilizar para crear valores literales u objetos con identificadores.

[10] Son parecidos a los constructores de tipos correspondientes descritos en la Sección 20.2.2.

(*tipo*) y class (*clase*) en el Capítulo 20. De hecho, interfaz es un término más apropiado, ya que describe la interfaz de tipos de objetos (esto es, sus atributos, relaciones y operaciones visibles).[11] Normalmente, estas interfaces no son instanciables (es decir, no se crean objetos para la interfaz), pero sirven para definir operaciones que los objetos definidos por el usuario pueden heredar para una aplicación en particular. La palabra clave *clase* en el modelo de objeto está reservada para las declaraciones de clase especificadas por el usuario que forman un esquema de base de datos y se utilizan para crear objetos de aplicación. La Figura 21.1 es una versión simplificada del modelo de objeto. Si desea unas especificaciones completas, consulte Cattell y otros (1997). Describiremos las construcciones que aparecen en la Figura 21.1 como describimos el modelo de objeto.

En el modelo de objeto, todos los objetos heredan la interfaz básica de **Object**, que se muestra en la Figura 21.1(a). Por tanto, las operaciones básicas que todos los objetos heredan (a partir de la interfaz Object) son **copy** (crea una copia nueva del objeto), **delete** (elimina el objeto) y **same_as** (compara la identidad del objeto con otro objeto).[12] En general, las operaciones se aplican mediante la **notación de punto**. Por ejemplo, dado un objeto *O*, para compararlo con otro objeto *P*, tenemos que escribir:

*O*.same_as(*P* )

El resultado devuelto por esta expresión es booleano y sería verdadero si la identidad de *P* es igual que la de *O*, y falso en caso contrario. De forma parecida, para crear una copia *P* del objeto *O*, tendríamos que escribir:

*P* = *O*.copy()

Una alternativa a la notación de punto es la **notación de flecha**: *O*–>same_as(*P*) o *O*–>copy().

La herencia de tipo, que se utiliza para definir las relaciones tipo/subtipo, se especifica en el modelo de objeto utilizando la notación de los dos puntos (:), como en el lenguaje de programación C++. Por tanto, en la Figura 21.1 podemos ver que todas las interfaces, como Collection, Date y Time, heredan la interfaz Object básica. En el modelo de objeto, hay dos tipos principales de objetos: objetos colección, que describimos en la Sección 21.1.2, y objetos atómicos (y estructurados), que describimos en la Sección 21.1.3.

## 21.1.2 Interfaces integradas para los objetos colección

Cualquier **objeto colección** hereda la interfaz Collection básica de la Figura 21.1(c), que muestra las operaciones para todos los objetos colección. Dado un objeto colección *O*, la operación *O*.cardinality() devuelve el número de elementos de la colección. La operación *O*.is_empty() devuelve verdadero si la colección *O* está vacía, y falso en caso contrario. Las operaciones *O*.insert_element(*E*) y *O*.remove_element(*E*) insertan o eliminan un elemento *E* de la colección *O*. Por último, la operación *O*.contains_element(*E* ) devuelve verdadero si la colección *O* incluye el elemento *E*, y falso en caso contrario. La operación *I* = *O*.create_iterator() crea un **objeto iterador** *I* para el objeto colección *O*, que puede iterar por todos los elementos de la colección. La interfaz para los objetos iteradores también aparece en la Figura 21.1(c). La operación *I*.reset() establece el iterador al primer elemento de una colección (en el caso de una colección desordenada sería algún elemento arbitrario), e *I*.next_position() establece el iterador al siguiente elemento. La operación *I*.get_element() recupera el **elemento actual**, que es el elemento en el que actualmente está situado el iterador.

El modelo de objeto ODMG utiliza **excepciones** para informar de los errores o de condiciones particulares. Por ejemplo, la excepción ElementNotFound de la interfaz Collection surgiría si la operación *O*.remove_element(*E*) determinase que *E* no es un elemento de la colección *O*. La excepción NoMoreElements en la interfaz de iterador surgiría por parte de la operación *I*.next_position() si el iterador está posicionado actualmente en el último elemento de la colección, y por tanto no hay más elementos a los que el iterador pueda apuntar.

[11] Interfaz (*interface*) es la palabra clave que también se utiliza en el lenguaje de programación Java.

[12] Hay definidas operaciones adicionales para temas de *bloqueo* que no aparecen en la Figura 21.1. Los conceptos de bloqueo para las bases de datos los explicamos en el Capítulo 18.

**Figura 21.1.** Visión general de las definiciones de interfaz para parte del modelo de objeto ODMG. (a) La interfaz Object básica, heredada por todos los objetos. (b) Algunas interfaces estándar para los literales estructurados.

**(a)**
```
interface Object {
    . . .
    boolean     same_as(in object other_object);
    object      copy();
    void        delete();
};
```

**(b)**
```
interface Date : Object {
    enum        Weekday
                        {Domingo, Lunes, Martes, Miércoles, Jueves, Viernes, Sábado};
    enum        Month
                        {Enero, Febrero, Marzo, Abril, Mayo, Junio, Julio, Agosto, Septiembre,
                        Octubre, Noviembre, Diciembre};
    unsigned short      year();
    unsigned short      month();
    unsigned short      day();
    . . .
    boolean             is_equal(in date other_date);
    boolean             is_greater(in date other_date);
    . . . };

interface Time : Object {
    . . .
    unsigned short      hour();
    unsigned short      minute();
    unsigned short      second();
    unsigned short      millisecond();
    . . .
    boolean             is_equal(in time other_time);
    boolean             is_greater(in time other_time);
    . . .
    time                add_interval(in interval some_interval);
    time                subtract_interval(in interval some_interval);
    interval            subtract_time(in time other_time); };

interface Timestamp : Object {
    . . .
    unsigned short      year();
    unsigned short      month();
    unsigned short      day();
    unsigned short      hour();
    unsigned short      minute();
    unsigned short      second();
    unsigned short      millisecond();
    . . .
    timestamp plus(in interval some_interval);
    timestamp minus(in interval some_interval);
```

**Figura 21.1.** (*Continuación*) Visión general de las definiciones de interfaz para parte del modelo de objeto ODMG. (c) Definiciones de interfaz para los objetos colección.

**(b)** (*Continuación*)

```
    boolean             is_equal(in timestamp other_timestamp);
    boolean             is_greater(in timestamp other_timestamp);
    . . . };

  interface Interval : Object {
    unsigned short      day();
    unsigned short      hour();
    unsigned short      minute();
    unsigned short      second();
    unsigned short      millisecond();
    . . .
    interval            plus(in interval some_Interval);
    interval            minus(in Interval some_Interval);
    interval            product(in long some_value);
    interval            quotient(in long some_value);
    boolean             is_equal(in interval other_Interval);
    boolean             is_greater(in interval other_interval);
    . . . };
```

**(c)**

```
  interface Collection : Object {
    . . .
    exception           ElementNotFound{any element; };
    unsigned long       cardinality();
    boolean             is_empty();
    . . .
    boolean             contains_element(in any element);
    void                insert_element(in any element);
    void                remove_element(in any element)
                           raises(ElementNotFound);
    iterator            create_iterator(in boolean stable);
    . . . };

  interface Iterator {
    exception           NoMoreElements();
    . . .
    boolean             is_stable();
    boolean             at_end();
    void                reset();
    any                 get_element() raises(NoMoreElements);
    void                next_position() raises(NoMoreElements);
    . . . };

  interface set : Collection {
    set                 create_union(in set other_set);
    . . .
    boolean             is_subset_of(in set other_set);
    . . . };
```

**Figura 21.1.** (*Continuación*) Visión general de las definiciones de interfaz para parte del modelo de objeto ODMG. (c) Definiciones de interfaz para los objetos colección.

**(c)** (*Continuación*)

```
interface bag : Collection {
   unsigned long      occurrences_of(in any element);
   bag                create_union(in bag other_bag);
   . . . };

interface list : Collection {
   exception          Invalid_Index{unsigned_long index; );
   any                remove_element_at(in unsigned long position)
                        raises(InvalidIndex);
   any                retrieve_element_at(in unsigned long position)
                        raises(InvalidIndex);
   void               replace_element_at(in any element, in unsigned long position)
                        raises(InvalidIndex);
   void               insert_element_after(in any element, in unsigned long position)
                        raises(InvalidIndex);
   . . .
   void               insert_element_first(in any element);
   . . .
   any                remove-first-element() raises(InvalidIndex);
   . . .
   any                retrieve_first_element() raises(InvalidIndex);
   . . .
   list               concat(in list other_list);
   void               append(in list other_list);
};

interface array : Collection {
   exception          Invalid_Index{unsigned_long index; };
   any                remove_element_at(in unsigned long index)
                        raises(InvalidIndex);
   any                retrieve_element_at(in unsigned long index)
                        raises(InvalidIndex);
   void               replace_element_at(in unsigned long index, in any element)
                        raises(InvalidIndex);
   void               resize(in unsigned long new_size);
};

struct association {any key; any value; };

interface dictionary : Collection {
   exception          KeyNotFound{any key; };
   void               bind(in any key, in any value);
   void               unbind(in any key) raises(KeyNotFound);
   any                lookup(in any key) raises(KeyNotFound);
   boolean            contains_key(in any key);
};
```

Los objetos Collection están especializados en set, list, bag, array y dictionary, que heredan las operaciones de la interfaz Collection. Un tipo de objeto set<*T*> se puede utilizar para crear objetos tal que el valor del objeto *O* es un *conjunto cuyos elementos son de tipo T*. La interfaz Set incluye la operación adicional *P* = *O*.create_union(*S*) (véase la Figura 21.1[c]), que devuelve un nuevo objeto *P* de tipo set<*T*> que es la unión de los conjuntos *O* y *S*. Otras operaciones parecidas a create_union (no aparecen en la Figura 21.1[c]) son create_intersection(*S*) y create_difference(*S*). Entre las operaciones de comparación de conjuntos encontramos la operación *O*.is_subset_of(*S*), que devuelve verdadero si el objeto set *O* es un subconjunto de algún otro objeto set *S*, y devuelve falso en caso contrario. Operaciones parecidas (que no aparecen en la Figura 21.1[c]) son is_proper_subset_of(*S*), is_superset_of(*S*) e is_proper_superset_of(*S*). El tipo de objeto bag<*T*> permite elementos duplicados en la colección y también hereda la interfaz Collection. Tiene tres operaciones [create_union(b), create_intersection(b) y create_difference(b)] que devuelven todas ellas un objeto nuevo de tipo bag<*T*>. Por ejemplo, *P* = *O*.create_union(b) devuelve un objeto bag *P* que es la unión de *O* y *B* (conservando los duplicados). La operación *O*.occurrences_of(*E*) devuelve el número de ocurrencias duplicadas del elemento *E* en la bolsa *O*.

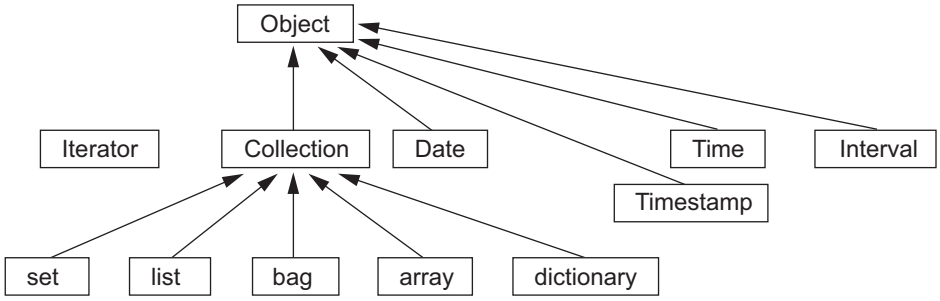
Un tipo de objeto list<*T*> hereda las operaciones Collection y puede utilizarse para crear colecciones en las que es importante el orden de los elementos. El valor de cada objeto *O* semejante es una *lista ordenada cuyos elementos son de tipo T*. Así pues, podemos referirnos al primer, último e *i*-ésimo elemento de la lista. Además, cuando añadimos un elemento a la lista, debemos especificar la posición de la misma donde queremos insertar el elemento. Algunas de las operaciones list se muestran en la Figura 21.1(c). Si *O* es un objeto de tipo list<*T*>, la operación *O*.insert_element_first(*E*) (véase la Figura 21.1[c]) inserta el elemento *E* por delante del primer elemento de la lista *O*, de modo que *E* se convierte en el primer elemento de la lista. Una operación parecida (no mostrada) es *O*.insert_element_last(*E*). La operación *O*.insert_element_after (*E*, *I*) de la Figura 21.1(c) inserta el elemento *E* después del *i*-ésimo elemento de la lista *O* y surgirá la excepción InvalidIndex si no existe el elemento *i*-ésimo en *O*. Una operación parecida (que tampoco se muestra) es *O*.insert_element_before(*E*, *I*). Para eliminar elementos de la lista, las operaciones son *E* = *O*.remove_first_element(), *E* = *O*.remove_last_element() y *E* = *O*.remove_element_at(*I*); estas operaciones eliminan de la lista el elemento indicado y devuelven el elemento como resultado de la operación. Hay otras operaciones para recuperar un elemento sin eliminarlo de la lista: *E* = *O*.retrieve_first_element(), *E* = *O*.retrieve_last_element() y *E* = *O*.retrieve_element_at(*I*). Por último, se definen dos operaciones para manipular listas: *P* = *O*.concat(*I*), que crea una lista *P* nueva como resultado de la concatenación de las listas *O* e *I* (los elementos de la lista *O* van seguidos por los de la lista *I*), y *O*.append(*I*), que añade los elementos de la lista *I* al final de la lista *O* (sin crear un nuevo objeto de lista).

El tipo de objeto array<*T*> también hereda las operaciones de Collection. Se parece a una lista, excepto que un array tiene una cantidad fija de elementos. Las operaciones específicas para un objeto array *O* son *O*.replace_element_at(*I*, *E*), que reemplaza el elemento del array que se encuentra en la posición *I* por el elemento *E*; *E* = *O*.remove_element_at(*I*), que recupera el elemento *i*-ésimo y lo reemplaza por un valor NULL; y *E* = *O*.retrieve_element_at(*I*), que simplemente recupera el elemento *i*-ésimo del array. Cualquiera de estas operaciones puede provocar la excepción InvalidIndex si *I* es mayor que el tamaño del array. La operación *O*.resize(*N*) cambia el número de elementos del array a *N*.

El último tipo de objetos colección es dictionary<*K*,*V*>, que permite la creación de una colección de pares <*K*,*V*> de asociación, donde todos los valores *K* (clave) son únicos. Esto permite la recuperación asociativa de un par particular dado su valor clave (parecido a un índice). Si *O* es un objeto colección de tipo dictionary<*K*,*V*>, entonces *O*.bind(*K*,*V*) vincula el valor v a la clave *K* como una asociación <*K*,*V*> en la colección, mientras que *O*.unbind(*K*) elimina de *O* la asociación con la clave *K*, y *V* = *O*.lookup(*K*) devuelve el valor *V* asociado con la clave *K* en *O*. Las dos últimas operaciones pueden provocar la excepción KeyNotFound. Por último, *O*.contains_key(*K*) devuelve verdadero si la clave *K* existe en *O*, y devuelve falso en caso contrario.

La Figura 21.2 es un diagrama que ilustra la jerarquía de herencia de los constructores integrados del modelo de objeto. Las operaciones se heredan del supertipo al subtipo. Las interfaces del objeto colección descritas anteriormente *no son directamente instanciables*; es decir, no podemos crear directamente objetos basados

**Figura 21.2.** Jerarquía de herencia para las interfaces integradas del modelo de objeto.




en esas interfaces. En cambio, podemos utilizar las interfaces para especificar objetos colección definidos por el usuario (de tipo set, bag, list, array o dictionary) para una aplicación de bases de datos particular. Cuando los usuarios diseñan el esquema de una base de datos, declararán sus propias interfaces de objeto y clases relativas a la aplicación de bases de datos. Si una interfaz o clase es uno de los objetos colección (por ejemplo, un conjunto), entonces heredará las operaciones de la interfaz set. Por ejemplo, en una aplicación de bases de datos UNIVERSIDAD, el usuario puede especificar una clase para set<ESTUDIANTE>, cuyos objetos serían conjuntos de objetos ESTUDIANTE. El programador puede utilizar entonces las operaciones para set<*T*> a fin de manipular un objeto de tipo set<ESTUDIANTE>. La creación de clases de aplicación normalmente se hace utilizando el lenguaje de definición de objetos ODL (consulte la Sección 21.2).

Es importante reseñar que todos los objetos de una colección concreta *deben ser del mismo tipo*. Por tanto, aunque la palabra clave any aparece en las especificaciones de las interfaces de colección en la Figura 21.1(c), esto no significa que dentro de la misma colección puedan entremezclarse objetos de cualquier tipo. Más bien, significa que podemos utilizar cualquier tipo para especificar el tipo de los elementos para una colección en particular (¡incluyendo otros tipos colección!).

### 21.1.3 Objetos atómicos (definidos por el usuario)

La sección anterior describió los tipos colección integrados del modelo de objeto. Ahora explicaremos cómo se pueden construir tipos objeto para los *objetos atómicos*. Se especifican mediante la palabra clave **class** en ODL. En el modelo de objeto, cualquier objeto definido por el usuario y que no es un objeto colección se denomina **objeto atómico**.[13]

Por ejemplo, en una aplicación de bases de datos UNIVERSIDAD, el usuario puede especificar un tipo de objeto (class) para los objetos ESTUDIANTE. La mayoría de esos objetos serán **objetos estructurados**; por ejemplo, un objeto ESTUDIANTE tendrá una estructura compleja, con muchos atributos, relaciones y operaciones, pero se le seguirá considerando atómico porque no es una colección. Semejante objeto atómico definido por el usuario se define como una clase especificando sus **propiedades** y sus **operaciones**. Las propiedades definen el estado del objeto y se distinguen además en **atributos** y **relaciones**. En esta subsección, ampliamos los tres tipos de componentes (atributos, relaciones y operaciones) que un tipo de objeto definido por el usuario para los objetos atómicos (estructurados) puede incluir. Ilustramos nuestra explicación con las clases EMPLEADO y DEPARTAMENTO de la Figura 21.3.

Un attribute (atributo) es una propiedad que describe algún aspecto de un objeto. Los atributos tienen valores (normalmente son literales que tienen una estructura sencilla o compleja) que están almacenados dentro del objeto. Sin embargo, los valores de los atributos también pueden ser Object_ids de otros objetos. Los valores de atributo pueden especificarse incluso a través de métodos que se utilizan para calcular el valor del atri-

[13] Como mencionamos anteriormente, esta definición de *objeto atómico* en el modelo de objeto ODMG es diferente de la definición de constructor atómico dado en el Capítulo 20, que es la definición utilizada en mucha de la literatura de bases de datos orientada a objetos.

**Figura 21.3.** Atributos, relaciones y operaciones en una definición de clase.

```
class EMPLEADO
(    extent          TODOS_EMPLEADOS
     key             Dni )
{
     attribute       string                  Nombre;
     attribute       string                  Dni;
     attribute       date                    FechaNac;
     attribute       enum Genero{H, M}       Sexo;
     attribute       short                   Edad;
     relationship    DEPARTAMENTO            TrabajaPara
                       inverse DEPARTAMENTO::Tiene_emps;
     void            reasignar_emp(in string Nuevo_nomdpto)
                       raises(nomdpto_no_valido);
};

class DEPARTAMENTO
(    extent          TODOS_DEPARTAMENTOS
     key             NombreDpto, NúmeroDpto )
{
     attribute       string                  NombreDpto;
     attribute       short                   NúmeroDpto;
     attribute       struct Dire_dpto {EMPLEADO Director, date FechaIngreso}
                                             Dire;
     attribute       set<string>             Ubicaciones;
     attribute       struct Proyectos {string Nombre_proy, time Horas_trabajo)
                                             Proyectos;
     relationship    set<EMPLEADO>           Tiene_emps inverse EMPLEADO::TrabajaPara;
     void            agregar_emp(in string Nuevo_nombreemp) raises(nombreemp_no_valido);
     void            cambiar_director(in string Nuevo_nombre_dire; in date FechaIngreso);
};
```

buto. En la Figura 21.3[14] los atributos para EMPLEADO son Nombre, Dni, FechaNac, Sexo y Edad, y los de DEPARTAMENTO son NombreDpto, NúmeroDpto, Dire, Ubicaciones y Proyectos. Los atributos Dire y Proyectos de DEPARTAMENTO tienen una estructura compleja y se definen mediante **struct**, que es equivalente al *constructor de tupla* del Capítulo 20. Por tanto, el valor de Dire en cada objeto DEPARTAMENTO tendrá dos componentes: Director, cuyo valor es un Object_id que hace referencia al objeto EMPLEADO que administra el DEPARTAMENTO, y FechaIngreso, cuyo valor es una fecha. El atributo Ubicaciones de DEPARTAMENTO se define a través de un constructor set, ya que cada objeto DEPARTAMENTO puede tener un conjunto de ubicaciones.

Una relationship (relación) es una propiedad que especifica que dos objetos de la base de datos están relacionados. En el modelo de objeto de ODMG, sólo las relaciones binarias (consulte el Capítulo 3) se representan explícitamente, y cada relación binaria se representa mediante un *par de referencias inversas* especificado mediante la relación de palabra clave. En la Figura 21.3 existe una relación que relaciona a cada EMPLEADO con el DEPARTAMENTO en el que trabaja (la relación TrabajaPara de EMPLEADO). En la dirección contraria, cada DEPARTAMENTO está relacionado con el conjunto de EMPLEADOS que trabajan en el DEPARTA-

[14] En la Figura 21.3 estamos utilizando la notación ODL, que explicaremos más en profundidad en la Sección 21.2.

MENTO (la relación Tiene_emps de DEPARTAMENTO). La palabra clave inverse indica que estas dos propiedades especifican una relación conceptual sencilla en direcciones inversas.[15]

Al especificar lo inverso, el sistema de bases de datos puede conservar automáticamente la integridad referencial de la relación. Es decir, si el valor de TrabajaPara para un EMPLEADO *E* concreto se refiere al DEPARTAMENTO *D*, entonces el valor de Tiene_emps para el DEPARTAMENTO *D* debe incluir una referencia a *E* en su conjunto de referencias EMPLEADO. Si el diseñador de la base de datos desea tener una relación que *sólo se represente en una dirección*, tiene que modelarla como un atributo (u operación). Un ejemplo es el componente Director del atributo Dire en DEPARTAMENTO.

Además de los atributos y las relaciones, el diseñador puede incluir **operaciones** en las especificaciones del tipo de objeto (clase). Cada tipo de objeto puede tener varias **firmas de operación**, que especifican el nombre de la operación, sus tipos de argumentos y el valor que devuelve, si es aplicable. Los nombres de las operaciones son únicos dentro de cada tipo de objeto, pero pueden sobrecargarse si el mismo nombre de operación aparece en distintos tipos de objeto. La firma de operación también puede especificar los nombres de las **excepciones** que pueden darse durante la ejecución de la operación. La implementación de la operación incluirá el código para alcanzar esas excepciones. En la Figura 21.3 la clase EMPLEADO tiene una operación, reasignar_emp, y la clase DEPARTAMENTO tiene dos, agregar_emp y cambiar_director.

### 21.1.4 Interfaces, clases y herencia

En el modelo de objeto ODMG, hay dos conceptos para especificar los tipos de objetos: interfaces y clases. Además, existen dos tipos de relaciones de herencia. En esta sección explicamos las diferencias y las similitudes entre estos dos conceptos. Siguiendo con la terminología ODMG, utilizamos la palabra **comportamiento** para referirnos a las *operaciones*, y **estado** para referirnos a las *propiedades* (atributos y relaciones).

Una **interfaz** es una especificación del comportamiento abstracto de un tipo de objeto, que especifica las firmas de operación. Aunque una interfaz puede tener propiedades de estado (atributos y relaciones) como parte de sus especificaciones, éstas no pueden ser heredadas de la interfaz, como veremos. Una interfaz es **no instanciable**, es decir, no podemos crear objetos que correspondan a una definición de interfaz.[16] Una **clase** es una especificación del comportamiento abstracto y del estado abstracto de un tipo de objeto, y es **instanciable** (es decir, podemos crear instancias de objetos individuales que es equivalente a la definición de una clase). Como las interfaces no son instanciables, se utilizan principalmente para especificar operaciones abstractas que las clases y otras interfaces pueden heredar. Es lo que se denomina **herencia de comportamiento** y se especifica con la notación de los dos puntos (:).[17] Por tanto, en el modelo de objeto ODMG, la herencia del comportamiento requiere que el supertipo sea una interfaz, mientras que el subtipo puede ser una clase u otra interfaz.

Otra relación de herencia, denominada EXTENDS y especificada con la palabra clave **extends**, se utiliza para heredar el estado y el comportamiento estrictamente entre clases. En una herencia de extensión, tanto el supertipo como el subtipo han de ser clases. La herencia múltiple a través de extends no está permitida. Sin embargo, la herencia múltiple sí está permitida para la herencia del comportamiento a través de la notación de los dos puntos (:). Por tanto, una interfaz puede heredar el comportamiento de otras interfaces. Una clase también puede heredar el comportamiento de varias interfaces a través de la notación de los dos puntos (:), además de heredar el comportamiento y el estado de, *a lo sumo*, una clase vía extends. En la Sección 21.2 ofreceremos ejemplos de cómo se pueden utilizar estas dos relaciones de herencia (":" y extends).

---

[15] El Capítulo 3 explica cómo se puede representar una relación mediante dos atributos en direcciones inversas.

[16] Es parecido al concepto de clase abstracta en el lenguaje de programación C++.

[17] El informe del ODMG también denomina a la herencia múltiple relaciones tipo/subtipo, es-un, y generalización/especialización, aunque en la literatura se han utilizado estos términos para describir la herencia del estado y de las operaciones (consulte los Capítulos 4 y 20).

### 21.1.5 Extensiones, claves y objetos factory

En el modelo de objeto ODMG, el diseñador de la base de datos puede declarar una *extensión* (utilizando la palabra clave **extent**) para cualquier tipo de objeto que se define mediante una declaración de **clase**. La extensión tiene un nombre y contendrá todos los objetos persistentes de la clase. En la Figura 21.3 las clases EMPLEADO y DEPARTAMENTO tienen extensiones denominadas TODOS_EMPLEADOS y TODOS_DEPARTAMENTOS, respectivamente. Esto es parecido a crear dos objetos (uno de tipo set<EMPLEADO> y otro de tipo set<DEPARTAMENTO>) y a hacerlos persistentes asignándoles los nombres TODOS_EMPLEADOS y TODOS_DEPARTAMENTOS. Las extensiones también se utilizan para implementar automáticamente la relación conjunto/subconjunto entre las extensiones de un supertipo y su subtipo. Si dos clases A y B tienen extents ALL_A y ALL_B, y la clase B es un subtipo de la clase A (es decir, la clase B extiende a la clase A), entonces la colección de objetos de ALL_B debe ser un subconjunto de los de ALL_A en cualquier punto. Esta restricción la implementa automáticamente el sistema de bases de datos.

Una clase con una extensión puede tener una o más claves. Una **clave** consta de una o más propiedades (atributos o relaciones) cuyos valores se restringen para ser únicos de cada objeto de la extensión. Por ejemplo, en la Figura 21.3 la clase EMPLEADO tiene el atributo Dni como clave (cada objeto EMPLEADO de la extensión debe tener un valor de Dni único), y la clase DEPARTAMENTO tiene dos claves distintas: NombreDpto y NúmeroDpto (cada DEPARTAMENTO debe tener un NombreDpto único y un NúmeroDpto único). En el caso de una clave compuesta[18] formada por varias propiedades, estas últimas deben ir entre paréntesis. Por ejemplo, si una clase VEHÍCULO con una extensión TODOS_VEHÍCULOS tiene una clave formada por una

**Figura 21.4.** Interfaces para ilustrar los objetos factory y los objetos de bases de datos.

```
interface ObjectFactory {
   Object              new();
};
interface DateFactory : ObjectFactory {
   exception           InvalidDate{};
. . .
   date                calendar_date( in unsigned short year,
in unsigned short month,
in unsigned short day)
            raises(InvalidDate);
. . .
   date                          current();
};
interface DatabaseFactory {
Database                         new();
};
interface Database {
   void                open(in string database_name);
   void                close();
   void                bind(in any some_object, in string object_name);
   object              unbind(in string name);
   object              lookup(in string object_name)
raises(ElementNotFound);
. . . };
```

[18] En el informe del ODMG se habla de clave compuesta (*compound key*).

combinación de dos atributos, Cifras y Letras, tendrá que colocarse entre paréntesis en la declaración de la clave: (Cifras, Letras).

A continuación, presentamos el concepto de **objeto factory**, un objeto que puede utilizarse para generar o crear objetos individuales a través de sus operaciones. En la Figura 21.4 se muestran algunas de las interfaces de los objetos factory que forman parte del modelo de objeto ODMG. La interfaz ObjectFactory tiene una sola operación, new(), que devuelve un objeto nuevo con un Object_id. Heredando esta interfaz, los usuarios pueden crear sus propias interfaces factory para cada tipo de objeto definido por el usuario (atómico), y el programador puede implementar la operación *nueva* de modo diferente para cada tipo de objeto. La Figura 21.4 también muestra una interfaz DateFactory, que tiene operaciones adicionales para crear una nueva fecha de calendario (calendar_date) y un objeto cuyo valor es la fecha actual (current_date), además de otras operaciones que no aparecen en la figura. Como podemos ver, un objeto factory proporciona básicamente las **operaciones de constructor** para los objetos nuevos.

Por último, explicamos el concepto de **database**. Como un ODBMS puede crear muchas bases de datos diferentes, cada una con su propio esquema, el modelo de objeto ODMG tiene interfaces para los objetos DatabaseFactory y Database (véase la Figura 21.4). Cada base de datos tiene un *nombre de base de datos* propio, y es posible utilizar la operación **bind** para asignar nombres únicos individuales a los objetos persistentes de una base de datos concreta. La operación **lookup** devuelve el objeto de la base de datos cuyo nombre es el especificado con object_name, y la operación **unbind** elimina el nombre de un objeto nombrado persistente de la base de datos.

# 21.2 El lenguaje de definición de objetos ODL

Después de la introducción del modelo de objeto ODMG de la sección anterior, ahora veremos cómo podemos utilizar esos conceptos para crear un esquema de base de datos de objetos utilizando el lenguaje de definición de objetos ODL.[19]

ODL está diseñado para dar soporte a las construcciones semánticas del modelo de objeto ODMG y es independiente de cualquier otro lenguaje de programación. Se utiliza principalmente para crear especificaciones de objetos (es decir, clases e interfaces). Por tanto, ODL no es un lenguaje de programación completo. Un usuario puede especificar en ODL un esquema de base de datos independientemente de cualquier lenguaje de programación, y después utilizar las vinculaciones con lenguajes concretos para especificar cómo pueden asignarse las construcciones de ODL a las construcciones de esos lenguajes de programación concretos, como C++, Smalltalk y Java. En la Sección 21.4 ofrecemos una panorámica de la vinculación con C++.

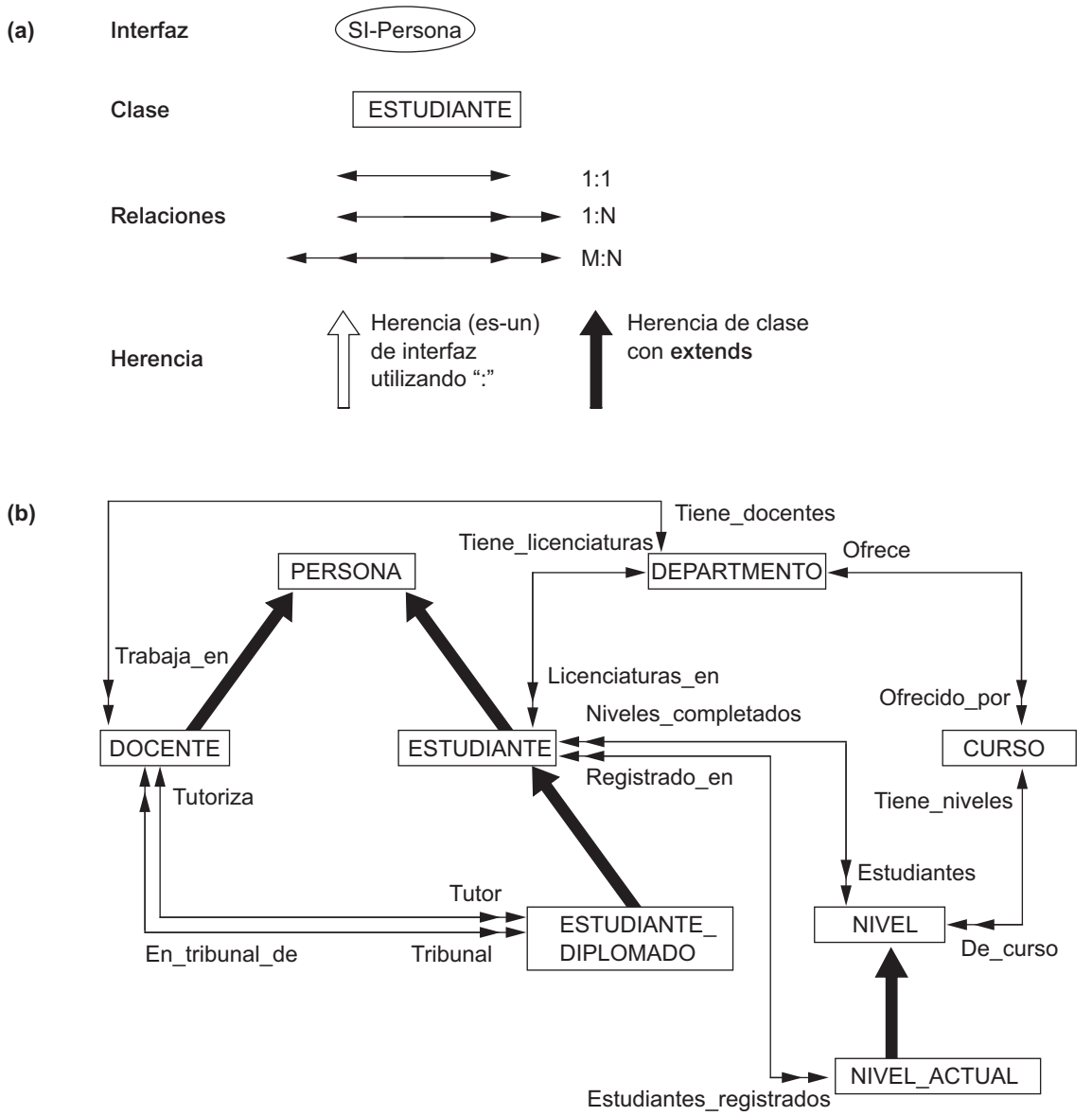
La Figura 21.5(b) muestra un posible esquema de objeto para parte de la base de datos UNIVERSIDAD, que presentamos en el Capítulo 4. Describiremos los conceptos de ODL utilizando este ejemplo y el de la Figura 21.7. La notación gráfica de la Figura 21.5(b) se muestra en la Figura 21.5(a) y puede considerarse como una variación de los diagramas EER (consulte el Capítulo 4) con el concepto añadido de herencia de interfaz, pero sin otros conceptos EER, como las categorías (tipos de unión) y los atributos de relaciones.

La Figura 21.6 muestra un posible conjunto de definiciones de clases ODL para la base de datos UNIVERSIDAD. En general, puede haber varias asignaciones posibles de un diagrama de esquema de objeto (o diagrama de esquema EER) en clases ODL. Explicaremos estas opciones en la Sección 21.5.

La Figura 21.6 muestra la forma directa de mapear parte de la base de datos UNIVERSIDAD del Capítulo 4. Los tipos de entidad son mapeados en clases ODL, y la herencia se realiza con EXTENDS. Sin embargo, no hay una forma directa de mapear categorías (tipos de unión) o de conseguir la herencia múltiple. En la Figura 21.6 las clases PERSONA, DOCENTE, ESTUDIANTE y ESTUDIANTE_DIPLOMADO tienen las extensiones

[19] La sintaxis y los tipos de datos de ODL pretenden ser compatibles con el lenguaje de definición de interfaz (IDL, *Interface Definition language*) de CORBA (*Common Object Request Broker Architecture*), con extensiones para las relaciones y otros conceptos de bases de datos.

**Figura 21.5.** Ejemplo de un esquema de base de datos. (a) Notación gráfica para representar los esquemas ODL. (b) Esquema gráfico de base de datos de objetos para parte de la base de datos UNIVERSIDAD (no se muestran las clases NOTA y TÍTULO).




PERSONAS, DOCENTE, ESTUDIANTES y ESTUDIANTES_GRADUADOS, respectivamente. DOCENTE y ESTUDIANTE extienden a PERSONA y ESTUDIANTE_DIPLOMADO extiende ESTUDIANTE. Por tanto, la colección de ESTUDIANTES (y la colección de DOCENTE) se restringirán para ser en cualquier momento un subconjunto de la colección de PERSONAs. De forma parecida, la colección ESTUDIANTE_DIPLOMADO será un subconjunto de ESTUDIANTEs. Al mismo tiempo, los objetos individuales ESTUDIANTE y DOCENTE heredarán las propiedades (atributos y relaciones) y las operaciones de PERSONA, y los objetos ESTUDIANTE_DIPLOMADO individuales heredarán los de ESTUDIANTE.

**Figura 21.6.** Posible esquema ODL para la base de datos UNIVERSIDAD de la Figura 21.5(b).

```
class PERSONA
( extent      PERSONAS
  key         Dni )
{ attribute   struct NomPers {    string  NombreP,
                                  string  Apellido1,
                                  string  Apellido2 } Nombre;
  attribute   string                                  Dni;
  attribute   date                                    FechaNac;
  attribute   enum Género{H, M}                       Sexo;
  attribute   struct Dirección {  short   No,
                                  string  Calle,
                                  short   NApt,
                                  string  Ciudad,
                                  string  Prov,
                                  short   CP }        Dirección;
  short       Edad();       };

class DOCENTE extends PERSONA
( extent                  DOCENTE )
{ attribute   string              Rango;
  attribute   float               Sueldo;
  attribute   string              Oficina;
  attribute   string              TlfOficina;
  relationship DEPARTAMENTO       Trabaja_en inverse DEPARTAMENTO::Tiene_docentes;
  relationship set<ESTUDIANTE_DIPLOMADO> Tutoriza inverse ESTUDIANTE_DIPLOMADO::Tutor;
  relationship set<ESTUDIANTE_DIPLOMADO> En_tribunal_de inverse
              ESTUDIANTE_DIPLOMADO::Tribunal;
  void                        aumentar_sueldo(in float aumento);
  void                        promocionar(in string nuevo_rango); };

class NOTA
( extent      NOTAS )
{
  attribute   enum ValoresNotas{A,B,C,D,F,I, P} Nota;
  relationship NIVEL Nivel inverse NIVEL::Estudiantes;
  relationship ESTUDIANTE Estudiante inverse ESTUDIANTE::Niveles_completados; };

class ESTUDIANTE extends PERSONA
( extent      ESTUDIANTES )
{ attribute   string              Clase;
  attribute   Department          Diplomatura_en;
  relationship Departamento Licenciaturas_en inverse DEPARTAMENTO::Tiene_licenciaturas;
  relationship set<NOTA> Niveles_completados inverse NOTA::Estudiante;
  relationship set<NIVEL_ACTUAL> Registrado_en inverse NIVEL_ACTUAL::Estudiantes_registrados;
  void                        cambiar_licenciatura(in string nombred) raises(nomdpto_no_válido);
  float                       media();
```

**Figura 21.6.** (*Continuación*).

```
   void                    registrar(in short numnivel) raises(nivel_no_válido);
   void                    asignar_nota(in short numnivel; in ValorNota nota)
                                raises(nivel_no_válido,nota_no_válida); };

class GRADO
{  attribute    string              Facultad;
   attribute    string              Título;
   attribute    string              Año; };

class ESTUDIANTE_DIPLOMADO extends ESTUDIANTE
(  extent       ESTUDIANTES_GRADUADOS )
{  attribute set<Título>            Títulos;
   relationship Tutor_facultad inverse DOCENTE::Tutoriza;
   relationship set<DOCENTE> Tribunal inverse DOCENTE::En_tribunal_de;
   void                    asignar_tutor(in string Apellido2; in string NombreP)
                                raises(facultad_no_válida);
   void                    asignar_miembro_tribunal(in string Apellido2; in string NombreP)
                                raises(facultad_no_válida); };

class DEPARTAMENTO
(  extent       DEPARTAMENTOS
   key          NombreDpto )
{  attribute    string              NombreDpto;
   attribute    string              TlfDpto;
   attribute    string              TlfOfic;
   attribute    string              Facultad;
   attribute    DOCENTE             Director;
   relationship set<DOCENTE> Tiene_facultad inverse DOCENTE::Trabaja_en;
   relationship set<ESTUDIANTE> Tiene_licenciaturas inverse ESTUDIANTE::Licenciaturas_en;
   relationship set<CURSO> Ofrece inverse CURSO::Ofrecido_por; };

class CURSO
(  extent       CURSOS
   key          NumCurso )
{  attribute    string              NombreCurso;
   attribute    string              NumCurso;
   attribute    string              Descripción;
   relationship set<NIVEL> Tiene_niveles inverse NIVEL::De_curso;
   relationship <DEPARTAMENTO> Ofrecido_por inverse DEPARTAMENTO::Ofrece; };

class NIVEL
(  extent       NIVELES )
{  attribute    short               NumNivel;
   attribute    string              Año;
   attribute    enum Trimestre{Otoño, Invierno, Primavera, Verano}      Trim;
   relationship set<Nota> Estudiantes inverse Nota::Nivel;
   relationship CURSO De_curso inverse CURSO::Tiene_niveles; };
```

```
class NIVEL_ACTUAL extends NIVEL
( extent      NIVELES_ACTUALES )
{ relationship set<ESTUDIANTE> Estudiantes_registrados
                    inverse ESTUDIANTE::Registrado_en
  void        registrar_estudiante(in string Dni)
                    raises(estudiante_no_válido, nivel_lleno); };
```

Las clases DEPARTAMENTO, CURSO, NIVEL y NIVEL_ACTUAL de la Figura 21.6 son mapeos directos de los correspondientes tipos de entidad de la Figura 21.5(b). Sin embargo, la clase NOTA requiere algo de explicación. Esta clase corresponde a la relación M:N entre ESTUDIANTE y NIVEL de la Figura 21.5(b). La razón de hacerlo en una clase independiente (en lugar de hacerlo como un par de relaciones inversas) es porque incluye el atributo de relación Nota.[20]

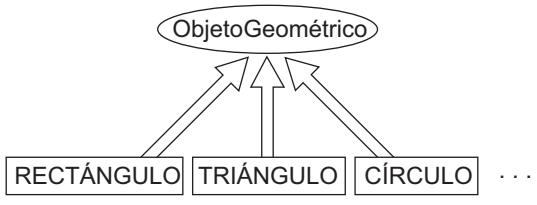
Por tanto, la relación M:N se mapea a la clase NOTA, y un par de relaciones 1:N, una entre ESTUDIANTE y NOTA y otra entre NIVEL y NOTA.[21] Estas relaciones se representan con las siguientes propiedades de relación: Niveles_completados de ESTUDIANTE; Nivel y Estudiante de NOTA; y Estudiantes de NIVEL (véase la Figura 21.6). Por último, la clase GRADO se utiliza para representar los grados del atributo compuesto y multivalor de ESTUDIANTE_DIPLOMADO (véase la Figura 4.10).

Como el ejemplo anterior no incluye interfaces, sólo clases, vamos a utilizar ahora un ejemplo diferente para ilustrar las interfaces y la herencia de interfaz (comportamiento). La Figura 21.7(a) es parte de un esquema de base de datos para almacenar objetos geométricos. Especificamos una interfaz ObjetoGeométrico, con operaciones para calcular el perímetro y el área de un objeto geométrico, además de operaciones para trasladar (mover) y rotar un objeto. Varias clases (RECTÁNGULO, TRIÁNGULO, CÍRCULO, ...) heredan la interfaz ObjetoGeométrico. Como ObjetoGeométrico es una interfaz, es *no instanciable* (es decir, no podemos crear objetos basándonos directamente en esta interfaz). No obstante, podemos crear objetos de tipo RECTÁNGULO, TRIÁNGULO, CÍRCULO, ..., y estos objetos heredarán todas las operaciones de la interfaz ObjetoGeometrico. Con la herencia de interfaz, sólo se heredan las operaciones, no las propiedades (atributos, relaciones). Por tanto, si en la herencia de clase se necesita una propiedad, debe repetirse en la definición de clase, como con el atributo PuntoReferencia de la Figura 21.7(b). Las operaciones heredadas pueden tener implementaciones diferentes en cada clase. Por ejemplo, las implementaciones de las operaciones de área y perímetro pueden ser diferentes para RECTÁNGULO, TRIÁNGULO y CÍRCULO.

La *herencia múltiple* de interfaces por parte de una clase está permitida, así como la herencia múltiple de interfaces por parte de otra interfaz. Sin embargo, con la herencia EXTENDS (clase), la herencia múltiple *no está permitida*. Por tanto, una clase puede heredar a través de EXTENDS a lo sumo de una clase (además de heredar de ninguna o más interfaces).

**Figura 21.7.** Ilustración de una herencia de interfaz a través de ":". (a) Representación gráfica del esquema.

**(a)**




[20] Explicaremos los mapeados alternativos para atributos de relaciones en la Sección 21.5.

[21] Esto es parecido a cómo se mapea una relación M:N en el modelo relacional (consulte el Capítulo 7).

**Figura 21.7.** (*Continuación*) (b) Definiciones de interfaz y clase en ODL.

**(b)**
```
interface ObjetoGeométrico
{   attribute   enum        Forma{RECTÁNGULO, TRIÁNGULO, CÍRCULO, . . . } Forma;
    attribute   struct      Punto{short x, short y}     PuntoReferencia;
    float       perimetro();
    float       area();
    void        trasladar(in short mover_x; in short mover_y);
    void        rotar(in float angulo_de_rotacion); };

class RECTÁNGULO : ObjetoGeometrico
(   extent      RECTÁNGULOS)
{   attribute   struct      Punto {short x, short y}    PuntoReferencia;
    attribute   short Longitud;
    attribute   short Altura;
    attribute   float  Angulo_orientacion; };

class TRIÁNGULO : ObjetoGeometrico
(   extent      TRIÁNGULOS )
{   attribute   struct      Punto {short x, short y} PuntoReferencia;
    attribute   short Lado_1;
    attribute   short Lado_2;
    attribute   float  Lado1_lado2_angulo;
    attribute   float  Lado1_angulo_orientacion; };

class CÍRCULO : ObjetoGeométrico
(   extent      CÍRCULOS )
{   attribute   struct      Punto {short x, short y} PuntoReferencia;
    attribute   short Radio; };
. . .
```

# 21.3 El lenguaje de consulta de objetos OQL

El lenguaje de consulta de objetos (OQL) es el lenguaje de consulta propuesto por el modelo de objeto ODMG. Está diseñado para trabajar estrechamente con los lenguajes de programación para los que se ha definido una vinculación ODMG, como C++, Smalltalk y Java. Por tanto, una consulta OQL incrustada en uno de estos lenguajes de programación puede devolver objetos que coincidan con el sistema de tipos de ese lenguaje. Además, las implementaciones de las operaciones de clase en un esquema ODMG pueden tener su propio código escrito en dichos lenguajes de programación. La sintaxis de OQL para las consultas es parecida a la sintaxis del lenguaje de consulta estándar (SQL) relacional, con algunas características añadidas para los conceptos ODMG, como la identidad de objeto, los objetos complejos, las operaciones, la herencia, el polimorfismo y las relaciones.

En la Sección 21.3.1 explicaremos la sintaxis de las consultas OQL sencillas y el concepto de utilizar objetos con nombre o extensiones como los puntos de entrada a la base de datos. Después, en la Sección 21.3.2 veremos la estructura del resultado de una consulta y el uso de las expresiones de ruta para recorrer las relaciones entre los objetos. Para la Sección 21.3.3 dejamos otras características de OQL encaminadas a manipular la identidad de los objetos, la herencia, el polimorfismo y otros conceptos de orientación a objetos. Los ejemplos para ilustrar las consultas OQ están basados en el esquema de base de datos UNIVERSIDAD de la Figura 21.6.

### 21.3.1 Consultas OQL sencillas, puntos de entrada a la base de datos y variables iteradoras

La sintaxis básica de OQL es una estructura select ... from ... where ..., como en SQL. Por ejemplo, la consulta para recuperar los nombres de todos los departamentos de la universidad de ingeniería se puede escribir de este modo:

```
C0:   select D.NombreDpto
      from   D in DEPARTAMENTOS
      where      D.Facultad = 'Ingeniería';
```

En general, necesitamos un **punto de entrada** a la base de datos para cada consulta, que puede ser cualquier *objeto persistente con nombre*. Para muchas consultas, el punto de entrada es el nombre de la extent de una clase. Recuerde que el nombre de extensión está considerado el nombre de un objeto persistente cuyo tipo es una colección (en muchos casos, un conjunto) de objetos de la clase. Mirando los nombres de extensión de la Figura 21.6, el objeto con nombre DEPARTAMENTOS es de tipo set<DEPARTAMENTO>; PERSONAS es de tipo set<PERSONA>; DOCENTE es de tipo set<DOCENTE>; etcétera.

El uso de un nombre de extensión (DEPARTAMENTOS en C0) como punto de entrada se refiere a una colección persistente de objetos. Siempre que en una consulta OQL se hace referencia a una colección, debemos definir una **variable iteradora**[22] (*D* en C0) que recorra los objetos de la colección. En muchos casos, como en C0, la consulta seleccionará ciertos objetos de la colección, basándose en las condiciones especificadas en la cláusula where. En C0, sólo los objetos persistentes *D* de la colección de DEPARTAMENTOS que satisfacen la condición *D*.Facultad = 'Ingeniería' son seleccionados para el resultado de la consulta. Para cada objeto seleccionado *D*, el valor de *D*.NombreDpto es recuperado en el resultado de la consulta. Por tanto, el *tipo del resultado* para C0 es bag<string> porque el tipo de cada valor de NombreDpto es string (aunque el resultado real es un conjunto porque NombreDpto es un atributo clave). En general, el resultado de una consulta sería de tipo bag para select ... from ... y de tipo set para select distinct ... from ..., como en SQL (al añadir la palabra clave distinct se eliminan los duplicados).

Continuando con el ejemplo C0, hay tres opciones sintácticas para especificar las variables iteradoras:

```
D IN DEPARTAMENTOS
DEPARTAMENTOS D
DEPARTAMENTOS AS D
```

Utilizaremos la primera estructura en nuestros ejemplos.[23]

Los objetos con nombre utilizados como puntos de entrada para las consultas OQL no están limitados a los nombres de extensiones. Es posible utilizar cualquier objeto persistente con nombre, tanto si se refiere a un objeto atómico (sencillo) como a un objeto colección, como punto de entrada a la base de datos.

### 21.3.2 Resultados de la consulta y expresiones de ruta

En general, el resultado de una consulta puede ser de cualquier tipo que pueda expresarse en el modelo de objeto ODMG. Una consulta no tiene que obedecer la estructura select ... from ... where ...; en el caso más sencillo, cualquier nombre persistente ya es una consulta de por sí, cuyo resultado es una referencia a ese objeto persistente. Por ejemplo, la siguiente consulta:

```
C1:   DEPARTAMENTOS;
```

---

[22] Esto es parecido a las variables de tupla que recorren las tuplas en las consultas SQL.

[23] Observe que las dos últimas opciones son parecidas a la sintaxis para especificar las variables de tupla en las consultas SQL.

devuelve una referencia a la colección de objetos DEPARTAMENTO persistentes, cuyo tipo es set<DEPARTAMENTO>. De forma parecida, supongamos que hubiéramos dado (a través de la operación de vinculación de base de datos; véase la Figura 21.4) un nombre persistente CC_DEPARTAMENTO a un objeto DEPARTAMENTO sencillo (el departamento de ciencias de la computación); en este caso, la consulta:

**C1A:** CC_DEPARTAMENTO;

devuelve una referencia a ese objeto individual de tipo DEPARTAMENTO. Una vez especificado un punto de entrada, el concepto de **expresión de ruta** se puede utilizar para especificar una *ruta de acceso* a los atributos y los objetos relacionados. Una expresión de ruta normalmente empieza con el *nombre de un objeto persistente*, o con la variable iteradora que recorre los objetos individuales de una colección. Este nombre irá seguido por ninguno o más nombres de relación o nombres de atributo conectados mediante un punto. Por ejemplo, y siguiendo con el ejemplo de la base de datos UNIVERSIDAD de la Figura 21.6, los siguientes son ejemplos de expresiones de ruta, que también son consultas válidas en OQL:

**C2:** CC_DEPARTAMENTO.Director;
**C2A:** CC_DEPARTAMENTO.Director.Rango;
**C2B:** CC_DEPARTAMENTO.Tiene_docentes;

La primera expresión, C2, devuelve un objeto de tipo DOCENTE, porque es el tipo del atributo Director de la clase DEPARTAMENTO. Será una referencia al objeto DOCENTE que está relacionado con el objeto DEPARTAMENTO cuyo nombre persistente es CC_DEPARTAMENTO a través del atributo Director; es decir, una referencia al objeto DOCENTE que es la persona que dirige el departamento de ciencias de la computación. La segunda expresión, C2A, es parecida excepto que devuelve el Rango de este objeto DOCENTE (director de informática) en lugar de la referencia al objeto; por tanto, el tipo devuelto por C2A es una cadena, que es el tipo de datos para el atributo Rango de la clase DOCENTE.

Las expresiones de ruta C2 y C2A devuelven valores sencillos, porque los atributos Director (de DEPARTAMENTO) y Rango (de DOCENTE) son de un solo valor y se aplican a un solo objeto. La tercera expresión, C2B, es diferente; devuelve un objeto de tipo set<DOCENTE> incluso cuando se aplica a un solo objeto, porque es el tipo de relación Tiene_docentes de la clase DEPARTAMENTO. La colección devuelta incluirá referencias a todos los objetos DOCENTE que están relacionados con el objeto DEPARTAMENTO cuyo nombre persistentes es CC_DEPARTAMENTO a través de la relación Tiene_docentes; es decir, referencias a todos los objetos DOCENTE que están trabajando en el departamento de ciencias de la computación. Ahora, para devolver los rangos de los docentes de ciencias de la computación, no podemos escribir:

**C3':** CC_DEPARTAMENTO.Tiene_docentes.Rango;

porque no está claro si el objeto devuelto sería de tipo set<string> o de tipo bag<string> (es más probable que sea de este último tipo, porque varios profesores pueden compartir el mismo rango). Debido a este problema de ambigüedad, OQL no permite expresiones como las de C3'. En su lugar, es preciso utilizar una variable iteradora sobre estas colecciones, como en C3A o C3B:

**C3A:** **select** *F*.Rango
**from** *F* **in** CC_DEPARTAMENTO.Tiene_docentes;
**C3B**: **select distinct** *F*.Rango
**from** *F* **in** CC_DEPARTAMENTO.Tiene_docentes;

Aquí, C3A devuelve bag<string> (en el resultado aparecen valores de rango duplicados), mientras que C3B devuelve set<string> (los duplicados se eliminan gracias a la palabra clave distinct). Tanto C3A como C3B ilustran cómo podemos definir una variable iteradora en la cláusula from para recorrer una colección restringida especificada en la consulta. La variable *F* de C3A y C3B recorre los elementos de la colección CC_DEPARTAMENTO.Tiene_docentes, que es de tipo set<DOCENTE>, y sólo incluye los profesores que pertenecen al departamento de informática.

En general, una consulta OQL puede devolver un resultado con una estructura compleja especificada en la propia consulta utilizando struct. Consideremos los siguientes ejemplos:

```
C4:  CC_DEPARTAMENTO.Director.Tutoriza;

C4A: select struct ( nombre:struct( apellido1: S.nombre.Apellido1,
                                    nombre_pila: S.nombre.NombreP),
                     licenciatura:( select struct ( lic: D.Título,
                                                    an: D.Año,
                                                    facultad: D.Facultad)
                                    from D in S.Licenciatura)
       from S in CC_DEPARTAMENTO.Director.Tutoriza;
```

Aquí, C4 es directa, devolviendo un objeto de tipo set<ESTUDIANTE_DIPLOMADO> como resultado; es la colección de estudiantes graduados tutorizados por el director del departamento de informática. Supongamos ahora que necesitamos una consulta para recuperar los nombres y los apellidos de esos estudiantes graduados y los títulos que tiene cada uno. Lo podemos escribir como en C4A, donde la variable *S* recorre la colección de estudiantes graduados tutorizados por el director, y la variable *D* recorre los títulos de cada estudiante *S*. El tipo del resultado de C4A es una colección de estructuras (primer nivel) donde cada una de ellas tiene dos componentes: nombre y licenciatura.[24]

El componente nombre es, a su vez, una estructura compuesta por apellido1 y nombre_pila, que son cadenas sencillas. El componente licenciatura está definido por una consulta incrustada y es, a su vez, una colección de otras estructuras (segundo nivel), cada una de tres componentes: lic, an y facultad.

OQL es *ortogonal* respecto a la especificación de expresiones de ruta; es decir, podemos utilizar atributos, relaciones y operaciones (métodos) en estas expresiones, siempre que el sistema de tipos de OQL no se vea comprometido. Por ejemplo, podemos escribir las siguientes consultas para recuperar la nota media de todos los estudiantes mayores licenciados en informática, con el resultado ordenado por la nota media y, después, por primer apellido y nombre de pila:

```
C5A: select struct (apellido1: S.nombre.Apellido1, nombre_pila:
                    S.nombre.NombreP, media: S.media )
     from           S in CC_DEPARTAMENTO.Tiene_licenciaturas
     where          S.Clase = 'senior'
     order by       media desc, apellido1 asc, nombre_pila asc;

C5B: select struct (apellido1: S.nombre.Apellido1, nombre_pila: S.nombre.NombreP,
                    media: S.media )
     from           S in ESTUDIANTES
     where          S.Licenciaturas_en.NombreDpto = 'Ciencias de la computación' and
                    S.Clase = 'senior'
     order by       media desc, apellido1 asc, nombre_pila asc;
```

C5A utiliza el punto de entrada con nombre CC_DEPARTAMENTO para localizar directamente la referencia al departamento de informática y localizar después los estudiantes a través de la relación Tiene_licenciaturas, mientras que C5B busca la extensión ESTUDIANTES para localizar todos los estudiantes licenciados en ese departamento. Los nombres de atributo, relación y operación (método) se utilizan indistintamente (de una manera ortogonal) en las expresiones de ruta: media es una operación; Licenciaturas_en y Tiene_licenciaturas son relaciones; y Clase, Nombre, NombreDpto, Apellido1 y NombreP son atributos. La implementación de la

[24] Como mencionamos anteriormente, struct es equivalente al constructor de tupla explicado en el Capítulo 20.

operación media calcula la nota media y devuelve su valor como de tipo flotante para cada ESTUDIANTE seleccionado.

La cláusula order by es parecida a la equivalente en SQL, y especifica el orden en el que se visualizará el resultado de la consulta. Por tanto, la colección devuelta por una consulta con una cláusula order by es de tipo *lista*.

### 21.3.3 Otras características de OQL

**Especificar vistas como consultas con nombre.** El mecanismo de vista de OQL utiliza el concepto de **consulta con nombre**. La palabra clave **define** se utiliza para especificar un identificador de la consulta con nombre, que debe ser un nombre único entre todos los objetos con nombre, nombres de clases, nombres de métodos o nombres de funciones del esquema. Si el identificador tiene el mismo nombre que una consulta con nombre existente, la definición nueva sustituye a la definición anterior. Una vez definida, una definición de consulta es persistente hasta que se redefine o elimina. Una vista también puede tener parámetros (argumentos) en su definición.

Por ejemplo, la siguiente vista V1 define la consulta con nombre Tiene_diplomatura para recuperar el conjunto de objetos estudiante que tienen una diplomatura en un departamento dado:

```
V1:  define  Tiene_diplomatura(Nombre_dpto) as
     select  S
     from    S in ESTUDIANTES
     where   S.Diplomaturas_en.NombreDpto = Nombre_dpto;
```

Como el esquema ODL de la Figura 21.6 sólo proporcionaba un atributo Diplomaturas_en unidireccional para un ESTUDIANTE, podemos utilizar la vista anterior para representar su inverso sin tener que definir explícitamente una relación. Este tipo de vista se puede utilizar para representar relaciones inversas que esperamos no utilizar con mucha frecuencia. Ahora podemos utilizar la vista anterior para escribir consultas como la siguiente:

```
Tiene_diplomatura('Ciencias de la computación');
```

que devolvería una bolsa de estudiantes matriculados en el departamento de informática. En la Figura 21.6, definimos Tiene_licenciaturas como una relación explícita, probablemente porque esperamos que se utilice con más frecuencia.

**Extraer elementos sencillos de colecciones sencillas.** Una consulta OQL devolverá, por regla general, una colección como resultado, como una bolsa, un conjunto (si especificamos distinct) o una lista (si utilizamos la cláusula order by). Si el usuario requiere que una consulta devuelva un elemento sencillo, hay un operador **element** en OQL que garantiza la devolución de un elemento *E* sencillo de una colección *C* sencilla que sólo contiene un elemento. Si *C* contiene más de un elemento o si *C* está vacía, entonces el operador de elemento *alcanza una excepción*. Por ejemplo, C6 devuelve el objeto sencillo referido al departamento de informática:

```
C6:  element ( select  D
               from    D in DEPARTAMENTOS
               where   D.NombreDpto = 'Ciencias de la computación');
```

Como el nombre de un departamento es único entre todos los departamentos, el resultado debe ser un departamento. El tipo del resultado es *D*:DEPARTAMENTO.

**Operadores de colección (funciones de agregación y cuantificadores).** Como muchas expresiones de consulta especifican colecciones como su resultado, se han definido varios operadores que se pueden aplicar a dichas colecciones. Son los operadores de agregación, así como la asociación y la cuantificación (universal y existencial) sobre una colección.

Los operadores de agregación (**min**, **max**, **count**, **sum** y **avg**) operan sobre una colección.[25] El operador count devuelve un tipo entero. El resto de operadores de agregación (min, max, sum, avg) devuelven el mismo tipo que la colección de operandos. A continuación tiene dos ejemplos. La consulta C7 devuelve el número de estudiantes diplomados en informática y C8 devuelve la nota media de todos los mayores licenciados en informática.

```
C7:  count (S in Tiene_diplomatura('Ciencias de la computación'));

C8:  avg ( select   S.Media
           from     S in ESTUDIANTES
           where    S.Licenciaturas_en.NombreDpto = 'Ciencias de la computación' and
                    S.Clase = 'Senior');
```

Las operaciones agregadas se pueden aplicar a cualquier colección del tipo apropiado y se pueden utilizar en cualquier parte de una consulta. Por ejemplo, la consulta para recuperar el nombre de todos los departamentos que tienen más de 100 licenciados puede escribirse de este modo:

```
C9:  select   D.NombreDpto
     from     D in DEPARTAMENTOS
     where    count (D.Tiene_licenciaturas) > 100;
```

Las expresiones *asociación* y *cuantificación* devuelven un tipo booleano (es decir, verdadero o falso). Si tenemos que *V* es una variable, *C* una expresión de colección, *B* una expresión de tipo booleano (es decir, una condición booleana) y *E* un elemento del tipo de los elemento de la colección *C*, entonces:

(*E* **in** *C*) devuelve verdadero si el elemento *E* es un miembro de la colección *C*.

(**for all** *V* **in** *C* : *B*) devuelve verdadero si *todos* los elementos de la colección *C* satisfacen *B*.

(**exists** *V* **in** *C* : *B*) devuelve verdadero si en *C* hay al menos un elemento que satisface *B*.

Para ilustrar la condición de membresía, supongamos que queremos recuperar los nombres de todos los estudiantes que han completado el curso denominado 'Sistemas de bases de datos I'. Lo podemos escribir como en C10, donde la consulta anidada devuelve la colección de nombres de cursos que cada ESTUDIANTE *S* ha completado, y la condición de membresía devuelve verdadero si 'Sistemas de bases de datos I' está en la colección para un ESTUDIANTE *S* concreto:

```
C10: select   S.nombre.Apellido2, S.nombre.NombreP
     from     S in ESTUDIANTES
     where    'Sistemas de bases de datos I' in
              ( select C.NombreCurso
                from   C in S.Niveles_completados.Nivel.De_curso);
```

C10 también ilustra una forma sencilla de especificar la cláusula select de las consultas que devuelven una colección de estructuras; el tipo devuelto por C10 es bag<struct(string, string)>.

También podemos escribir consultas que devuelvan resultados verdadero/falso. A modo de ejemplo, supongamos que existe un objeto con nombre denominado JUAN de tipo ESTUDIANTE. La consulta C11 responde a la siguiente cuestión: *¿Juan es diplomado en informática?* De forma parecida, C12 responde a la pregunta: *¿Todos los estudiantes graduados en informática están tutorizados por los profesores de informática?* Tanto C11 como C12 devuelven verdadero o falso, valores que se pueden interpretar como "sí" o como "no" en respuesta a las preguntas.

```
C11: JUAN in Tiene_diplomatura('Ciencias de la computación');
```

[25] Son equivalentes a las funciones agregadas de SQL.

```
C12: for all G in
     ( select   S
       from     S in ESTUDIANTES_GRADUADOS
       where    S.Licenciaturas_en.NombreDpto = 'Ciencias de la computación' )
     : G.Tutor in CC_DEPARTAMENTO.Tiene_docentes;
```

La consulta C12 también ilustra cómo la herencia de atributo, relación y operación se aplica a las consultas. Aunque *S* es un iterador que recorre la extensión ESTUDIANTES_GRADUADOS, podemos escribir *S*.Licenciaturas_en porque la relación Licenciaturas_en es heredada por ESTUDIANTE_DIPLOMADO a partir de ESTUDIANTE a través de las extensiones (véase la Figura 21.6). Por último, para ilustrar el cuantificador exists, la consulta C13 responde a la siguiente pregunta: *¿Algún licenciado en informática tiene una media de 4.0?* Una vez más, la operación media es heredada por ESTUDIANTE_DIPLOMADO a partir de ESTUDIANTE a través de las extensiones.

```
C13: exists G in
     ( select S
       from   S in ESTUDIANTES_GRADUADOS
       where  S.Licenciaturas_en.NombreDpto = 'Ciencias de la computación' )
     : G.Media = 4;
```

**Ordenar (indexar) expresiones de colección.** Como explicamos en la Sección 21.1.2, las colecciones que son listas y arrays tienen operaciones adicionales, como la recuperación de los elementos *i*-ésimo, primero y último. Además, existen operaciones para extraer una subcolección y concatenar dos listas. Por tanto, las expresiones de consulta que involucran a listas o arrays pueden invocar estas operaciones. Ilustraremos unas cuantas de estas operaciones utilizando consultas de ejemplo. C14 recupera el primer apellido del profesor que tiene el sueldo más elevado:

```
C14: first ( select     struct(nombrep: F.nombre.Apellido1, sueldo: F.Sueldo)
             from       F in DOCENTE
             order by   F.Sueldo desc);
```

C14 ilustra el uso del operador **first** sobre una colección lista que contiene los sueldos de los profesores ordenados descendentemente por lo que ganan. Por tanto, el primer elemento de esta lista ordenada será el profesor con el sueldo más alto. Esta consulta asume que sólo un profesor gana el sueldo máximo. La siguiente consulta, C15, recupera los tres mejores licenciados en informática en base a su nota media:

```
C15: ( select struct   (apellido1: S.nombre.Apellido1, nombre_pila: S.nombre.NombreP,
                                media: S.Media )
       from            S in CC_DEPARTAMENTO.Tiene_licenciaturas
       order by        media desc) [0:2];
```

La consulta select-from-order-by devuelve una lista de estudiantes de informática ordenada descendentemente por su nota media. El primer elemento de una colección ordenada tiene la posición de índice 0, por lo que la expresión [0:2] devuelve una lista que contiene los tres primeros elementos del resultado select ... from ... order by ....

**Operador de agrupamiento.** La cláusula **group by** de OQL, aunque parecida a la cláusula equivalente de SQL, proporciona una referencia explícita a la colección de objetos dentro de cada *grupo* o *partición*. Primero ofrecemos un ejemplo y después describimos la forma general de estas consultas.

La consulta C16 recupera el número de licenciados de cada departamento. En esta consulta, los estudiantes están agrupados en la misma partición (grupo) si tienen la misma licenciatura, es decir, el mismo valor para *S*.Licenciaturas_en.NombreDpto:

```
C16: ( select   struct( nombre_dpto, num_licenciados: count (partition) )
       from     S in ESTUDIANTES
       group by nombre_dpto: S.Licenciaturas_en.NombreDpto;
```

El resultado de la especificación de agrupamiento es de tipo set<struct(nombre_dpto: string, partition: bag<struct(S:ESTUDIANTE)>)>, que contiene una estructura para cada grupo (partition) que tiene dos componentes: el valor del atributo de agrupamiento (nombre_dpto) y la bolsa de objetos ESTUDIANTE del grupo (partition). La cláusula select devuelve el atributo de agrupamiento (nombre del departamento) y un recuento del número de elementos de cada partición (es decir, el número de estudiantes de cada departamento), donde **partition** es la palabra clave utilizada para referirnos a cada partición. El tipo de resultado de la cláusula de selección es set<struct(nombre_dpto: string, num_licenciados: integer)>. En general, la sintaxis de la cláusula group by es:

**group by** $F_1$: $E_1$, $F_2$: $E_2$, ... , $F_k$: $E_k$

donde $F_1$: $E_1$, $F_2$: $E_2$, ... , $F_k$: $E_k$ es una lista de atributos de partición (agrupamiento) y cada especificación de atributo de partición $F_i$: $E_i$ define un nombre de atributo (campo) $F_i$ y una expresión $E_i$. El resultado de aplicar el agrupamiento (especificado en la cláusula group by) es un conjunto de estructuras:

set<struct($F_1$: $T_1$, $F_2$: $T_2$, ... , $F_k$: $T_k$, partition: bag<$B$>)>

donde $T_i$ es el tipo devuelto por la expresión $E_i$, partition es un nombre de campo distinguido (una palabra clave), y $B$ es una estructura cuyos campos son las variables iteradoras ($S$ en C16) declaradas en la cláusula from teniendo el tipo apropiado.

Al igual que en SQL, una cláusula **having** se puede utilizar para filtrar los conjuntos particionados (es decir, selecciona sólo algunos de los grupos basándose en las condiciones de grupo). En C17 modificamos la consulta anterior para ilustrar la cláusula having (y también muestra la sintaxis más simplificada de la cláusula select). C17 recupera para cada departamento que tiene más de 100 licenciados, la nota media de esos licenciados. La cláusula having de C17 sólo selecciona aquellas particiones (grupos) que tienen más de 100 elementos (es decir, los departamentos con más de 100 estudiantes).

```
C17: select   nombre_dpto, nota_media: avg (select P.S.media from P in partition)
     from     S in ESTUDIANTES
     group by nombre_dpto: S.Licenciaturas_en.NombreDpto
     having   count (partition) > 100;
```

La cláusula select de C17 devuelve la nota media de los estudiantes de la partición. La expresión:

**select** *P*.*S*.Media **from** *P* in **partition**

devuelve una bolsa de notas medias de estudiantes para esta partición. La cláusula from declara una variable iteradora *P* sobre la colección de la partición, que es de tipo bag<struct(*S*: ESTUDIANTE)>. Después, la expresión de ruta *P*.*S*.media se utiliza para acceder a la nota media de cada estudiante de la partición.

## 21.4 Visión general de la vinculación del lenguaje C++

La vinculación del lenguaje C++ especifica cómo las construcciones ODL se mapean a las construcciones de C++. Esto se hace a través de la biblioteca de clases de C++ que proporciona clases y operaciones que implementan las construcciones ODL. Necesitamos un lenguaje de manipulación de objetos (OML) para especificar cómo se recuperan y manipulan los objetos de base de datos dentro de un programa C++, y esto está basado en la sintaxis y la semántica del lenguaje de programación C++. Además de las vinculaciones ODL/OML,

se define un conjunto de construcciones denominadas **pragmas físicos** para que el programador pueda tener algo de control sobre los asuntos de almacenamiento físico, como el agrupamiento de objetos, la utilización de índices y la administración de la memoria.

La biblioteca de clases añadida a C++ para el estándar ODMG utiliza el prefijo d_ para las declaraciones de clase que se ocupan de los conceptos de bases de datos.[26] El objetivo es que el programador piense que sólo está utilizando un lenguaje, no dos lenguajes diferentes. Para que el programador se refiera a los objetos de la base de datos de un programa, definimos una clase D_Ref<*T*> para cada clase *T* de base de datos del esquema. Por tanto, las variables de programa de tipo D_Ref<*T*> pueden referirse a objetos persistentes y transitorios de clase *T*.

A fin de utilizar los distintos tipos integrados del modelo de objeto ODMG como tipos colección, en la biblioteca se especifican diferentes clases de plantilla. Por ejemplo, una clase abstracta D_Object<*T*> especifica las operaciones que todos los objetos heredarán. De forma parecida, una clase abstracta D_Collection<*T*> especifica las operaciones de las colecciones. Estas clases no son instanciables, sino que sólo especifican las operaciones que todos los objetos y objetos de colección pueden heredar, respectivamente. Se especifica una clase plantilla para cada tipo de colección; incluye D_Set<*T*>, D_List<*T*>, D_Bag<*T*>, D_Varray<*T*> y D_Dictionary<*T*>, y equivale a los tipos colección del modelo de objeto (consulte la Sección 21.1). Por tanto, el programador puede crear clases de tipos como D_Set<D_Ref<ESTUDIANTE>> cuyas instancias serían conjuntos de referencias a objetos ESTUDIANTE, o D_Set<string> cuyas instancias serían conjuntos de cadenas. Además, una clase d_Iterator corresponde a la clase iteradora del modelo de objeto.

El ODL C++ permite a un usuario especificar las clases de un esquema de base de datos utilizando las construcciones de C++, así como las construcciones proporcionadas por la biblioteca de base de datos de objetos. Al especificar los tipos de datos de los atributos,[27] se proporcionan los tipos básicos, como d_Short (entero corto), d_Ushort (entero corto sin signo), d_Long (entero largo) y d_Float (número en coma flotante). Además de los tipos de datos básicos, se suministran varios tipos literales estructurados equivalentes a los tipos literales estructurados del modelo de objeto ODMG: d_String, d_Interval, d_Date, d_Time y d_Timestamp (véase la Figura 21.1[b]).

Para especificar las relaciones, se utiliza la palabra clave rel_ dentro del prefijo de los nombres de tipo. Por ejemplo, al escribir:

```
d_Rel_Ref<DEPARTAMENTO, _tiene_licenciaturas> Licenciaturas_en;
```

en la clase ESTUDIANTE, y:

```
d_Rel_Set<ESTUDIANTE, _licenciaturas_en> Tiene_licenciaturas;
```

en la clase DEPARTAMENTO, estamos declarando que Licenciaturas_en y Tiene_licenciaturas son propiedades de relación que son inversas entre sí y, por tanto, representan una relación binaria 1:N entre DEPARTAMENTO y ESTUDIANTE.

Para OML, la vinculación sobrecarga la *nueva* operación, de modo que podemos utilizarla para crear objetos persistentes o transitorios. Para crear objetos persistentes, debemos proporcionar el nombre de la base de datos y el nombre persistente del objeto. Por ejemplo, al escribir:

```
D_Ref<ESTUDIANTE> S = new(DB1, 'José_Pérez') ESTUDIANTE;
```

el programador crea un objeto persistente con nombre de tipo ESTUDIANTE en la base de datos DB1 con el nombre persistente José_Pérez. Otra operación, delete_object(), se utiliza para borrar objetos. La modificación de un objeto se lleva a cabo mediante otras operaciones (métodos) que el programador define en cada clase.

---

[26] Probablemente, d_ se refiere a clases de *bases de datos*.

[27] Es decir, *variables miembro* en la terminología de programación orientada a objetos.

La vinculación C++ también permite la creación de extensiones utilizando d_Extent. Por ejemplo, con:

```
D_Extent<PERSONA> TODAS_PERSONAS(DB1);
```

el programador crearía un objeto de colección con nombre TODAS_PERSONAS (cuyo tipo sería D_Set<PERSONA>) en la base de datos DB1 que albergaría objetos persistentes de tipo PERSONA. Sin embargo, las restricciones clave no están soportadas en la vinculación C++, y cualquier comprobación de clave debe programarse en los métodos de clase.[28] Además, la vinculación C++ no soporta la persistencia a través de la noción de alcance; el objeto debe ser declarado estáticamente como persistente en el momento de crearlo.

# 21.5 Diseño conceptual de bases de datos de objetos

La Sección 21.5.1 explica las diferencias entre el diseño de bases de datos de objeto (ODB) y el diseño de bases de datos relacionales (RDB). La Sección 21.5.2 describe un algoritmo de mapeado que puede utilizarse para crear un esquema ODB, compuesto por definiciones de clase ODL ODMG, a partir de un esquema EER conceptual.

## 21.5.1 Diferencias entre diseño conceptual de ODB y RDB

Una de las principales diferencias entre el diseño ODB y el diseño RDB es la manipulación de las relaciones. En ODB, las relaciones normalmente se manipulan teniendo propiedades de relación o atributos de referencia que incluyen OID(s) de los objetos relacionados. Se pueden considerar como *referencias OID* a los objetos relacionados. Están permitidas tanto las referencias sencillas como las colecciones de referencias. Las referencias para una relación binaria sólo se pueden declarar en una dirección, o en ambas direcciones, en función de los tipos de acceso esperados. Si la declaración es en ambas direcciones, deben especificarse como inversas entre sí, cumpliéndose así el equivalente ODB de la restricción de integridad referencial relacional.

En RDB, las relaciones entre tuplas (registros) se especifican mediante atributos con valores coincidentes, que pueden considerarse como *referencias de valor* y se especifican a través de *foreign keys* (claves externas), que son valores de los atributos clave principales repetidos en tuplas de la relación referida. Tienen que ser monovalor en cada registro porque los atributos multivalor no están permitidos en el modelo relacional básico. Por tanto, las relaciones M:N no deben representarse directamente, sino como una relación separada (tabla), como explicamos en la Sección 7.1.

El mapeo de relaciones binarias que contienen atributos no es directo en los ODBs, ya que el diseñador debe elegir la dirección en la que deben incluirse los atributos. Si los atributos se incluyen en las dos direcciones, existirá redundancia en almacenamiento, lo que puede llevar a unos datos inconsistentes. En consecuencia, a veces es preferible utilizar el método relacional consistente en crear una tabla independiente mediante la creación de una clase separada para representar la relación. Este método también se puede utilizar para las relaciones $n$-arias, con un grado $n > 2$.

Otra área importante en la que se diferencia el diseño ODB y el diseño RDB es la manipulación de la herencia. En ODB, estas estructuras se integran en el modelo, por lo que el mapeado se consigue utilizando las construcciones de herencia como *derivadas* (:) y *EXTENDS*. En el diseño relacional, como explicamos en la Sección 7.2, hay varias opciones entre las que elegir, ya que no existen construcciones integradas para la herencia en el modelo relacional básico. Es importante reseñar, no obstante, que los sistemas de objetos relacionales y relacionales extendidos están añadiendo características para modelar estas construcciones

[28] Sólo hemos ofrecido una breve panorámica de la vinculación de C++. Si desea información más detallada, consulte el Capítulo 5 de Cattell y otros (1997).

directamente, así como para incluir especificaciones de operación en los tipos de datos abstractos (consulte el Capítulo 22).

La tercera diferencia importante es que en el diseño ODB es necesario especificar las operaciones en una fase temprana del diseño, porque forman parte de las especificaciones de clase. Aunque es importante especificar durante la fase de diseño las operaciones para todos los tipos de bases de datos, en el diseño RDB es una tarea que se puede demorar hasta la fase de implementación.

Hay una diferencia filosófica entre el modelo relacional y el modelo de objetos de datos en lo que se refiere a la especificación del comportamiento. En el modelo relacional no es obligatorio predefinir un conjunto de comportamientos u operaciones válidos, mientras que es un requisito tácito en el modelo de objetos. Una de las ventajas demandadas del modelo relacional es el soporte de consultas y transacciones específicas considerando que van contra el principio de encapsulamiento.

En la práctica, es cada vez más común contar con equipos de diseño de bases de datos que aplican metodologías basadas en objetos en etapas muy tempranas del diseño conceptual, de modo que durante esta fase se tienen en consideración la estructura y el uso o las operaciones de datos y se desarrolla una especificación completa. Estas especificaciones se mapean después en esquemas relaciones, restricciones y comportamientos como *triggers* o procedimientos almacenados (consulte las Secciones 8.7 y 9.4).

### 21.5.2 Mapeado de un esquema EER a un esquema ODB

Es relativamente rápido diseñar las declaraciones de tipos de las clases de objetos para un ODBMS a partir de un esquema EER que no contiene categorías ni relaciones $n$-arias con $n > 2$. Sin embargo, en el diagrama EER no se especifican las operaciones de clases y deben añadirse a las declaraciones de clase una vez completado el mapeado estructural. A continuación mostramos el mapeado de EER a ODL:

**Paso 1.** Cree una *clase* ODL para cada tipo de entidad o subclase EER. El tipo de la clase ODL debe incluir todos los atributos de la clase EER.[29] Los *atributos multivalor* se declaran utilizando los constructores de conjunto, bolsa o lista.[30] Si los valores del atributo multivalor para un objeto deben estar ordenados, debe elegir el constructor de lista; si están permitidos los duplicados, debe elegir el constructor de bolsa; en caso contrario, debe elegir el constructor de conjunto. Los *atributos compuestos* se mapean en un constructor de tupla (utilizando una declaración struct en ODL).

Declare una extensión para cada clase y especifique los atributos clave como claves de la extensión. (Esto sólo es posible si el ODBMS cuenta con un servicio de extensión y declaraciones de restricción de clave.)

**Paso 2.** Añada propiedades de relación o atributos de referencia para cada *relación binaria* en las clases ODL que participen en la relación. Esto lo puede crear en una o en las dos direcciones. Si una relación binaria está representada por referencias en las dos direcciones, declare las referencias para que sean propiedades de la relación inversas entre sí, si existe tal posibilidad.[31] Si una relación binaria está representada por una referencia en una sola dirección, declare la referencia para que sea un atributo en la clase de referencia cuyo tipo es el nombre de la clase referenciada.

---

[29] Esto utiliza implícitamente un constructor de tupla en el nivel superior de la declaración del tipo, pero en general, el constructor de tupla no se muestra explícitamente en las declaraciones de clase ODL.

[30] Se necesita un análisis posterior del dominio de la aplicación para decidir el constructor que ha de utilizarse, porque esta información no está disponible a partir del esquema EER.

[31] El estándar ODL proporciona la definición explícita de relaciones inversas. Algunos productos ODBMS no ofrecen este soporte; en esos casos, los programadores deben mantener las relaciones explícitamente, codificando adecuadamente los métodos que actualizan los objetos.

En función de la razón de cardinalidad de la relación binaria, las propiedades de relación o los atributos de referencia pueden ser tipos monovalor o colecciones. Serán monovalor para las relaciones binarias en las direcciones 1:1 o N:1; serán tipos colección (conjunto o lista[32]) para las relaciones en la dirección 1:N o M:N. En el paso 7 explicamos una forma alternativa de mapear las relaciones M:N binarias.

Si existen atributos de relación, puede utilizar un constructor de tupla (struct) para crear una estructura de la forma <referencia, atributos relación>, que puede incluir en sustitución del atributo de referencia. Sin embargo, esto no permite el uso de la restricción inversa. Además, si representa esta opción en las dos direcciones, los valores de atributo se representarán dos veces, lo que dará lugar a redundancia.

**Paso 3.** Incluya las operaciones adecuadas para cada clase. No están disponibles a partir del esquema EER y debe añadirlas al diseño de la base de datos haciendo referencia a los requisitos originales. Un método constructor debe incluir código de programa que compruebe las restricciones que deban mantenerse cuando se crea un objeto. Un método destructor debe comprobar las restricciones que pueden violarse al eliminar un objeto. Otros métodos deben incluir cualquier restricción posterior que sea relevante.

**Paso 4.** Una clase ODL que corresponda a una subclase del esquema EER hereda (a través de EXTENDS) el tipo y los métodos de su superclase en el esquema ODL. Debe especificar sus atributos (no heredados) *específicos*, referencias de relación y operaciones, como explicamos en los pasos 1, 2 y 3.

**Paso 5.** Los tipos de entidad débiles se pueden mapear de la misma forma que los tipos de entidad normales. Es posible un mapeado alternativo para los tipos de entidad débiles que no participan en ninguna relación, excepto su relación de identificación; puede mapearlos como si fueran *atributos multivalor compuestos* del tipo de entidad propietario, utilizando los constructores set<struct<... >> o list<struct<... >>. Los atributos de la entidad débil se incluyen en la construcción struct<... >, que corresponde a un constructor de tupla. Los atributos se mapean como explicamos en los pasos 1 y 2.

**Paso 6.** Las categorías (tipos de unión) de un esquema EER son difíciles de mapear a ODL. Es posible crear un mapeado parecido al mapeado EER-a-relacional (consulte la Sección 7.2) declarando una clase que represente la categoría y definiendo relaciones 1:1 entre la categoría y cada una de sus superclases. Otra opción es utilizar un *tipo unión*, si lo hay.

**Paso 7.** Una relación *n*-aria con un grado $n > 2$ se puede mapear en dos clases separadas, con las referencias apropiadas a cada clase participante. Esas referencias están basadas en el mapeado de una relación 1:N desde cada clase que representa un tipo de entidad participante a la clase que representa la relación *n*-aria. Una relación binaria M:N, especialmente si contiene atributos de relación, también puede utilizar esta opción de mapeado, si lo desea.

El mapeado se ha aplicado a un subconjunto del esquema de la base de datos UNIVERSIDAD de la Figura 4.10 en el contexto del estándar de bases de datos de objetos ODMG. En la Figura 21.6 se muestra el esquema de objeto mapeado utilizando la notación ODL.

# 21.6 Resumen

En este capítulo hemos explicado el estándar ODMG 2.0 para las bases de datos de objetos. Empezamos describiendo las distintas construcciones del modelo de objetos. Los diferentes tipos integrados, como Object, Collection, Iterator, set, list, etcétera, se describieron por sus interfaces, que especifican las operaciones integradas de cada tipo. Estos tipos integrados son los cimientos en los que se basan el lenguaje de definición de

[32] La decisión de si utilizar set o list no está disponible a partir del esquema EER y debe determinarse a partir de los requisitos.

objetos (ODL) y el lenguaje de consulta de objetos (OQL). También describimos la diferencia entre objetos, que tienen un id de objeto, y literales, que son valores sin OID. Los usuarios pueden declarar clases para su aplicación que hereden operaciones de las interfaces integradas adecuadas. En una clase definida por el usuario es posible especificar dos tipos de propiedades, atributos y relaciones, además de las operaciones que pueden aplicarse a los objetos de la clase. El ODL permite a los usuarios especificar interfaces y clases, y permite dos tipos de herencia diferentes: herencia de interfaz a través de ":" y herencia de clase mediante EXTENDS. Una clase puede tener una extensión y claves.

A esto le siguió una descripción de ODL y utilizamos un ejemplo de esquema de base de datos para la base de datos UNIVERSIDAD para ilustrar las construcciones de ODL. Ofrecimos una visión general del lenguaje de consulta de objetos (OQL). El OQL obedece al concepto de ortogonalidad en la construcción de consultas, lo que significa que una operación se puede aplicar al resultado de otra operación, siempre y cuando el tipo del resultado sea del tipo de entrada correcto para la operación. La sintaxis de OQL obedece muchas de las construcciones de SQL, pero incluye conceptos adicionales, como expresiones de ruta, herencia, métodos, relaciones y colecciones. También ofrecimos ejemplos de cómo se utiliza OQL sobre la base de datos UNIVERSIDAD.

A continuación ofrecimos una panorámica de la vinculación del lenguaje C++, que extiende las declaraciones de clase de C++ con los constructores de tipo de ODL, pero permite una integración perfecta de C++ con el ODBMS.

Tras la descripción del modelo ODMG, describimos una técnica general para diseñar esquemas de bases de datos de objetos. Explicamos las diferencias entre las bases de datos de objetos y las bases de datos relacionales en tres áreas principales: referencias para representar las relaciones, inclusión de operaciones y herencia. Por último, mostramos cómo mapear un diseño de base de datos conceptual en el modelo EER a las construcciones de bases de datos de objetos. En 1997 Sun apoyó la API (interfaz de programación de aplicaciones) ODMG. O2 technologies fue la primera empresa en distribuir un DBMS compatible con ODMG. Muchos desarrolladores de OODBMS, como Object Design (now eXcelon), Gemstone systems, Poet Software y Versant Object Technology, han respaldado el estándar ODMG.

## Preguntas de repaso

**21.1.** ¿Cuáles son las diferencias y las similitudes entre los objetos y los literales en el modelo de objeto ODMG?

**21.2.** Enumere las operaciones básicas de las siguientes interfaces integradas del modelo de objeto ODMG: Object, Collection, Iterator, Set, List, Bag, Array y Dictionary.

**21.3.** Describa los literales estructurados integrados del modelo de objeto ODMG y las operaciones de cada uno.

**21.4.** ¿Cuáles son las diferencias y las similitudes entre las propiedades de atributo y de relación de una clase definida por el usuario (atómica)?

**21.5.** ¿Cuáles son las diferencias y las similitudes entre EXTENDS y la herencia ":" de interfaz?

**21.6.** Explique cómo se especifica la persistencia en el modelo de objeto ODMG en la vinculación C++.

**21.7.** ¿Por qué son importantes los conceptos de extensiones y claves en las aplicaciones de bases de datos?

**21.8.** Describa los siguientes conceptos OQL: *puntos de entrada a la base de datos*, *expresiones de ruta*, *variables iteradoras*, *consultas con nombre (vistas)*, *funciones agregadas*, *agrupamiento* y *cuantificadores*.

**21.9.** ¿Qué se entiende por ortogonalidad de tipo de OQL?

**21.10.** Explique los principios generales tras la vinculación C++ del estándar ODMG.

**21.11.** ¿Cuáles son las principales diferencias entre diseñar una base de datos relacional y una de objetos?

**21.12.** Describa los pasos del algoritmo para el diseño de una base de datos de objetos por mapeado EER-a-OO.

## Ejercicios

**21.13.** Diseñe un esquema OO para una aplicación de bases de datos en la que esté interesado. Cree un esquema EER para la aplicación, y después cree en ODL las clases correspondientes. Especifique varios métodos para cada clase, y después especifique consultas en OQL para su aplicación de bases de datos.

**21.14.** Considere la base de datos AEROPUERTO descrita en el Ejercicio 4.21. Especifique varias operaciones/métodos que considera aplicables a esta aplicación. Especifique las clases ODL y los métodos para la base de datos.

**21.15.** Mapee el esquema ER EMPRESA de la Figura 3.2 en clases ODL. Incluya los métodos apropiados para cada clase.

**21.16.** Especifique en OQL las consultas de los ejercicios de los Capítulos 7 y 8 que se aplican a la base de datos EMPRESA.

**21.17.** Con motores de búsqueda y otras fuentes, determine en qué medida los distintos productos OODBMS son compatibles con el estándar ODMG 3.0.

## Bibliografía seleccionada

Cattell y otros (1997) describe el estándar ODMG 2.0, que hemos explicado en este capítulo, y Cattell y otros (1993) describe las versiones anteriores del estándar. Bancilhon y Ferrari (1995) ofrecen una presentación tutorial de los aspectos más importantes del estándar ODMG. Cattell y otros (2000) describe los últimos avances en el estándar ODMG 3.0. Varios libros describen la arquitectura CORBA; por ejemplo, Baker (1996). En la bibliografía del Capítulo 20 encontrará otras referencias generales relacionadas con las bases de datos orientadas a objetos.

El sistema O2 se describe en Deux y otros (1991), y Bancilhon y otros (1992) incluye una lista de referencias a otras publicaciones que describen distintos aspectos de O2. El modelo O2 se formalizó en Velez y otros (1989). El sistema ObjectStore se describió en Lamb y otros (1991). Fishman y otros (1987) y Wilkinson y otros (1990) explican IRIS, un DBMS orientado a objetos desarrollado en los laboratorios de Hewlett-Packard. Maier y otros (1986) y Butterworth y otros (1991) describen el diseño de GEMSTONE. En Thompson et al. (1993) se describe un sistema OO que soporta la arquitectura abierta desarrollada por Texas Instruments. El sistema ODE desarrollado por AT&T Bell Labs se describe en Agrawal y Gehani (1989). El sistema ORION desarrollado en MCC se describe en Kim y otros (1990). Morsi y otros (1992) describe un testbed OO.

Cattell (1991) examina los conceptos relacionados con las bases de datos relacionales y de objetos, y explica varios prototipos de sistemas de bases de datos basados en objetos y relacionales extendidos. Alagic (1997) señala las discrepancias entre el modelo de datos ODMG y sus vinculaciones de lenguaje y propone algunas soluciones. Bertino y Guerrini (1998) propone una extensión del modelo ODMG que soporta objetos compuestos. Alagic (1999) presenta varios modelos de datos pertenecientes a la familia ODMG.

CAPÍTULO 22

# Sistemas de objetos relacionales y relacionales extendidos

En los capítulos anteriores explicamos principalmente tres modelos de datos: el modelo entidad-relación (ER) y su versión mejorada, el modelo EER, en los Capítulos 3 y 4; el modelo de datos relacional y sus lenguajes y sistemas en los Capítulos 5 a 9; y el modelo de datos orientado a objetos y los lenguajes y estándares de las bases de datos de objetos en los Capítulos 20 y 21. Ahora vamos a ver cómo se han desarrollado todos estos modelos de datos ampliamente en lo que se refiere a las siguientes características:

- Modelado de estructuras para desarrollar esquemas para las aplicaciones de bases de datos.
- Servicios de restricción para expresar ciertos tipos de relaciones y restricciones sobre los datos según determina la semántica de la aplicación.
- Operaciones y servicios del lenguaje para manipular la base de datos.

De estos modelos, el modelo ER y sus variantes se han empleado principalmente en las herramientas CASE que se utilizan para diseñar bases de datos y software, mientras que los otros modelos se han utilizado como base para los DBMSs comerciales. Este capítulo explica la clase emergente de DBMSs comerciales que se denominan *sistemas objeto-relacional* (de objetos relacionales) o *relacionales extendidos*, y algunas de las bases conceptuales para estos sistemas. Estos sistemas, que a menudo se conocen como DBMSs objeto-relacional (ORDBMSs), aparecen como una forma de mejorar las capacidades de los DBMSs relacionales (RDBMSs) con algunas de las características que aparecen en los DBMSs de objetos (ODBMSs).

Empezaremos en la Sección 22.1 con una visión general del estándar SQL:99, que proporciona capacidades extendidas y de objetos al estándar SQL para los RDBMSs. En la Sección 22.2 ofrecemos una perspectiva histórica de la evolución de la tecnología de bases de datos y las tendencias actuales para entender por qué aparecieron estos sistemas. En la Sección 22.3 ofrecemos una visión general del servidor de bases de datos Informix a modo de ejemplo de ORDBMS extendido comercial. En la Sección 22.4 explicamos las características objeto-relacional y extendidas de Oracle. La Sección 22.5 está dedicada a algunos de los problemas relacionados con la implementación de los sistemas relacionales extendidos y la Sección 22.6 presenta una panorámica del modelo relacional anidado, que proporciona algunas de las bases teóricas que hay tras el modelo relacional con objetos complejos. La Sección 22.7 es un resumen del capítulo.

Los lectores interesados en las características típicas de un ORDBMS pueden leer las Secciones 22.1 a 22.4; pueden omitir las demás secciones si únicamente están interesados en una introducción.

# 22.1 Visión general de SQL y sus características objeto-relacional

En el Capítulo 8 hicimos una introducción de SQL como lenguaje estándar para los RDBMSs. Como dijimos, Chamberlin y Boyce (1974) especificó SQL por primera vez y experimentó mejoras en 1989 y 1992. El lenguaje continuó su evolución hacia un nuevo estándar denominado SQL3, que añade características de orientación a objetos, entre otras. También se aprobó un subconjunto del estándar SQL3, ahora conocido como SQL:99. Esta sección destaca algunas de las características de SQL3 y SQL:99, centrándose principalmente en los conceptos de orientación a objetos.

## 22.1.1 El estándar SQL y sus componentes

Primero enumeraremos las partes del estándar SQL que explicaremos, y después describiremos algunas de las características de SQL relacionadas con las extensiones de objeto de SQL. El estándar SQL incluye ahora las siguientes partes:[1]

- SQL/Framework, SQL/Foundation, SQL/Bindings y SQL/Object.
- Nuevas partes del direccionamiento temporal, las transacciones y otros aspectos de SQL.
- SQL/CLI (Interfaz de nivel de llamadas).
- SQL/PSM (Módulos almacenados persistentes).

SQL/Foundation se encarga de los nuevos tipos de datos, los predicados nuevos, las operaciones relacionales, los cursores, las reglas y los *triggers* (disparadores), los tipos definidos por el usuario, las capacidades de transacción y las rutinas almacenadas. SQL/CLI (Interfaz de nivel de llamadas) (consulte la Sección 9.3) suministra reglas que permiten la ejecución de código de aplicación sin proporcionar el código fuente y evita la necesidad del preprocesamiento. Contiene aproximadamente 50 rutinas para tareas como la conexión con el servidor SQL, la asignación y la "desasignación" de recursos, la obtención de diagnósticos e información de implementación, y el control de la terminación de transacciones. SQL/PSM (Módulos almacenados persistentes) (consulte la Sección 9.4) especifica los servicios para particionar una aplicación entre un cliente y un servidor. El objetivo es mejorar el rendimiento minimizando el tráfico en la red. SQL/Bindings incluye Embedded SQL y Direct Invocation. Embedded SQL se ha mejorado para incluir declaraciones de excepción adicionales.

SQL/Temporal se encarga de los datos históricos, los datos de series temporales y otras extensiones temporales, y lo está proponiendo el comité TSQL2.[2] La especificación SQL/Transaction formaliza la interfaz XA para que puedan utilizarla los implementadores SQL.

## 22.1.2 Soporte objeto-relacional en SQL-99

La especificación SQL/Object extiende SQL-92 para incluir capacidades de orientación a objetos. Explicaremos algunas de estas características refiriéndonos a los conceptos de orientación a objetos correspondientes que explicamos en el Capítulo 20. A continuación tiene algunas de las características que se han incluido en SQL-99:

- Se han añadido algunos **constructores de tipos** para especificar objetos complejos. Entre ellos se encuentra el *tipo row* (fila), que corresponde al constructor de tupla (o estructura) del Capítulo 20.

[1] La explicación sobre el estándar estará en su mayor parte basada en Melton y Mattos (1996).

[2] La propuesta completa aparece en Snodgrass y Jensen (1996). En el Capítulo 24 explicamos el modelado temporal e introducimos TSQL2.

También se suministra el *tipo array* para especificar colecciones. Otros constructores de tipo colección, como los de conjunto, lista y bolsa, todavía no forman parte de las especificaciones SQL-99, aunque algunos sistemas los incluyen y es de esperar que, en las versiones futuras, formen parte del estándar.

- Asimismo, se incluye un mecanismo para especificar la **identidad de objeto** mediante el uso del *tipo referencia*.
- La **encapsulación de operaciones** se suministra a través del mecanismo de tipos definidos por el usuario, que puede incluir operaciones como parte de su declaración.
- También se ofrecen mecanismos de **herencia**.

Ahora pasaremos a explicar cada uno de estos detalles más en profundidad.

**Constructores de tipos.** Los constructores de tipos row y array se utilizan para especificar tipos complejos. También se conocen como **tipos definidos por el usuario**, o **UDTS**, ya que el usuario los define para una aplicación en particular. Un **tipo row** puede especificarse utilizando la siguiente sintaxis:

```
CREATE TYPE NOMBRE_TIPO_ARRAY AS [ ROW ] (<declaraciones componente>);
```

La palabra clave ROW es opcional. A continuación tiene un ejemplo de cómo especificar un tipo fila para las direcciones y los empleados:

```
CREATE TYPE TIPO_DIRECC AS (
        Calle       VARCHAR (45),
        Ciudad      VARCHAR (25),
        CPostal     CHAR (5)
);

CREATE TYPE TIPO_EMP AS (
        Nombre      VARCHAR (35),
        Direcc      TIPO_DIRECC,
        Edad        INTEGER
);
```

Podemos utilizar un tipo definido anteriormente como tipo para un atributo, como ilustramos antes para el atributo Direcc. Un **tipo array** puede especificarse para un atributo cuyo valor será una colección. Por ejemplo, supongamos que una empresa tiene hasta diez emplazamientos. Podríamos definir un tipo *row* para la empresa, de este modo:

```
CREATE TYPE TIPO_EMPRESA AS (
Nombre_empr       VARCHAR (20),
Emplazamiento     VARCHAR (20) ARRAY [10] );
```

Los tipos array de longitud fija tienen sus elementos referenciados mediante la notación común de los corchetes. Por ejemplo, Emplazamiento[1] se refiere a la primera ubicación del atributo Emplazamiento. En el caso de los tipos *row*, se utiliza la notación de punto para referirse a los componentes. Por ejemplo, Direcc.Ciudad se refiere al componente Ciudad de un atributo Direcc. Realmente, los elementos de un array no pueden ser arrays a su vez, lo que limita la complejidad de las estructuras de objeto que podemos crear.

**Identificadores de objeto utilizando referencias.** Un tipo definido por el usuario se puede utilizar como tipo para un atributo, como lo ilustrábamos con el atributo Direcc de TIPO_EMP, o para especificar los tipos *row* de las tablas. Por ejemplo, podemos crear dos tablas basadas en las declaraciones de tipo *row* ofrecidas anteriormente:

```
CREATE TABLE EMPLEADO OF TIPO_EMP REF IS Id_emp
    SYSTEM GENERATED;

CREATE TABLE EMPRESA OF TIPO_EMPRESA (
    REF IS Id_empresa SYSTEM GENERATED,
    PRIMARY KEY (Nombre_empr) );
```

Los ejemplos anteriores también ilustran cómo podemos especificar que deben crearse los identificadores de objeto generados por el sistema para las filas individuales de una tabla. Con la sintaxis:

```
REF IS <atributo_oid> <método_generación_valor> ;
```

declaramos que el atributo denominado <atributo_oid> se utilizará para identificar tuplas individuales de la tabla. Las opciones para <método_generación_valor> son SYSTEM GENERATED o DERIVED. En el primer caso, el sistema genera automáticamente un identificador único para cada tupla. En el segundo caso, se aplica el método tradicional de utilizar el valor de la clave principal proporcionada por el usuario.

Un atributo componente de una tupla puede ser una **referencia** (especificado con la palabra clave REF) a una tupla de otra (o posiblemente la misma) tabla. Por ejemplo, podemos definir el siguiente tipo fila adicional y la tabla correspondiente para relacionar un empleado con una empresa:

```
CREATE TYPE TIPO_EMPLEO AS (
    Empleado REF (TIPO_EMP) SCOPE (EMPLEADO),
    Empresa REF (TIPO_EMPRESA) SCOPE (EMPRESA) );

CREATE TABLE EMPLEO OF TIPO_EMPLEO;
```

La palabra clave SCOPE especifica el nombre de la tabla cuyas tuplas pueden ser referenciadas por el atributo de referencia. Esto es parecido a la clave externa (*foreign key*), excepto que se utiliza el valor generado por el sistema en lugar del valor de la clave principal.

SQL utiliza la **notación de punto** para crear **expresiones de ruta** que se refieren a los atributos componente de las tuplas y los tipos fila. Sin embargo, para un atributo cuyo tipo es REF, se utiliza el símbolo de derreferencia –>. Por ejemplo, la siguiente consulta recupera los empleados que trabajan en la empresa denominada 'ABCXYZ', y lo hace consultando la tabla EMPLEO:

```
SELECT    E.Empleado–>Nombre
FROM      EMPLEO AS E
WHERE     E.Empresa–>Nombre_empr = 'ABCXYZ';
```

En SQL, se utiliza –> para **derreferenciar** y tiene el mismo significado que en el lenguaje de programación C. Así, si *r* es una referencia a una tupla y *a* es un atributo componente de esa tupla, entonces *r* –> *a* es el valor de atributo *a* en esa tupla.

Los identificadores de objeto también pueden declararse explícitamente en la definición de tipo, en lugar de hacerlo en la declaración de tabla. Por ejemplo, la definición de TIPO_EMP puede modificarse de este modo:

```
CREATE TYPE TIPO_EMP AS (
    Nombre      CHAR (35),
    Direcc      TIPO_DIRECC,
    Edad        INTEGER,
    Id_emp      REF (TIPO_EMP)
);
```

En el ejemplo anterior, los valores Id_emp pueden especificarse para que sean generados por el sistema utilizando el siguiente comando:

```
CREATE TABLE EMPLEADO OF TIPO_EMP
VALUES FOR Id_emp ARE SYSTEM GENERATED;
```

Si existen varias relaciones del mismo tipo fila, SQL proporciona la palabra clave SCOPE por la que un atributo de referencia puede apuntar a una tabla específica de ese tipo:

```
SCOPE FOR <atributo> IS <relación>
```

**Encapsulación de operaciones en SQL.** SQL suministra una construcción parecida a la definición de una clase gracias a la cual el usuario puede crear un **tipo definido por el usuario** (UDT) con nombre y con su propio comportamiento, especificando los métodos (u operaciones) además de los atributos. La forma general de una especificación UDT con métodos es la siguiente:

```
CREATE TYPE <nombre-tipo> (
        lista de atributos componente con tipos individuales
        declaración de funciones EQUAL y LESS THAN
declaración de otras funciones (métodos)
);
```

Por ejemplo, supongamos que queremos extraer el número de apartamento (si viene dado) de una cadena que forma el atributo componente Calle del tipo fila TIPO_DIRECC declarado anteriormente. Podemos especificar un método para TIPO_DIRECC de este modo:

```
CREATE TYPE TIPO_DIRECC AS (
        Calle   VARCHAR (45),
        Ciudad      VARCHAR (25),
        CPostal     CHAR (5) )

METHOD num_apto() RETURNS CHAR (8);
```

Todavía tenemos que escribir el código para implementar el método. Podemos referirnos a la implementación del método especificando el archivo que contiene el código del método:

```
METHOD
CREATE FUNCTION num_apto() RETURNS CHAR (8) FOR TIPO_DIRECC AS
EXTERNAL NAME '/x/y/numapto.class' LANGUAGE 'java';
```

En este ejemplo, la implementación se encuentra en lenguaje Java, y el código está almacenado en el fichero especificado en la ruta de acceso.

SQL proporciona ciertas funciones integradas para los tipos definidos por el usuario. Para un UDT denominado TIPO_T, la **función constructora** TIPO_T() devuelve un nuevo objeto de ese tipo. En el nuevo objeto UDT, cada atributo se inicializa con su valor predeterminado. Una **función observadora** *A* se crea implícitamente para cada atributo *A* para leer su valor. Por tanto, *A*(*X*) o *X.A* devuelve el valor de atributo *A* de TIPO_T si *X* es de tipo TIPO_T. Una **función mutadora** para actualizar un atributo establece el valor del atributo a un valor nuevo. SQL permite bloquear estas funciones para el uso público; necesitamos un privilegio EXECUTE para poder acceder a estas funciones.

En general, un UDT puede tener asociadas varias funciones definidas por el usuario. Su sintaxis es:

```
METHOD <nombre> (<lista_argumentos>) RETURNS <tipo>;
```

Podemos definir dos tipos de funciones: internas y externas. Las funciones internas se escriben en el lenguaje PSM extendido de SQL (consulte el Capítulo 9). Las funciones externas se escriben en un lenguaje *host*, con sólo su firma (interfaz) en la definición UDT. La definición de una función externa se puede declarar de este modo:

**DECLARE EXTERNAL** <nombre_función> <firma>
**LANGUAGE** <nombre_lenguaje>;

Muchos ORBDMSs han adoptado el método de definir un paquete de Tipos de datos abstractos (ADT) y las funciones asociadas para dominios de aplicación específicos. Éstos podrían adquirirse independientemente del sistema básico. Por ejemplo, Data Blades en Informix Universal Server, Data Cartridges en Oracle, y Extenders en DB2 se pueden considerar como paquetes o librerías de ADTs para dominios de aplicación específicos.

Los UDTs se pueden utilizar como tipos para los atributos en SQL y como tipos de parámetro en una función o procedimiento, y como un tipo de origen en un tipo distinto. La **equivalencia de tipo** se define en SQL a dos niveles. Dos tipos son **equivalentes en nombre** si, y sólo si, tienen el mismo nombre. Dos tipos son **equivalentes estructuralmente** si, y sólo si, tienen el mismo número de componentes y los componentes son equivalentes dos a dos en tipo.

Los atributos y las funciones en los UDTs están divididos en tres categorías:

- PUBLIC (visible en la interfaz UDT).
- PRIVATE (no visible en la interfaz UDT).
- PROTECTED (visible sólo a los subtipos).

También es posible definir atributos virtuales como parte de los UDTs, que se calculan y actualizan mediante funciones.

**Herencia y sobrecarga de funciones en SQL.** En el Capítulo 20 ya explicamos muchos de los principios de la herencia. SQL tiene reglas para tratarla (que se especifican con la palabra clave UNDER). Asociadas a la herencia encontramos las reglas que permiten sobrecargar las implementaciones de las funciones, y resolver los nombres de las funciones. Estas reglas se pueden resumir así:

- Todos los atributos se heredan.
- El orden de los supertipos en la cláusula UNDER determina la jerarquía de herencia.
- Una instancia de un subtipo puede utilizarse en todo contexto en el que se utilice una instancia del supertipo.
- Un subtipo puede redefinir cualquier función que se defina en su supertipo, con la restricción de que la firma sea la misma.
- Cuando se llama a una función, se selecciona la mejor coincidencia en base a los tipos de todos los argumentos.
- En la vinculación dinámica, se tienen en consideración los tipos de tiempo de ejecución de los parámetros.

El siguiente ejemplo ilustra la herencia de tipo. Supongamos que queremos crear un subtipo TIPO_DIRECTOR que hereda todos los atributos (y métodos) de TIPO_EMP, pero tiene un atributo adicional, Dpto_dirigido. Podemos escribir lo siguiente:

**CREATE TYPE** TIPO_DIRECTOR **UNDER** TIPO_EMP **AS** (Dpto_dirigido CHAR (20) );

Esto hereda todos los atributos y métodos del supertipo TIPO_EMP, y tiene un atributo específico adicional, Dpto_dirigido. También podríamos especificar métodos específicos adicionales para el subtipo.

Otra posibilidad que ofrece SQL es la instalación supertabla/subtabla, que se parece a la herencia de clase o EXTENDS de la que hablamos en el Capítulo 20. Aquí, una subtabla hereda cada columna de su supertabla; cada fila de una subtabla se corresponde a una, y sólo una, fila de la supertabla; cada fila de la supertabla se corresponde como máximo a una fila de una subtabla. Las operaciones INSERT, DELETE y UPDATE se propagan en consecuencia. Por ejemplo, supongamos la siguiente definición de la tabla INFO_INMUEBLES:

```
CREATE TABLE INFO_INMUEBLES (
        Propiedad    INMUEBLES,
        Propietario  CHAR(25),
        Precio IMPORTE );
```

Podemos definir las siguientes subtablas:

```
CREATE TABLE INMUEBLES_AMÉRICA UNDER INFO_INMUEBLES;
CREATE TABLE INMUEBLES_GEORGIA UNDER INMUEBLES_AMÉRICA;
CREATE TABLE INMUEBLES_ATLANTA UNDER INMUEBLES_GEORGIA;
```

En este ejemplo, cada tupla de la subtable INMUEBLES_AMÉRICA debe existir en su supertabla INFO_INMUEBLES; cada tupla de la subtabla INMUEBLES_GEORGIA debe existir en su supertabla INMUEBLES_AMÉRICA, etcétera. Sin embargo, puede haber tuplas en una supertabla que no existan en la subtabla.

**Objetos complejos no estructurados en SQL.** SQL dispone de tipos nuevos para los objetos grandes (LOB), y localizadores de objetos grandes. Existen dos variaciones para los objetos binarios grandes (BLOBs) y los objetos grandes de caracteres (CLOBs). SQL propone la manipulación de los LOBs dentro del DBMS sin necesidad de utilizar archivos externos. Determinados operadores no se aplican a los atributos de valor LOB; por ejemplo, comparaciones aritméticas, agrupaciones y ordenaciones. Por el contrario, la recuperación de un valor parcial, la comparación LIKE, la concatenación, la extracción de subcadenas, la posición y la longitud son operaciones que es posible aplicar a los LOBs. En la Sección 22.4 veremos cómo se utilizan los objetos grandes en Oracle 8.

Hemos ofrecido una visión general de los servicios de orientación a objetos propuestos en SQL. En este momento, se han estandarizado las especificaciones SQL/Foundations y SQL/Object. Es evidente que los servicios que hacen que SQL sea orientado a objetos sigan lo que se ha implementado en los ORDBMSs comerciales. SQL/MM (multimedia) se ha propuesto como un estándar independiente para la administración de bases de datos multimedia con varias partes: entramado, texto completo, espacial, servicios de propósito general e imagen fija. Explicaremos el uso de los tipos de datos bidimensionales y de las imágenes y el texto Data Blades en Informix Universal Server.

### 22.1.3 Algunas operaciones y características nuevas de SQL

Una operación nueva e importante es la **recursividad lineal** para poder crear consultas recursivas. A fin de ilustrarlo, supongamos que tenemos una tabla denominada TABLA_PIEZAS(Parte1, Parte2), que contiene una tupla $<p_1, p_2>$ siempre que la pieza $p_1$ contiene la pieza $p_2$ como un componente. Una consulta para producir la **factura de los materiales** de alguna pieza $p_1$ (es decir, todas las piezas necesarias para producir $p_1$) se escribe como una consulta recursiva, de este modo:

```
WITH RECURSIVE
FACTURA_MATERIAL (Parte1, Parte2) AS
        ( SELECT Parte1, Parte2
          FROM TABLA_PIEZAS
          WHERE Parte1 = 'p1'
                UNION ALL
          SELECT FACTURA_MATERIAL.Parte1, TABLA_PIEZAS.Parte2
          FROM FACTURA_MATERIAL, TABLA_PIEZAS
          WHERE TABLA_PIEZAS.Parte1 = FACTURA_MATERIAL.Parte2 )
SELECT * FROM FACTURA_MATERIAL
ORDER BY Parte1, Parte2;
```

El resultado final queda en FACTURA_MATERIAL(Parte1, Parte2). La operación UNION ALL se evalúa realizando una unión de todas las tuplas generadas por el bloque interno hasta que dejen de generarse tuplas nuevas. Como SQL2 carece de recursividad, el programador debe llevarla a cabo con la iteración apropiada.

Por seguridad, en SQL3 se introdujo el concepto de **rol**, que se parece a una *descripción de trabajo* y está sujeto a la autorización de privilegios. Las personas actuales (cuentas de usuario) asignadas a un rol pueden cambiar, pero la autorización de rol en sí mismo no tiene que cambiar. SQL3 también incluye una sintaxis para especificar y utilizar los ***triggers*** (consulte el Capítulo 24) como reglas activas. La activación o disparo de eventos incluye las operaciones INSERT, DELETE y UPDATE en un tabla. Es posible especificar el *trigger* para que sea considerado BEFORE (anterior) o AFTER (posterior) al evento de activación. En SQL3 se incluye el concepto de **granularidad de *trigger***, que permite especificar un *trigger* a nivel de fila (el *trigger* se considera para cada fila afectada) o un *trigger* a nivel de sentencia (el *trigger* se considera sólo una vez para cada evento de activación).[3] En las bases de datos distribuidas (cliente-servidor) (consulte el Capítulo 25), se incluye el concepto de **módulo cliente** en SQL3. Un módulo cliente puede contener procedimientos, cursores y tablas temporales invocados externamente, lo que puede especificarse con la sintaxis SQL3.

SQL3 también se está ampliando con servicios de lenguaje de programación. Las rutinas escritas en SQL/CLI con una coincidencia total de los tipos de datos y un entorno integrado se conocen como **rutinas SQL**. Para que el lenguaje sea computacionalmente completo, en la sintaxis de SQL3 se incluyen las siguientes estructuras de control de la programación: CALL/RETURN, BEGIN/END, FOR/END_FOR, IF/THEN/ELSE/END_IF, CASE/END_CASE, LOOP/END_LOOP, WHILE/END_WHILE, REPEAT/UNTIL/END_REPEAT y LEAVE. Las variables se declaran con DECLARE, y las asignaciones se especifican con SET. Las **rutinas externas** referidas a programas escritos en un lenguaje *host* (ADA, C, COBOL, PASCAL, etcétera), posiblemente tienen SQL incrustado y posibles desigualdades en los tipos. La ventaja de las rutinas externas es que existen librerías de dichas rutinas que se utilizan ampliamente, lo que puede reducir mucho el esfuerzo de implementación de las aplicaciones. Por el contrario, las rutinas SQL son más *puras*, pero no se utilizan tan ampliamente. Estas rutinas se pueden utilizar para las rutinas de servidor (rutinas a nivel de esquema o módulos) o como módulos cliente, y pueden ser procedimientos o funciones que devuelven valores. SQL/CLI se describe en el Capítulo 9.

## 22.2 Evolución de los modelos de datos y tendencias actuales de la tecnología de bases de datos

En el ámbito comercial mundial hay varias familias de productos DBMS. Dos de las más importantes son los RDBMS y los ODBMS, que se refieren a los modelos de datos relacional y de objetos, respectivamente. Los otros dos tipos más importantes de productos DBMS (jerárquicos y de red) ahora se conocen como **DBMSs heredados**; están basados en los modelos de datos jerárquico y de red, y los dos aparecieron a mediados de la década de 1960. La familia jerárquica cuenta con un producto destacado, IMS de IBM, mientras que la familia de red dispone de una gran cantidad de DBMSs, como IDS II (Honeywell), IDMS (Computer Associates), IMAGE (Hewlett Packard), VAX-DBMS (Digital) y TOTAL/SUPRA (Cincom), por citar unos cuantos. Los modelos de datos jerárquico y de red se resumen en los Apéndices D y E.[4]

A medida que evoluciona la tecnología de bases de datos, se van reemplazando gradualmente los DBMSs heredados por ofertas más nuevas. Por ahora, nos tenemos que enfrentar a un problema de **interoperabilidad**

[3] Estos conceptos se explican más en profundidad en el Capítulo 24.

[4] Los capítulos de la edición anterior dedicados al modelo de datos de red y al modelo de datos jerárquico están disponibles en el sitio web de este libro.

importante (la interoperación de varias bases de datos que pertenecen a familias de DBMSs distintas), así como a los sistemas de administración de ficheros heredados. También están apareciendo series completas de sistemas y herramientas nuevas para afrontar este problema. Más recientemente, XML ha surgido como un estándar nuevo para el intercambio de datos por la Web (consulte el Capítulo 27).

Las principales fuerzas que hay detrás del desarrollo de ORDBMSs extendidos proceden de la incapacidad de los DBMSs heredados y del modelo de datos relacional, así como de los RDBMSs más antiguos, de satisfacer los desafíos de las aplicaciones nuevas. Esto se da principalmente en áreas donde se utilizan varios tipos de datos; por ejemplo, texto en la maquetación; imágenes en el procesamiento de imágenes de satélite o de pronóstico del tiempo; datos complejos no convencionales en los diseños de ingeniería, información biológica del genoma y dibujos arquitectónicos; datos de series temporales en las transacciones del mercado de valores; y datos espaciales y geográficos en los mapas, datos sobre la contaminación del aire y el agua, y datos del tráfico. Por tanto, hay una necesidad clara de diseñar bases de datos que puedan desarrollar, manipular y mantener los objetos complejos que surgen de aplicaciones como las mencionadas. Es más, cada vez es más necesaria la manipulación de información digitalizada que representa flujos de datos de audio y vídeo (particionados en fotogramas individuales) que, a su vez, requiere el almacenamiento de BLOBs (objetos binarios grandes) en los DBMSs.

La popularidad del modelo relacional está ayudada por una infraestructura muy robusta de los DBMSs comerciales que se han diseñado para soportarlo. Sin embargo, el modelo relacional básico y las versiones antiguas de su lenguaje SQL no son adecuados para enfrentarse a los desafíos mencionados. Los modelos de datos heredados, como el modelo de datos de red, tienen un servicio para modelar las relaciones explícitamente, pero sufren un uso abusivo de punteros en la implementación, y no tienen conceptos como la identidad de objeto, la herencia, la encapsulación o el soporte de varios tipos de datos y objetos complejos. El modelo jerárquico se adapta mejor a algunas de las jerarquías que se dan en la vida real y en las empresas, pero está muy limitado respecto a las rutas jerárquicas integradas en los datos. Por tanto, empezó a surgir una tendencia a combinar las mejores características del modelo de datos de objeto y los lenguajes en el modelo relacional, a fin de ampliarlo con las aplicaciones actuales.

En el resto del capítulo destacaremos las características de dos DBMSs representativos que ejemplifican la metodología ORDBMS: Informix Universal Server y Oracle 8. Concluimos con una explicación breve del modelo relacional anidado, que tiene su origen en una serie de propósitos de investigación e implementaciones prototipo; esto nos ofrece un marco teórico de incrustación de objetos complejos jerárquicamente estructurados dentro del marco relacional.

## 22.3 Informix Universal Server[5]

Informix Universal Server es un ORDBMS que combina las tecnologías de bases de datos relacional y de objetos de los productos existentes anteriormente: Informix e Illustra. Este último se originó a partir del DBMS POSTGRES, que era un proyecto de investigación de la Universidad de California en Berkeley que se comercializó como Montage DBMS y que se bautizó como Miro antes de renombrarlo como Illustra. Illustra fue entonces adquirido por Informix, integrado en su RDBMS e introducido como Informix Universal Server, un ORDBMS. El producto actual, después de la adquisición de Informix por IBM, se denominó IBM Informix Dynamic Server.

Para ver por qué emergieron los ORDBMSs, empezaremos centrándonos en cómo clasificar las aplicaciones DBMS de acuerdo con dos dimensiones o ejes: complejidad de los datos (la dimensión *X*) y complejidad de las consultas (la dimensión *Y*). Podemos disponer estos ejes en un espacio 0-1 sencillo con cuatro cuadrantes:

[5] La explicación de esta sección está principalmente basada en el libro *Object-Relational DBMS*s, de Michael Stonebraker y Dorothy Moore (1996), y en la información proporcionada por Magdi Morsi. Nuestra explicación puede referirse a versiones de Informix que no son necesariamente las más recientes.

Cuadrante 1 ($X = 0$, $Y = 0$): datos simples, consultas simples.

Cuadrante 2 ($X = 0$, $Y = 1$): datos simples, consultas complejas.

Cuadrante 3 ($X = 1$, $Y = 0$): datos complejos, consultas simples.

Cuadrante 4 ($X = 1$, $Y = 1$): datos complejos, consultas complejas.

Los RDBMSs tradicionales pertenecen al Cuadrante 2. Aunque soportan consultas y actualizaciones temporalmente complejas (así como el procesamiento de transacciones), sólo pueden tratar con datos simples que puedan modelarse como un conjunto de filas en una tabla. Muchas bases de datos de objetos (ODBMSs) encajan en el Cuadrante 3, puesto que se concentran en administrar datos complejos, pero tienen unas capacidades de consulta basadas en la navegación algo limitadas.[6]

A fin de moverse al Cuadrante 4 y así soportar datos y consultas complejos, los RDBMSs han ido incorporando más objetos de datos complejos, mientras que los ODBMSs han ido incorporando más consultas complejas (por ejemplo, el lenguaje de consulta de alto nivel OQL, explicado en el Capítulo 21). Informix Universal Server pertenece al Cuadrante 4 porque ha ampliado su modelo relacional básico mediante la incorporación de varias características que lo convierten en objeto-relacional.

Otros ORDBMSs actuales que evolucionaron a partir de los RDBMSs son Oracle de Oracle Corporation, Universal DB (UDB) de IBM, Odapter de Hewlett Packard (HP) (que extiende el DBMS de Oracle) y Open ODB de HP (que extiende el producto Allbase/SQL de HP). Los productos más exitosos parecen ser aquellos que mantienen la opción de trabajar como un RDBMS a la vez que introducen funcionalidades adicionales. Nuestra intención no es ofrecer un análisis comparativo de estos productos, sino únicamente ofrecer una visión general de dos sistemas representativos.

**Cómo Informix Universal Server extiende el modelo de datos relacional.** Las extensiones al modelo de datos relacional proporcionadas por Illustra e incorporadas en Informix Universal Server encajan en las siguientes categorías:

- Soporte de tipos de datos adicionales o extensibles.
- Soporte para rutinas definidas por el usuario (procedimientos o funciones).
- Noción implícita de la herencia.
- Soporte para indexar extensiones.
- API (Interfaz de programación de aplicaciones) Data Blades.[7]

En las siguientes secciones ofrecemos una visión general de estas características. Ya hemos introducido, de una forma general, los conceptos de tipos de datos, constructores de tipos, objetos complejos y herencia en el contexto de los modelos orientados a objetos (consulte el Capítulo 20).

### 22.3.1 Tipos de datos extensibles

La arquitectura de Informix Universal Server abarca el DBMS básico más varios **módulos Data Blade** (palas de datos). La idea es tratar el DBMS como una maquinilla de afeitar en la que se inserta una cuchilla en particular para dar soporte a un tipo de datos específico. Se proporcionan así varios tipos de datos, entre los que cabe citar objetos geométricos bidimensionales (puntos, líneas, círculos y elipses, por ejemplo), imágenes,

[6] El Cuadrante 1 incluye paquetes de software que tratan la manipulación de datos sin características sofisticadas de recuperación y manipulación de datos. Nos referimos a hojas de cálculo como Microsoft Excel, procesadores de texto como Microsoft Word, o cualquier aplicación de administración de ficheros.

[7] Data Blades suministra extensiones al sistema básico, como explicamos en la Sección 22.3.6.

series temporales, texto y páginas web. Cuando Informix anunció Universal Server, ya había disponibles 29 Data Blades.[8]

También es posible para una aplicación crear sus propios tipos, lo que hace que la noción de tipo de datos sea completamente extensible. Además de los tipos integrados, Informix Universal Server proporciona al usuario los siguientes constructores para declarar tipos adicionales:

1. Tipo opaco.
2. Tipo distinto.
3. Tipo fila.
4. Tipo colección.

Al crear un tipo basado en una de las tres primeras opciones, el usuario tiene que proporcionar funciones y rutinas para la manipulación y la conversión, incluyendo las funciones integradas, de agregación y de operador, así como las funciones y rutinas adicionales definidas por el usuario. Los detalles de estos tipos se presentan en las siguientes secciones.

**Tipo opaco.** El tipo opaco (opaque) tiene oculta su representación interna, por lo que se utiliza para encapsular un tipo. El usuario tiene que proporcionar funciones de formateo para convertir un objeto opaco entre su representación oculta en el servidor (base de datos) y su representación visible, como lo ve el cliente (programa de llamada). Es necesario *enviar/recibir* funciones de usuario para convertir a/de la representación interna de servidor de/a la representación cliente. De forma parecida, las funciones de *importación/exportación* se utilizan para convertir a/de una representación externa para copia bulk de/a la representación interna. Se pueden definir otras funciones para procesar los tipos opacos, incluyendo assign(), destroy() y compare().

La especificación de un tipo opaco incluye su nombre, la longitud interna si es fija, la longitud interna máxima si es una longitud variable, la alineación (que es el límite byte), y si se puede o no dispersar (para crear una estructura de acceso *hash*). Si escribimos:

```
CREATE OPAQUE TYPE UDT_OPACO_FIJO (INTERNALLENGTH = 8,
    ALIGNMENT = 4, CANNOTHASH);

CREATE OPAQUE TYPE UDT_OPACO_VAR (INTERNALLENGTH = variable,
    MAXLEN=1024, ALIGNMENT = 8);
```

la primera sentencia crea un tipo opaco definido por el usuario y de longitud fija, UDT_OPACO_FIJO, mientras que la segunda sentencia crea uno de longitud variable, UDT_OPACO_VAR. Los dos se describen en una implementación con parámetros internos que no son visibles al cliente.

**Tipo distinto.** El tipo de datos distinct se utiliza para extender un tipo existente a través de la herencia. El tipo recién definido hereda las funciones/rutinas de su tipo base, si no se sobrescriben. Por ejemplo, la sentencia:

```
CREATE DISTINCT TYPE FECHA_CONTRATO AS DATE;
```

crea un nuevo tipo definido por el usuario, FECHA_CONTRATO, que puede utilizarse como cualquier otro tipo integrado.

**Tipo fila.** El tipo row (fila), que representa un atributo compuesto, es análogo a un tipo struct en el lenguaje de programación C.[9] Es un tipo compuesto que contiene uno o más campos. El tipo fila también se utiliza para dar soporte a la herencia, mediante la palabra clave UNDER, pero el sistema de tipos sólo soporta la herencia

[8] Si desea más información sobre Data Blades para Informix Universal Server, consulte el sitio web: http://www-306.ibm.com/software/data/informix/blades/

[9] Esto se parece al *constructor de tupla* explicado en el Capítulo 20.

simple. Mediante la creación de tablas cuyas tuplas son de un tipo fila particular, es posible tratar una relación como parte de un esquema orientado a objetos y establecer relaciones de herencia entre las relaciones. En las siguientes declaraciones de tipos fila, EMPLEADO_T y ESTUDIANTE_T heredan (o están *declarados bajo*) PERSONA_T:

```
CREATE ROW TYPE PERSONA_T(Nombre VARCHAR(60),
    Dni NUMERIC(9), Fecha_nac DATE);

CREATE ROW TYPE EMPLEADO_T(Sueldo NUMERIC(10,2), Contratado_en
    FECHA_CONTRATO) UNDER PERSONA_T;

CREATE ROW TYPE ESTUDIANTE_T(Media NUMERIC(4,2), Direcc
    VARCHAR(200)) UNDER PERSONA_T;
```

**Tipo colección.** Las colecciones de Informix Universal Server incluyen listas, conjuntos y multiconjuntos (bolsas) de tipos integrados, así como tipos definidos por el usuario.[10] Una colección puede ser del tipo de cualquier campo de un tipo fila o de una columna de una tabla. Los elementos de una colección **set** (conjunto) no pueden contener valores duplicados, y no tienen un orden concreto. Una **lista** puede contener elementos duplicados y el orden es significativo. Por último, el multiconjunto (**multiset**) puede incluir duplicados y no tiene un orden específico. Consideremos el siguiente ejemplo:

```
CREATE TABLE EMPLEADO (Nombre VARCHAR(50) NOT NULL,
    Comisión MULTISET(IMPORTE) );
```

Aquí, la tabla EMPLEADO contiene la columna Comisión, que es de tipo multiconjunto.

## 22.3.2 Soporte de rutinas definidas por el usuario

Informix Universal Server soporta las funciones y las rutinas definidas por el usuario para manipular los tipos definidos por el usuario. La implementación de dichas funciones se puede realizar en SPL (Lenguaje de procedimiento almacenado), en C o en Java. Las funciones definidas por el usuario nos permiten definir funciones de operador como plus(), minus(), times(), divide(), positive() y negate(), funciones integradas como cos() y sin(), funciones de agregación como sum() y avg(), y rutinas definidas por el usuario. De este modo, Informix Universal Server puede manipular los tipos definidos por el usuario como si fueran tipos integrados siempre que se hayan definido las funciones necesarias. El siguiente ejemplo especifica una función de igualdad para comparar dos objetos del tipo UDT_OPACO_FIJO declarado anteriormente:

```
CREATE FUNCTION igual (Arg1 UDT_OPACO_FIJO, Arg2
    UDT_OPACO_FIJO) RETURNING BOOLEAN;
EXTERNAL NAME "/usr/lib/informix/libopaque.so
    (fixed_opaque_udt_equal)" LANGUAGE C;
END FUNCTION;
```

Informix Universal Server también soporta **cast**, una función que convierte objetos de un tipo de origen a un tipo de destino. Hay dos tipos de conversiones definidas por el usuario: implícitas y explícitas. Las conversiones implícitas se invocan automáticamente, mientras que las explícitas sólo se invocan cuando se especifica explícitamente el operador cast mediante "::" o CAST AS. Si los tipos de origen y de destino tienen la misma estructura interna (como al utilizar la especificación de *tipos distintos*), no se necesitan funciones definidas por el usuario.

Consideremos el siguiente ejemplo para ilustrar la conversión explícita, donde la tabla de empleados tiene una columna Col1 de tipo UDT_OPACO_VAR y una columna Col2 de tipo UDT_OPACO_FIJO.

[10] Son parecidos a los *tipos colección* explicados en los Capítulos 20 y 21.

```
SELECT Col1 FROM EMPLEADO WHERE UDT_OPACO_FIJO::Col1 = Col2;
```

Con el fin de comparar Col1 con Col2, se aplica el operador cast a Col1 para convertirla de UDT_OPACO_VAR a UDT_OPACO_FIJO.

### 22.3.3 Soporte de la herencia

La herencia se controla a dos niveles en Informix Universal Server: herencia de datos (atributo) y herencia de función (operación).

**Herencia de datos.** Para crear subtipos bajo los tipos fila existentes, utilizamos la palabra clave UNDER, como explicamos anteriormente. Considere el siguiente ejemplo:

```
CREATE ROW TYPE TIPO_EMPLEADO (
        NombreE VARCHAR(25),
        Dni CHAR(9),
        Sueldo INT) ;
CREATE ROW TYPE TIPO_INGENIERO (
        Grado VARCHAR(10),
        Licencia VARCHAR(20) )
        UNDER TIPO_EMPLEADO;
CREATE ROW TYPE TIPO_DIRE_ING (
        Fecha_inicio_director VARCHAR(10),
        Dpto_dirigido VARCHAR(20) )
        UNDER TIPO_INGENIERO;
```

Las sentencias anteriores crean un TIPO_EMPLEADO y un subtipo denominado TIPO_INGENIERO, que representa a los empleados que son ingenieros y, por tanto, heredan todos los atributos de los empleados; tiene, además, dos propiedades adicionales, Grado y Licencia. El tipo TIPO_DIRE_ING es un subtipo bajo TIPO_INGENIERO y, por tanto, hereda de TIPO_INGENIERO e, implícitamente, también de TIPO_EMPLEADO. Informix Universal Server no soporta la herencia múltiple. Ahora podemos crear las tablas EMPLEADO, INGENIERO y DIRE_ING basándonos en estos tipos fila.

Las opciones de almacenamiento para almacenar las jerarquías de tipos en las tablas varían. Informix Universal Server ofrece la opción de almacenar instancias en diferentes combinaciones (por ejemplo, una instancia [registro] en cada nivel, o una instancia que se consolida a todos los niveles), que se corresponden con las opciones de mapeado de la Sección 7.2. Los atributos heredados o se representan repetidamente en las tablas en los niveles inferiores, o se representan con una referencia al objeto del supertipo. El procesamiento de los comandos SQL se modifica en consecuencia según la jerarquía de tipos. Por ejemplo, la consulta:

```
SELECT    *
FROM      EMPLEADO
WHERE     Sueldo > 100000;
```

devuelve la información de empleado de *todas las tablas* donde están representados los empleados seleccionados. De este modo, el ámbito de la tabla EMPLEADO se extiende a todas las tuplas bajo EMPLEADO. De forma predeterminada, las consultas sobre la supertabla devuelven columnas de la supertabla, así como de las subtablas que heredan de la supertabla. Por el contrario, la siguiente consulta sólo devuelve instancias de la tabla EMPLEADO debido al uso de ONLY:

```
SELECT    *
FROM ONLY (EMPLEADO)
WHERE     Sueldo > 100000;
```

Es posible consultar una supertabla utilizando una *variable de correlación* para que el resultado no sólo contenga columnas de tipo supertabla de las subtablas, sino también columnas específicas del subtipo de las subtablas. Una consulta semejante devuelve filas de diferentes tamaños. La recuperación de toda la información sobre un empleado a todos los niveles se acomete de este modo:

```
SELECT    E
FROM      EMPLEADO E ;
```

Para cada empleado, en función de que sea ingeniero o de algún otro subtipo(s), devolverá conjuntos adicionales de atributos de las tablas subtipo apropiadas.

Las vistas definidas sobre supertablas no se pueden actualizar porque la colocación de las filas insertadas es ambigua.

**Herencia de función**. De la misma forma que se pueden heredar datos entre las tablas junto con la jerarquía de tipos, en los ORDBMSs es posible heredar también las funciones. Por ejemplo, podemos definir la función sobrepagado en TIPO_EMPLEADO para seleccionar los empleados que tienen un sueldo superior al de Luis Campos:

```
CREATE FUNCTION sobrepagado (TIPO_EMPLEADO)
RETURNS BOOLEAN AS
RETURN $1.Sueldo > ( SELECT   Sueldo
                     FROM     EMPLEADO
                     WHERE    NombreE = 'Luis Campos' );
```

Las tablas bajo la tabla EMPLEADO heredan automáticamente esta función. Sin embargo, es posible redefinir la misma función para TIPO_DIRE_ING para los empleados que tienen un sueldo superior al de Juan Martínez:

```
CREATE FUNCTION sobrepagado (TIPO_DIRE_ING)
RETURNS BOOLEAN AS
RETURN $1.Sueldo > ( SELECT   Sueldo
                     FROM     EMPLEADO
                     WHERE    NombreE = 'Juan Martínez');
```

Por ejemplo, la siguiente consulta:

```
SELECT    E.NombreE
FROM ONLY (EMPLEADO) E
WHERE     sobrepagado (E);
```

se evalúa con la primera definición de sobrepagado. La consulta:

```
SELECT    G.NombreE
FROM      INGENIERO G
WHERE     verpagado (G);
```

también utiliza la primera definición de sobrepagado (porque no se redefinió para ingeniero), mientras que:

```
SELECT    GM.NombreE
FROM      DIRE_ING GM
WHERE     sobrepagado (GM);
```

utiliza la segunda definición de sobrepagado, que sobrescribe la primera. Es lo que se conoce como **sobrecarga de la operación** (o de la **función**), como explicamos en la Sección 20.6 cuando hablábamos del poli-

morfismo. La función sobrepagado (y otras funciones) también pueden tratarse como *atributos virtuales*; por tanto, a sobrepagado podemos referirnos como EMPLEADO.sobrepagado o DIRE_ING.sobrepagado en una consulta.

### 22.3.4 Soporte para las extensiones de indexación

Informix Universal Server soporta la indexación sobre las rutinas definidas por el usuario en tablas sencillas o en jerarquías de tablas. Por ejemplo:

```
CREATE INDEX CIUDAD_EMPL ON EMPLEADO (Ciudad (Direcc) );
```

crea un índice en la tabla EMPLEADO utilizando el valor de la función Ciudad.

A fin de soportar los índices definidos por el usuario, Informix Universal Server soporta las clases de operador, que se utilizan para dar soporte a los tipos de datos definidos por el usuario en el árbol B genérico, así como en otros métodos de acceso secundarios, como los árboles R.

### 22.3.5 Soporte de orígenes de datos externos

Informix Universal Server soporta orígenes de datos externos (por ejemplo, los datos almacenados en un sistema de ficheros) que están mapeados a una tabla de la base de datos denominada **interfaz de tabla virtual**. Esta interfaz permite al usuario definir operaciones que se pueden utilizar como *proxies* para las demás operaciones, que son necesarias para acceder y manipular la fila o filas asociadas con el origen de datos subyacente. Estas operaciones permiten abrir, cerrar, buscar, insertar y eliminar. Informix Universal Server también soporta un conjunto de funciones que permiten llamar a sentencias SQL dentro de una rutina definida por el usuario, sin la incomodidad de pasar por una interfaz cliente.

### 22.3.6 Soporte de la interfaz de programación de aplicaciones Data Blades

La Interfaz de programación de aplicaciones (API) Data Blades de Informix Universal Server proporciona nuevos tipos de datos y funciones para tipos específicos de aplicaciones. Estudiaremos los tipos de datos extensibles para las operaciones de dos dimensiones (necesarias en las aplicaciones GIS o CAD),[11] los tipos de datos relacionados con el almacenamiento y gestión de imágenes, los tipos de datos de series temporales, y unas cuantas características de los tipos de datos de texto. La fuerza de los ORDBMSs para tratar con las nuevas aplicaciones no convencionales se atribuye normalmente a estos tipos de datos especiales y la funcionalidad a medida que ofrecen.

**Tipos de datos bidimensionales (espaciales).** Para una aplicación bidimensional, podemos tener los siguientes tipos de datos pertinentes:

- Un **punto** definido por las coordenadas (*X*, *Y*).
- Una **línea** definida por sus dos puntos finales.
- Un **polígono** definido por una lista ordenada de *n* puntos que forman sus vértices.
- Una **ruta** definida por una secuencia (lista ordenada) de puntos.
- Un **círculo** definido por su punto central y su radio.

[11] GIS significa *Geographic Information Systems* (Sistemas de información geográfica) y CAD significa *Computer Aided Design* (Diseño asistido por computador). GIS se explica en la Sección 30.3.

Dados los tipos de datos anteriores, una función como la *distancia* puede definirse entre dos puntos, un punto y una línea, una línea y un círculo, etcétera, implementando las expresiones matemáticas adecuadas a la distancia en un lenguaje de programación. De forma parecida, una función de intersección booleana (que devuelve verdadero o falso en función de si dos objetos geométricos se cruzan o intersectan) puede definirse entre una línea y un polígono, una ruta y un polígono, una línea y un círculo, etcétera. Existen otras funciones booleanas relativas a las aplicaciones GIS, como la *superposición* (polígono, polígono) o *contiene* (polígono, polígono), *contiene* (punto, polígono), etcétera. El concepto de sobrecarga (polimorfismo de operación) se aplica cuando el mismo nombre de función se utiliza con diferentes tipos de argumentos.

**Tipos de datos de imagen.** Las imágenes se almacenan en diferentes formatos estándar, como *tiff*, *gif*, *jpeg*, *photoCD*, *Group 4* y *FAX* (así que podemos definir un tipo de datos para cada uno de estos formatos y utilizar las funciones apropiadas de la librería para introducir imágenes de otros medios o para representar imágenes para su visualización). Alternativamente, IMAGE puede considerarse como un tipo de datos sencillo con un gran número de opciones para el almacenamiento de datos. La última opción permitiría que una columna de una tabla fuera de tipo IMAGE y aceptar imágenes de varios formatos. A continuación mostramos algunas de las funciones (operaciones) posibles con imágenes:

```
rotate (imagen, ángulo) returns image.
crop (imagen, polígono) returns image.
enhance (imagen) returns image.
```

La función crop extrae la porción de una imagen que intersecta con un polígono. La función enhance mejora la calidad de una imagen mediante la mejora del contraste. En las siguientes funciones se pueden suministrar varias imágenes como parámetros:

```
common (imagen1, imagen2) returns image.
union (imagen1, imagen2) returns image.
similarity (imagen1, imagen2) returns number.
```

La función similarity normalmente tiene en cuenta la distancia entre dos vectores con los componentes <color, forma, textura, borde> que describen el contenido de las dos imágenes. El VIR Data Blade de Informix Universal Server puede realizar una búsqueda por contenido en las imágenes basándose en la medida de similitud anterior.

**Tipos de datos de series temporales.** Informix Universal Server soporta un tipo de datos de series temporales que simplifican mucho la manipulación de datos de series temporales que se almacenan en varias tablas. Por ejemplo, consideremos el almacenamiento del precio de cierre de los más de 3.000 títulos de la Bolsa de Nueva York por cada día laborable cuando abre el mercado. Una tabla semejante puede definirse de de este modo:

```
CREATE TABLE PRECIOS_ACCIÓN (
        Nombre_empresa      VARCHAR(30),
        Símbolo             VARCHAR(5),
        Precios             TIME_SERIES OF FLOAT
);
```

Respecto al precio de acción de las 3.000 empresas por un periodo de tiempo completo de, por ejemplo, varios años, sólo una relación es adecuada gracias a los tipos de datos de series temporales para el atributo Precios. Sin estos tipos de datos, cada empresa necesitaría una tabla. Por ejemplo, la empresa Coca-Cola necesitaría una tabla que podría declararse de la siguiente forma:

```
CREATE TABLE COCA_COLA (
        Fecha_registro      DATE,
```

```
    Precio          FLOAT );
```

En esta tabla habría aproximadamente 260 tuplas por año, una por cada día de cotización. Los tipos de datos de series temporales tienen en cuenta el calendario, la hora de inicio, el intervalo de registro (por ejemplo, diario, semanal, mensual), etcétera. Son apropiadas funciones como la extracción de un subconjunto de series temporales (por ejemplo, los precios de cierre durante enero de 1999), el resumen con una granularidad mayor (por ejemplo, precio de cierre medio semanal a partir de los precios de cierre diarios), y la creación de promedios variables.

Una consulta sobre la tabla de precios de las acciones que obtiene el promedio variable para 30 días empezando el 1 de junio de 1999 para las acciones de Coca-Cola puede utilizar la función promedio_var:

```
SELECT    promedio_var(Precios, 30, '1999-06-01')
FROM      PRECIOS_ACCIÓN
WHERE     Simbolo = "KO";
```

La misma consulta en SQL sobre la tabla coca_cola es mucho más compleja de escribir y debe acceder a varias tuplas, mientras que la consulta anterior sobre la tabla de precios de las acciones trata con una sola fila de la tabla correspondiente a la empresa en cuestión. El uso de los tipos de datos de series temporales proporciona un aumento del rendimiento del orden de magnitud en el procesamiento de dichas consultas.

**Tipos de datos de texto.** Text DataBlade soporta el almacenamiento, la búsqueda y la recuperación de objetos de texto. Define un tipo de datos simple denominado **DOC**, cuyas instancias se almacenan como objetos grandes que pertenecen al tipo de datos integrado LARGE-TEXT. Explicaremos brevemente unas cuantas características importantes de este tipo de datos.

El almacenamiento subyacente para large-text es el mismo que para el tipo de datos large-object. Las referencias a un objeto grande simple se registran en la tabla REFCOUNT del sistema, que almacena información como el número de filas que se refieren al objeto grande, su OID, su gestor de almacenamiento, su hora de modificación y su gestor de almacenamiento de archivo. La conversión automática entre los tipos de datos LARGE-TEXT y text permite que cualquier función con argumentos de texto se pueda aplicar a objetos LARGE-TEXT. De este modo, es posible la concatenación de objetos LARGE-TEXT como cadenas, así como la extracción de subcadenas de un objeto LARGE-TEXT.

Los parámetros Text DataBlade incluyen un formato para ASCII, con otras posibilidades como Postscript, Dvipostscript, Nroff, Troff y Text. Se necesita Text Conversion DataBlade, que es independiente de Text DataBlade, para convertir los documentos entre varios formatos. Un parámetro External File instruye a la representación interna de doc a almacenar un puntero a un fichero externo, en lugar de copiarlo en un objeto grande.

En la manipulación de objetos de documento se utilizan funciones como las siguientes:

```
import_doc (doc, text) returns doc.
export_doc (doc, text) returns text.
assign (doc) returns doc.
destroy (doc) returns void.
```

Las funciones de asignación y destrucción ya existen para los tipos de datos LARGE-OBJECT y LARGE-TEXT integrados, pero deben redefinirse para los objetos de tipo doc. La siguiente sentencia crea una tabla denominada DOCUMENTOS_LEGALES, en la que cada fila tiene un título de documento en una columna y el documento propiamente dicho en otra columna.

```
CREATE TABLE DOCUMENTOS_LEGALES (
    Título      TEXT,
    Documento   DOC );
```

Para insertar una fila nueva en esta tabla para un documento denominado 'lease.contract', habría que utilizar la siguiente sentencia:

```
INSERT INTO DOCUMENTOS_LEGALES (Título, Documento)
VALUES ('lease.contract', 'format {troff}:/user/local/
        documents/lease');
```

El segundo valor de la cláusula values es el nombre de la ruta de acceso que especifica la ubicación del fichero de este documento; la especificación del formato significa que es un documento TROFF. Para buscar el texto es preciso crear un índice, como en la siguiente sentencia:

```
CREATE INDEX ÍNDICE_LEGAL
ON           DOCUMENTOS_LEGALES
USING        DTREE (Documento TEXT_OPS);
```

En la sentencia anterior, TEXT_OPS es una *op-class* (clase operador) aplicable a una estructura de acceso denominada índice dtree, que es una estructura de índice especial para los documentos. Cuando en una tabla se inserta un documento del tipo de datos doc, el texto se analiza por palabras individuales. El texto DataBlade no hace distinción entre mayúsculas y minúsculas; por tanto, Tlfcasa, TlfCasa o tlfcasa se consideran la misma palabra. Las palabras se *reducen* según el diccionario de sinónimos WORDNET de lengua inglesa. Por ejemplo, *houses* o *housing* se reduce a *house*, *quickly* a *quick*, y *talked* a *talk*. Se guarda un archivo **stopword**, que contiene las palabras insignificantes, como los artículos o las preposiciones que se ignoran en las búsquedas.

Informix Universal Server proporciona dos conjuntos de rutinas (rutinas "contiene" y rutinas "texto-cadena") para que las aplicaciones determinen los documentos que contienen una determinada palabra o palabras y los documentos que son parecidos. Al utilizar estas funciones en una condición de búsqueda, los datos son devueltos en orden descendente del grado de coincidencia, mostrándose en primer lugar los documentos que ofrecen el mayor grado de coincidencia. Hay una función WeightContains(índice a usar, id de tupla del documento, cadena de entrada) y una función WeightContainsWords parecida que devuelven un número de precisión entre 0 y 1, que indica la precisión de coincidencia entre la cadena o las palabras introducidas y el documento específico para este id de tupla. Para ilustrar el uso de estas funciones, observe la siguiente consulta, que localiza los títulos de los documentos legales que contienen los diez primeros términos del documento titulado 'lease contract', y que puede especificarse de este modo:

```
SELECT   D.Título
FROM     DOCUMENTOS_LEGALES D, DOCUMENTOS_LEGALES L
WHERE    CONTAINS (D.Documento, AndTerms (TopNTerms(L.Documento,10)))
         AND L.Título = 'lease.contract' AND D.Titulo <> 'lease.contract';
```

Esta consulta ilustra cómo puede mejorarse SQL con estas funciones específicas de estos tipos de datos para disponer de unas funciones de manipulación de texto más poderosas. En esta consulta, la variable *D* se refiere al cuerpo legal entero, mientras que *L* se refiere al documento específico cuyo título es 'lease.contract'. TopNTerms extrae los diez primeros términos del documento 'lease.contract' (*L*); AndTerms combina estos términos en una lista; y contains compara los términos de esa lista con las palabras reducidas de cualquier otro documento (*D*) de la tabla DOCUMENTOS_LEGALES.

**Resumen de Data Blades.** Como puede ver, Data Blades mejora un RDBMS al suministrar varios constructores para los tipos de datos abstractos (ADTs) que permiten operar con dichos datos como si estuvieran almacenados en un ODBMS utilizando los ADTs como clases. Esto hace que el sistema relacional se comporte como un ODBMS, y reduce drásticamente el esfuerzo de programación necesario, si lo comparamos con el hecho de conseguir la misma funcionalidad incrustando SQL en un lenguaje de programación.

# 22.4 Características objeto-relacional de Oracle 8

En esta sección repasaremos algunas de las características relacionadas con la versión del producto DBMS de Oracle denominada Release 8.X, que han mejorado con la incorporación de características objeto-relacional. Es posible que en versiones posteriores de Oracle se vayan añadiendo más características. Se han añadido varios tipos de datos relacionados con lo que se conoce como cartuchos (*cartridges*).[12] Por ejemplo, el cartucho espacial permite la manipulación de información geográfica o basada en mapas. La manipulación de datos multimedia también es posible gracias a los nuevos tipos de datos. Aquí destacamos las diferencias entre la versión 8.X de Oracle (que era la versión disponible en el momento de escribir este libro) y la versión anterior en cuanto a las características de orientación a objetos y a los tipos de datos nuevos, sin olvidar algunas opciones de almacenamiento. Los fragmentos del lenguaje SQL-99, que explicamos en la Sección 22.1, serán aplicables a Oracle. No explicaremos aquí estas características.

## 22.4.1 Algunos ejemplos de características objeto-relacional de Oracle

Como ORDBMS, Oracle 8 continúa ofreciendo las capacidades de un RDBMS y soporte adicional del concepto de orientación a objetos. Esto ofrece unos niveles más altos de abstracción, de modo que los desarrolladores de aplicaciones pueden manipular los objetos de aplicación en oposición a la creación de objetos de datos relacionales. La información compleja sobre un objeto se puede ocultar, pero en el modelo de datos pueden identificarse las propiedades (atributos, relaciones) y los métodos (operaciones) del objeto. Además, las declaraciones del tipo de objeto se pueden reutilizar a través de la herencia, reduciéndose en consecuencia el tiempo y el esfuerzo de desarrollo de la aplicación. Para facilitar el modelado del objeto, Oracle introduce las siguientes características (así como algunas características de SQL-99 que vimos en la Sección 22.1).

**Representación de atributos multivalor utilizando VARRAY.** Algunos atributos de un objeto/entidad podrían ser multivalor. En el modelo relacional, los atributos multivalor tendrían que manipularse creando una tabla nueva (consulte las Secciones 7.1 y 10.3.2 de la primera forma normal). Si en una tabla grande tuviéramos diez atributos multivalor, tras la normalización tendríamos once tablas, generadas a partir de una sola tabla. Para recuperar los datos, el desarrollador tendría que realizar diez concatenaciones entre estas tablas. Esto no ocurre en un modelo de objetos porque todos los atributos de un objeto (incluyendo los que son multivalor) están encapsulados dentro del objeto. Oracle 8 lo consigue mediante el tipo de datos de array de longitud variable (VARRAY), que tiene las siguientes propiedades:

1. **COUNT**. Número actual de elementos.
2. **LIMIT**. Número máximo de elementos que el VARRAY puede contener; es una cantidad que define el usuario.

Vamos a ver el ejemplo de una entidad VARRAY denominada CLIENTE que tiene los atributos Nombre y Numeros_telefono, donde Numeros_telefono es multivalor. En primer lugar, necesitamos definir un tipo de objeto que represente a un Numero_telefono , y lo hacemos de este modo:

```
CREATE TYPE TIPO_NUM_TLF AS OBJECT (Numero_telefono CHAR(10) );
```

Después, definimos un VARRAY cuyos elementos serían los objetos de tipo TIPO_NUM_TLF:

```
CREATE TYPE TIPO_LISTA_TLF AS VARRAY (5) OF TIPO_NUM_TLF;
```

Ahora podemos crear el tipo de datos TIPO_CLIENTE como un objeto con los atributos Nombre_cliente y Numeros_telefono:

[12] Los cartuchos de Oracle son algo parecido a los Data Blades de Informix.

```
CREATE TYPE TIPO_CLIENTE AS
OBJECT (    Nombre_cliente VARCHAR(20),
            Numeros_telefono TIPO_LISTA_TLF );
```

Ahora es posible crear la tabla cliente de este modo:

```
CREATE TABLE CLIENTE OF TIPO_CLIENTE;
```

Para recuperar una lista de todos los clientes y sus números de teléfono, podemos emitir una consulta sencilla sin concatenaciones:

```
SELECT      Nombre_cliente, Numeros_telefono
FROM        CLIENTES;
```

**Uso de tablas anidadas para representar objetos complejos.** En el modelado de objetos, algunos atributos de un objeto pueden ser objetos a su vez. Oracle 8 acomete esto mediante **tablas anidadas** (consulte la Sección 20.6). Aquí, las columnas (que son equivalentes a los atributos de un objeto) se pueden declarar como tablas. En el ejemplo anterior asumíamos que teníamos una descripción asociada a cada número de teléfono (por ejemplo: casa, oficina y móvil). Esto puede modelarse con una tabla anidada redefiniendo en primer lugar TIPO_NUM_TLF de este modo:

```
CREATE TYPE TIPO_NUM_TLF AS
OBJECT ( Numero_telefono CHAR(10), Descripción CHAR(30) );
```

Después redefinimos TIPO_LISTA_TLF como una tabla de TIPO_NÚMERO_TLF de este otro modo:

```
CREATE TYPE TIPO_LISTA_TLF AS TABLE OF TIPO_NÚMERO_TLF;
```

Después podemos crear el tipo tipo_cliente y la tabla cliente como antes. La única diferencia es que TIPO_LISTA_TLF es ahora una tabla anidada en lugar de un VARRAY. Las dos estructuras tienen funciones parecidas con unas cuantas diferencias. Las tablas anidadas no tienen un límite superior en cuanto al número de elementos, mientras que los VARRAY tienen un límite. De las tablas anidadas es posible recuperar elementos individuales, algo que no es posible con los VARRAY. En las tablas anidadas también pueden crearse índices adicionales para acceder más rápidamente a los datos.

**Vistas de objetos.** Las vistas de objetos se pueden utilizar para crear objetos virtuales a partir de los datos relacionales, de modo que los programadores pueden hacer evolucionar los esquemas existentes para que soporten los objetos. Así, en la misma base de datos pueden coexistir aplicaciones relacionales y de objetos. En nuestro ejemplo, habíamos modelado la base de datos de nuestro cliente utilizando un modelo relacional, pero la dirección decidió que todas las aplicaciones futuras fueran en el modelo de objeto. Moverse por la vista de objeto de los mismos datos relacionales facilitaría la transición.

### 22.4.2 Administración de objetos grandes y otras características de almacenamiento

Oracle puede almacenar ahora objetos extremadamente grandes, como vídeo, audio y documentos de texto. Con este propósito se han introducido nuevos tipos de datos, como los siguientes:

- BLOB (objeto binario grande).
- CLOB (objeto grande de caracteres).
- BFILE (fichero binario almacenado fuera de la base de datos).
- NCLOB (CLOB multibyte de anchura fija).

Todos los tipos anteriores, excepto BFILE que se almacena fuera de la base de datos, se almacenan dentro de la base de datos, junto con otros datos. Sólo el nombre de directorio de un BFILE se almacena en la base de datos.

***Index Only Tables.*** El estándar Oracle 7.X implica el mantenimiento de índices como un árbol $B^+$ que contienen punteros a los bloques de datos (consulte el Capítulo 14). En la mayoría de las situaciones esto ofrece un buen rendimiento. Sin embargo, debe accederse tanto al índice como al bloque de datos para leer los datos. Además, los valores clave se almacenan dos veces (en la tabla y en el índice), lo que aumenta el coste de almacenamiento. Oracle 8 y posteriores soportan el esquema de indexación estándar e ***index only tables***, donde los registros de datos y los índices se mantienen juntos en una estructura de árbol B (consulte el Capítulo 14). Esto permite una recuperación más rápida de los datos y requiere menos espacio de almacenamiento para ficheros de pequeño y medio tamaño, donde el tamaño del registro no es demasiado grande.

**Tablas e índices particionados.** Las tablas grandes y los índices se pueden dividir en particiones más pequeñas. La tabla se convierte ahora en una estructura lógica y las particiones en las estructuras físicas reales que albergan los datos. Esto tiene las siguientes ventajas:

- Disponibilidad continuada de los datos en el caso de fallos parciales de algunas particiones.
- Rendimiento escalable que permite un crecimiento sustancial de los volúmenes de datos.
- Mejora del rendimiento global del procesamiento de consultas y transacciones.

## 22.5 Implementación y problemas relacionados con los sistemas de tipos extendidos

Hay varios problemas de implementación relacionados con el soporte de un sistema de tipos extendido con funciones (operaciones) asociadas. Vamos a resumirlo brevemente.[13]

- El ORDBMS debe enlazar dinámicamente una función definida por el usuario en su espacio de direcciones sólo cuando es requerido. Como vimos en el caso de los ORDBMSs, se necesitan muchas funciones para operar sobre datos espaciales de dos o tres dimensiones, imágenes, texto, etcétera. Con una vinculación estática de todas las librerías de funciones, el espacio de direcciones del DBMS puede crecer según un orden de magnitud. La vinculación dinámica está disponible en los dos ORDBMSs que estudiamos.
- Los problemas cliente-servidor tienen que ver con la colocación y la activación de funciones. Si el servidor tiene que ejecutar una función, es mejor hacerlo en el espacio de direcciones del DBMS que remotamente, debido a la gran cantidad de sobrecarga. Si la función demanda cálculos intensivos o si el servidor está atendiendo a un gran número de clientes, el servidor puede enviar la función a una máquina cliente separada. Por razones de seguridad, es mejor ejecutar las funciones en el cliente utilizando el ID de usuario del cliente. En el futuro, las funciones probablemente se escribirán en lenguajes interpretados, como Java.
- Debe ser posible ejecutar consultas dentro de funciones. Una función debe operar de la misma forma si se utiliza desde una aplicación utilizando la interfaz de programación de aplicaciones (API), como si es llamada por el DBMS como parte de la ejecución de SQL con la función incrustada en una sentencia SQL. Los sistemas deben soportar una anidación de esas retrollamadas (*callbacks*).
- Debido a la variedad de tipos de datos en un ORDBMS y los operadores asociados, es importante que el almacenamiento y el acceso a los datos sean eficaces. Para los datos espaciales o datos multidimen-

[13] Esta explicación la hemos obtenido principalmente de Stonebraker y Moore (1996).

sionales, pueden utilizarse nuevas estructuras de almacenamiento, como los árboles R, los árboles cuadrados o los ficheros rejilla. El ORDBMS debe permitir la definición de tipos nuevos con estructuras de acceso nuevas. El tratamiento de grandes cadenas de texto o ficheros binarios también abre muchas posibilidades de almacenamiento y búsqueda. Debe ser posible explorar dichas opciones mediante la definición de tipos de datos nuevos dentro del ORDBMS.

**Otros problemas relacionados con los sistemas objeto-relacional.** En la explicación anterior sobre Informix Universal Server y Oracle 8, nos hemos centrado en cómo un ORDBMS extiende el modelo relacional. Explicamos las características y las capacidades que proporcionan para operar sobre los datos relacionales almacenados como tablas como si fuera una base de datos de objetos. Debemos considerar otros problemas obvios en el contexto de un ORDBMS:

- **Diseño de la base de datos objeto-relacional.** En la Sección 21.5 describimos un procedimiento para diseñar esquemas de objetos. El diseño objeto-relacional es más complejo porque no sólo tenemos que considerar las estimaciones de diseño subyacente de la semántica de aplicación y las dependencias en el modelo de datos relacional (que explicamos en los Capítulos 10 y 11), sino también la naturaleza de orientación a objetos de las características extendidas que acabamos de explicar.
- **Procesamiento y optimización de consultas.** Mediante la ampliación de SQL con funciones y reglas, este problema queda más allá de la panorámica de optimización de consultas que explicamos para el modelo relacional en el Capítulo 15.
- **Interacción de reglas con transacciones.** El procesamiento de reglas como implícito en SQL abarca más que únicamente las reglas actualizar-actualizar (consulte la Sección 24.1), que están implementadas en los RDBMSs como *triggers*. Además, actualmente los RDBMSs sólo implementan la ejecución inmediata de *triggers*. Una ejecución diferida de *triggers* implica un procesamiento adicional.

## 22.6 El modelo relacional anidado

Para completar esta explicación, en esta sección resumimos un método que propone el uso de tablas anidadas, también conocido como relaciones de formas no normales. Ningún DBMS comercial ha elegido implementar este concepto en su forma original. El **modelo relacional anidado** elimina la restricción de la primera forma normal (1FN; consulte el Capítulo 11) del modelo relacional básico, y por tanto también se conoce como **No-1FN** o Non-First Normal Form (NFNF) o modelo relacional **FN**$^2$. En el modelo relacional básico (también conocido como **modelo relacional plano**), es necesario que los atributos sean monovalor y que tengan dominios atómicos. El modelo relacional anidado permite atributos compuestos y multivalor, lo que lleva a tuplas complejas con una estructura jerárquica. Esto resulta de utilidad para representar objetos que están estructurados jerárquicamente de una forma natural. La Figura 22.1(a) muestra un esquema relacional anidado, DEPT, basado en parte de la base de datos EMPRESA y la Figura 22.1(b) ofrece un ejemplo de una tupla No-1FN de DEPT.

Para definir el esquema DEPT como una estructura anidada, podemos escribir lo siguiente:

```
DEPT = (Dno, NombreDpto, Director, EMPLEADOS, PROYECTOS, UBICACIONES)
EMPLEADOS = (NombreE, SUBORDINADOS)
PROYECTOS = (NombreProyecto, UbicaciónProyecto)
UBICACIONES = (UbicaciónDpto)
SUBORDINADOS = (NombreDpto, Edad)
```

En primer lugar, definimos todos los atributos de la relación DEPT. A continuación, definimos los atributos anidados de DEPT (EMPLEADOS, PROYECTOS y UBICACIONES). Después, definimos los atributos anidados de segundo nivel, como SUBORDINADOS de EMPLEADOS, etcétera. Los nombres de todos los atribu-

**Figura 22.1.** Ilustración de una relación anidada. (a) Esquema de DEPT. (b) Ejemplo de una tupla No-1FN de DEPT. (c) Representación en árbol del esquema de DEPT.

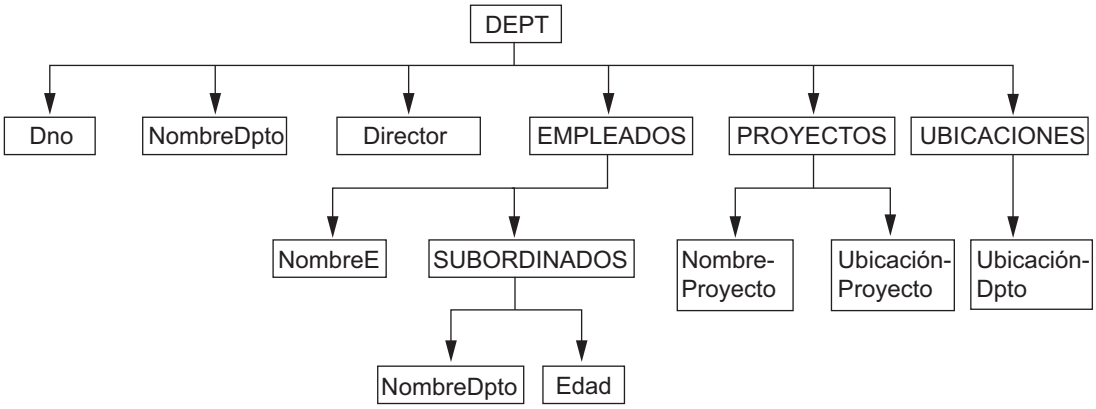
**(a)**

| Dno | NombreDpto | Director | EMPLEADOS | | | PROYECTOS | | UBICACIONES |
|---|---|---|---|---|---|---|---|---|
| | | | NombreE | SUBORDINADOS | | Nombre-Proyecto | Ubicación-Proyecto | UbicaciónDpto |
| | | | | Nombre-Dpto | Edad | | | |

**(b)**

| 4 | Administración | Oreja | Celaya | Tomás | 8 | Beneficios nuevos | Gijón | Gijón |
|---|---|---|---|---|---|---|---|---|
| | | | | Teresa | 6 | Informatización | Gijón | Toledo |
| | | | Oreja | Juan | 18 | Sistema telefónico | Toledo | |
| | | | | Roberto | 15 | | | |
| | | | | María | 10 | | | |
| | | | Morera | | | | | |

**(c)**




tos deben ser distintos en la definición de la relación anidada. Un atributo anidado es normalmente un **atributo compuesto multivalor**, lo que nos lleva a una "relación anidada" *dentro de cada tupla*. Por ejemplo, el valor del atributo PROYECTOS dentro de cada tupla de DEPT es una relación con dos atributos (NombreProyecto, UbicaciónProyecto). En la tupla DEPT de la Figura 22.1(b), el atributo PROYECTOS contiene tres tuplas como su valor. Otros atributos anidados pueden ser **atributos simples multivalor**, como UBICACIONES de DEPT. También es posible tener un atributo anidado que sea **monovalor y compuesto**, aunque la mayoría de los modelos relacionales anidados tratan a un atributo semejante como si fuera multivalor.

Cuando se define un esquema de bases de datos relacional, consta de varios esquemas de relación externos, que definen el nivel superior de las relaciones anidadas individuales. Además, los atributos anidados se denominan **esquemas de relación internos**, puesto que definen las estructuras relacionales que se anidan dentro de otra relación. En nuestro ejemplo, DEPT es la única relación externa. Todas las demás (EMPLEADOS, PROYECTOS, UBICACIONES y SUBORDINADOS) son relaciones internas. Por último, los **atributos simples** aparecen en el nivel hoja y no están anidados. Cada esquema de relación puede representarse mediante una estructura en árbol, como se muestra en la Figura 22.1(c), donde la raíz es un esquema de relación externa, las hojas son atributos simples y los nodos internos son esquemas de relación interiores. Observe la similitud entre esta representación y un esquema jerárquico (consulte el Apéndice D) y XML (consulte el Capítulo 27).

Es importante ser consciente de que las tres relaciones anidadas de primer nivel de DEPT representan *información independiente*. Por tanto, EMPLEADOS representa a los empleados *que trabajan para* el departamen-

to, PROYECTOS representa los proyectos *controlados por* el departamento, y UBICACIONES representa la ubicación de los distintos departamentos. La relación entre EMPLEADOS y PROYECTOS no está representada en el esquema; es una relación M:N, que es difícil de representar en una estructura jerárquica.

Se han propuesto extensiones al álgebra relacional y al cálculo relacional, así como a SQL, para el tema de las relaciones anidadas. El lector interesado puede consultar la bibliografía seleccionada que aparece al final de este capítulo. Aquí, ilustramos dos operaciones, **NEST** y **UNNEST**, que se pueden utilizar para ampliar las operaciones estándar del álgebra relacional relativas a la conversión entre relaciones anidadas y planas. Consideremos la relación EMP_PROY plana de la Figura 10.3(b), y supongamos que la proyectamos sobre los atributos Dni, NumProyecto, Horas, NombreE de este modo:

$$\text{PROY_EMP_PLANO} \leftarrow \pi_{\text{Dni, NombreE, NumProyecto, Horas}}(\text{EMP_PROY})$$

Para crear una versión anidada de esta relación, donde existe una tupla por cada empleado y (NumProyecto, Horas) está anidado, utilizamos la operación NEST de este modo:

PROY_EMP_ANIDADO.PROYANIDADOS = (NumProyecto, Horas)(PROY_EMP_PLANO)

El efecto de esta operación es crear una relación anidada interna PROYS = (NumProyecto, Horas) dentro de la relación externa PROY_EMP_ANIDADO. Por tanto, NEST agrupa las tuplas *con el mismo valor* para los atributos *que no se especifican* en la operación NEST; en nuestro ejemplo, son los atributos Dni y NombreE. Para cada grupo de estas características, que en nuestro ejemplo representa a un empleado, se crea una tupla anidada simple con una relación anidada interna PROYS = (NumProyecto, Horas). Por tanto, la relación PROY_EMP_ANIDADO se parece a la relación EMP_PROY de las Figuras 11.9(a) y (b).

Observe la similitud entre anidamiento y agrupamiento para las funciones agregadas. En el primero, cada grupo de tuplas se convierte en una tupla anidada simple; en el último, cada grupo se convierte en una tupla resumen simple después de haberse aplicado una función de agregación al grupo.

La operación UNNEST es la inversa de la operación NEST. Podemos reconvertir PROY_EMP_ANIDADO en PROY_EMP_PLANO de este modo:

PROY_EMP_PLANO.DESANIDARPROYS = (NumProyecto, Horas)(PROY_EMP_ANIDADO)

Aquí, el atributo anidado PROYS se aplana en sus componentes NumProyecto, Horas.

# 22.7 Resumen

En este capítulo ofrecimos primero una visión general de las características de orientación a objetos de SQL-99, que se pueden aplicar a los sistemas objeto-relacional. Después explicamos la historia y las tendencias actuales en lo que se refiere a los sistemas de administración de bases de datos que conducen al desarrollo de DBMSs objeto-relacional (ORDBMSs). Nos hemos centrado en algunas de las características de Informix Universal Server y de Oracle 8 con el objetivo de ilustrar cómo se están ampliando los RDBMSs comerciales con las características de objetos. Aunque las versiones actuales de Informix y Oracle tienen características adicionales, el objetivo de este capítulo ha sido ilustrar las capacidades básicas que permiten presentar los datos relacionales al usuario en un modelo basado en objetos y en un lenguaje de consulta.

Otros RDBMSs comerciales proporcionan extensiones parecidas. Vimos que estos sistemas, como Data Blades (Informix) o Cartridges (Oracle), proporcionan extensiones de tipos específicos para los dominios de aplicación más novedosos, como el espacial, las series temporales o las bases de datos de texto/documentos. Debido a la extensibilidad de los ORDBMSs, estos paquetes pueden incluirse como librerías de tipos de datos abstractos (ADT) siempre que los usuarios tengan la necesidad de implementar los tipos de aplicaciones que soportan. Los usuarios también implementan sus propias extensiones según sus necesidades, utilizando los servicios ADT de estos sistemas. Hemos explicado brevemente algunos problemas de implementación para

los ADTs. Por último, ofrecimos una visión general del modelo relacional anidado, que extiende el modelo relacional plano con objetos complejos jerárquicamente estructurados.

## Bibliografía seleccionada

Las referencias proporcionadas para las bases de datos orientadas a objetos en los Capítulos 20 y 21 también se pueden aplicar a los sistemas objeto-relacional. Stonebraker y Moore (1996) ofrece una referencia global para los DBMSs objeto-relacional. La explicación de los conceptos relacionados con Illustra que aparecen en este libro se aplican principalmente a Informix Universal Server en su versión actual. Kim (1995) explica muchos temas relacionados con los sistemas de bases de datos modernos que incluyen la orientación a objetos. Si desea información más reciente sobre Informix y Oracle, consulte sus sitios web respectivos: *http://www.informix.com y http://www.oracle.com.*

El estándar SQL3 se describe en varias publicaciones del ISO WG3 (*Working Group 3*); por ejemplo, consulte Kulkarni y otros (1995) y Melton y otros (1991). En la Very Large Data Bases Conference por Melton y Mattos (1996) se ofreció un excelente tutorial sobre SQL3. Ullman y Widom (1997) ofrece una buena explicación de SQL3 con ejemplos.

Si lo que busca son temas relacionados con las reglas y los *triggers*, Widom y Ceri (1995) ofrece una colección de capítulos sobre bases de datos activas. Algunos estudios comparativos [por ejemplo, Ketabchi y otros (1990)] comparan los DBMSs relacionales con los DBMSs de objetos; su conclusión muestra la superioridad del método de orientación a objetos para las aplicaciones no convencionales. El modelo relacional anidado se explica en Schek y Scholl (1985), Jaeshke y Schek (1982), Chen y Kambayashi (1991), y Makinouchi (1977), entre otros. El álgebra y los lenguajes de consulta para las relaciones anidadas se presentan en Paredaens y VanGucht (1992), Pistor y Andersen (1986), Roth y otros (1988), y Ozsoyoglu y otros (1987), entre otras referencias. La implementación de sistemas relacionales anidados prototipo se describe en Dadam y otros (1986), Deshpande y VanGucht (1988), y Schek y Scholl (1989).

# PARTE 7

# Temas avanzados: seguridad, modelación avanzada y distribución

CAPÍTULO 23

# Seguridad en las bases de datos

En este capítulo se revisan las técnicas para proteger las bases de datos contra diferentes amenazas. También se presentan métodos para proporcionar privilegios de acceso a los usuarios autorizados. La Sección 23.1 muestra una introducción a los temas de seguridad y a las amenazas a las bases de datos y ofrece una revisión de las medidas de control que se tratan en el resto de este capítulo. En la Sección 23.2 se revisan los mecanismos utilizados para conceder y retirar privilegios en los sistemas de bases de datos relacionales y en SQL, mecanismos se conocen a veces como **control de acceso discrecional**. La Sección 23.3 ofrece un repaso a los mecanismos utilizados para reforzar diversos niveles de seguridad, una preocupación reciente en la seguridad de las bases de datos que se conoce como **control de acceso obligatorio**. También presenta la estrategia desarrollada recientemente para el **control de acceso basado en roles** y revisa brevemente el control de acceso XML. La Sección 23.4 hace un breve repaso al problema de la seguridad en las bases de datos estadísticas. La Sección 23.5 presenta los desafíos actuales en materia de seguridad en las bases de datos. La Sección 23.9 hace un resumen del capítulo. Los lectores que estén interesados sólo en mecanismos básicos de seguridad en las bases de datos tendrán suficiente con revisar el contenido de las Secciones 23.1 y 23.2.

## 23.1 Introducción a los temas de seguridad en las bases de datos

### 23.1.1 Tipos de seguridad

La seguridad en las bases de datos es un tema muy amplio que comprende muchos conceptos, entre los cuales se incluyen los siguientes:

- Aspectos legales y éticos en relación con el derecho de acceso a determinada información. Ciertos tipos de información están considerados como privados y no pueden ser accedidos legalmente por personas no autorizadas. En los Estados Unidos existen muchas leyes que regulan la privacidad de la información.
- Temas de políticas a nivel gubernamental, institucional o de empresa en relación con los tipos de información que no debería estar disponible públicamente (por ejemplo, la concesión de créditos o los informes médicos personales).

- Temas relativos al sistema, como los *niveles del sistema* en los que se deberían reforzar las distintas funciones de seguridad (por ejemplo, si una función de seguridad debería ser manipulada a nivel físico del hardware, a nivel del sistema operativo o a nivel del DBMS).
- La necesidad dentro de algunas organizaciones de identificar diferentes niveles de seguridad y de clasificar según éstos a los datos y a los usuarios: por ejemplo, alto secreto, secreto, confidencial y no clasificado. Se debe reforzar la política de seguridad de la organización en lo que respecta a las diferentes clasificaciones de los datos.

**Amenazas a las bases de datos.** Las amenazas a las bases de datos tienen como consecuencia la pérdida o la degradación de todos o de algunos de los siguientes objetivos de seguridad comúnmente aceptados: integridad, disponibilidad y confidencialidad.

- **Pérdida de integridad.** La integridad de la base de datos tiene relación con el requisito a cumplir de que la información se encuentre protegida frente a modificaciones inadecuadas. La modificación de datos incluye la creación, inserción, modificación, cambio del estado de los datos y el borrado. La integridad se pierde si se realizan cambios no autorizados en los datos mediante acciones intencionadas o accidentales. Si no se corrige la pérdida de integridad del sistema o de los datos, el uso continuado de un sistema contaminado o de datos corrompidos podría tener como consecuencia la toma de decisiones inexactas, fraudulentas o erróneas.
- **Pérdida de disponibilidad.** La disponibilidad de la base de datos tiene relación con que los objetos estén disponibles para un usuario humano o para un programa que tenga los derechos correspondientes.
- **Pérdida de confidencialidad.** La confidencialidad de la base de datos tiene relación con la protección de los datos frente al acceso no autorizado. El impacto del acceso no autorizado a la información confidencial puede variar desde la violación de las leyes sobre privacidad de los datos hasta la amenaza a la seguridad nacional. El acceso no autorizado, no previsto o no intencionado podría tener como resultado la pérdida de la confianza en la organización, el que ésta quede en entredicho o que sea objeto de acciones legales en su contra.

Para proteger las bases de datos contra estos tipos de amenazas, es habitual implementar cuatro tipos de medidas de control: control de accesos, control de inferencias, control de flujo y cifrado. En este capítulo revisaremos cada una de ellas.

En un sistema de base de datos multiusuario, el DBMS debe proporcionar técnicas que permitan a determinados usuarios o grupos de usuarios el acceso a partes concretas de una base de datos sin tener acceso al resto de esta última. Esto resulta muy importante cuando una base de datos integrada de gran tamaño va a ser utilizada por muchos usuarios distintos dentro de la misma organización. Por ejemplo, la información confidencial (por ejemplo, los salarios de los empleados o sus evaluaciones de rendimiento) deberían permanecer como confidenciales para la mayoría de los usuarios del sistema de base de datos. Un DBMS incluye, por lo general, un **subsistema de autorizaciones y seguridad en bases de datos** responsable de garantizar la seguridad de partes de una base de datos frente a accesos no autorizados. En este momento, resulta obligado indicar dos tipos de mecanismos de seguridad en bases de datos:

- **Mecanismos de seguridad discrecionales.** Se utilizan para conceder permisos a usuarios, incluyendo la capacidad de acceso a determinados archivos de datos, registros o campos en un modo en concreto (lectura, inserción, borrado o actualización).
- **Mecanismos de seguridad obligatorios.** Se utilizan para reforzar la seguridad a varios niveles mediante la clasificación de los datos y de los usuarios en varias clases de seguridad (o niveles) para después implementar la política de seguridad adecuada a la organización. Por ejemplo, una política de seguridad habitual es permitir a los usuarios de un determinado nivel de clasificación ver sólo los elementos de datos clasificados en el mismo (o menor) nivel de clasificación que el del usuario. Una

extensión a esto es la *seguridad basada en roles*, en la cual se refuerzan las políticas y permisos basándose en el concepto de roles.

Veremos la seguridad discrecional en la Sección 23.2 y la seguridad obligatoria y basada en roles en la Sección 23.3.

### 23.1.2 Medidas de control

Existen cuatro medidas de control principales que se utilizan para proporcionar seguridad a los datos en las bases de datos. Son las siguientes:

- Control de accesos.
- Control de inferencias.
- Control de flujo.
- Cifrado de datos.

Un problema de seguridad habitual en los sistemas de computadores es prevenir que personas no autorizadas tengan acceso al sistema, bien para obtener información o bien para realizar cambios malintencionados en partes de la base de datos. El mecanismo de seguridad de un DBMS debe incluir medidas para restringir el acceso al sistema de base de datos en su totalidad. Esta función se denomina **control de acceso** y se gestiona mediante la creación de cuentas de usuario y contraseñas para controlar el proceso de entrada al DBMS. Revisaremos las técnicas de control de acceso en la Sección 23.1.3.

Las **bases de datos estadísticas** se utilizan para proporcionar información estadística o resúmenes de valores basados en diversos criterios. Por ejemplo, una base de datos con información estadística de población puede proporcionar datos estadísticos basados en los grupos de edad, niveles de ingresos, dotación de los hogares, niveles de educación y otros criterios. Los usuarios de bases de datos estadísticas, como los institutos de estadística de los gobiernos o las compañías de investigación de mercados, tienen privilegios para acceder a la base de datos para obtener información estadística sobre un determinado grupo de población, pero no para acceder a la información confidencial detallada sobre individuos en particular. La seguridad en las bases de datos estadísticas debe garantizar que la información relativa a los individuos no pueda ser accesible. A veces es posible deducir ciertos datos acerca de los individuos a partir de las consultas relativas únicamente a valores resumen sobre grupos; esto tampoco se debe permitir. Este problema, llamado **seguridad en bases de datos estadísticas**, se revisa brevemente en la Sección 23.4. Las medidas de control correspondientes se denominan **medidas de control de inferencias**.

Otro tema en relación con la seguridad es el control de flujo, que previene que la información fluya de tal manera que llegue a usuarios no autorizados. Se verá en la Sección 23.5. Los canales que resultan ser caminos que permiten que la información fluya implícitamente de modo que se viole la política de seguridad de una organización se denominan **canales ocultos**. En la Sección 23.5.1 veremos brevemente algunos temas relacionados con los canales ocultos.

Una última medida de seguridad es el **cifrado de datos**, que se utiliza para proteger datos confidenciales (como los números de las tarjetas de crédito) que sean transmitidos a través de algún tipo de red de comunicación. El cifrado se puede utilizar también para proporcionar protección adicional a partes confidenciales de la base de datos. Los datos se **codifican** utilizando algún algoritmo de codificación o cifrado. Un usuario no autorizado que acceda a datos codificados tendrá dificultades para descifrarlos, pero a los usuarios autorizados se les proporcionarán algoritmos de descodificación o descifrado (claves) para descifrar los datos. En las aplicaciones militares se han desarrollado técnicas de cifrado en las que es muy difícil descifrar sin conocer la clave. En la Sección 23.6 se hace un breve repaso a las técnicas de cifrado, incluyendo técnicas muy populares, como el cifrado con clave pública, de amplia utilización en la red para posibilitar las transacciones sobre bases de datos, y las firmas digitales, que se utilizan en las comunicaciones personales.

Queda fuera del ámbito de este libro una revisión en profundidad de la seguridad en los sistemas de computadores y en las bases de datos. Aquí sólo haremos un breve repaso a las técnicas de seguridad en las bases de datos. Los lectores más interesados en el tema pueden dirigirse a algunas de las referencias mencionadas en la Bibliografía seleccionada que aparece al final de este capítulo.

### 23.1.3 La seguridad en bases de datos y el DBA

Según vimos en el Capítulo1, el administrador de la base de datos (DBA) es el responsable principal de la gestión de un sistema de base de datos. Entre las responsabilidades del DBA se encuentran la concesión y retirada de privilegios a los usuarios que necesitan utilizar el sistema, y la clasificación de usuarios y datos en función de la política de la organización. El DBA dispone de una **cuenta de DBA** en el DBMS, llamada también a veces **cuenta de sistema** o de **superusuario**, que proporciona enormes posibilidades que no están disponibles para las cuentas y los usuarios normales de la base de datos.[1] Entre los comandos privilegiados a nivel del DBA se encuentran los comandos para conceder o retirar privilegios a las cuentas o usuarios individuales o a los grupos de usuarios y para ejecutar los siguientes tipos de acciones:

1. **Creación de cuentas.** Mediante esta acción se crea una nueva cuenta y contraseña para posibilitar el acceso al DBMS a un usuario o grupo de usuarios.
2. **Concesión de privilegios.** Esta acción permite al DBA conceder determinados permisos a determinadas cuentas.
3. **Retirada de privilegios.** Esta acción permite al DBA retirar (cancelar) determinados permisos concedidos previamente a determinadas cuentas.
4. **Asignación del nivel de seguridad.** Esta acción consiste en asignar cuentas de usuario al nivel de clasificación de seguridad adecuado.

El DBA es el responsable de la seguridad general del sistema de bases de datos. La acción 1 de la lista anterior se utiliza para controlar el acceso al DBMS en su totalidad, mientras que las acciones 2 y 3 se utilizan para controlar las autorizaciones *discrecionales* en la base de datos y la acción 4 se utiliza para controlar la autorización *obligatoria*.

### 23.1.4 Protección de accesos, cuentas de usuario y auditorías de bases de datos

Siempre que una persona o grupo de personas necesitan acceder a un sistema de base de datos, ese individuo o grupo deben solicitar, en primer lugar, una cuenta de usuario. El DBA creará entonces una **cuenta de usuario** y una **contraseña** nuevas para el usuario si existe la necesidad legitimada de su acceso a la base de datos. El usuario deberá **iniciar sesión** en el DBMS introduciendo el número de cuenta y la contraseña cada vez que necesite acceso a la base de datos. EL DBMS comprueba que el número de cuenta y la contraseña son válidos; si lo son, se permite que el usuario utilice el DBMS y acceda a la base de datos. También se puede considerar a los programas de aplicación como usuarios y se les podrá solicitar la contraseña.

Resulta sencillo seguir la pista a los usuarios de la base de datos y a sus cuentas y contraseñas mediante la creación de una tabla cifrada o archivo con dos campos: Cuenta de usuario y Contraseña. El DBMS puede realizar el mantenimiento de esta tabla de una forma muy sencilla. Cada vez que se cree una nueva cuenta, se insertará un registro en la tabla. Cuando una cuenta sea cancelada, el registro correspondiente debe ser borrado de la tabla.

---

[1] Esta cuenta es similar a las cuentas de tipo *root (raíz)* o *superuser* (superusuario) que pertenecen a los administradores de sistemas y permite el acceso a comandos restringidos del sistema operativo.

El sistema de base de datos también debe seguir el rastro de todas las operaciones en la base de datos que un usuario determinado realice durante cada **sesión**, que comprende toda la secuencia de interacciones realizadas por ese usuario en la base de datos desde el momento en que inicia la sesión hasta el momento en que finaliza. Cuando un usuario inicia una sesión, el DBMS puede registrar el número de cuenta del usuario y asociarlo al terminal desde el cual el usuario abrió la sesión. Todas las operaciones desde ese terminal se asignarán a la cuenta del usuario hasta que éste cierre la sesión. Es particularmente importante seguir el rastro a las operaciones de actualización que se realicen sobre la base de datos, de modo que el DBA pueda determinar qué usuario provocó el daño en el caso de que la base de datos sea modificada de forma malintencionada.

Para mantener un registro de todas las actualizaciones realizadas en la base de datos y de los usuarios en particular que realizaron cada actualización, podemos modificar el registro de sucesos del sistema (*system log*). Recuerde de los Capítulos 17 y 19 que el **registro (de sucesos) del sistema** incluye una entrada por cada operación realizada sobre la base de datos que se pueda necesitar en caso de fallo transaccional o de caída del sistema. Es posible expandir las entradas del registro de forma que incluyan también en el registro el número de cuenta del usuario y el identificador del terminal desde el que se realizó la operación. Si se sospecha de alguna modificación malintencionada, se realizará una **auditoría de la base de datos**, que consiste en la revisión del archivo de registro para examinar todos los accesos y operaciones realizadas sobre la base de datos durante un determinado período de tiempo. Cuando se encuentre una operación ilegal o no autorizada, el DBA podrá determinar el número de cuenta utilizado para realizar la operación. Las auditorías en la base de datos son particularmente importantes en el caso de bases de datos con información confidencial que sean actualizadas por un gran número de transacciones y usuarios, como una base de datos bancaria actualizada por un gran número de cajeros del banco. A un registro de una base de datos que se utilice principalmente para propósitos de seguridad se le llama, a veces, **registro de auditoría**.

# 23.2 Control de acceso discrecional basado en la concesión y revocación de privilegios

El método habitual para reforzar el **control de acceso discrecional** en un sistema de base de datos se basa en la concesión y revocación de **privilegios**. Vamos a considerar los privilegios en el contexto de un DBMS relacional. En particular, veremos un sistema de privilegios similar en cierto modo al desarrollado originalmente para el lenguaje SQL (consulte el Capítulo 8). Muchos DBMSs relacionales de la actualidad utilizan alguna variación de esta técnica. La idea principal es incluir declaraciones en el lenguaje de consultas que permitan al DBA y a los usuarios seleccionados conceder y revocar privilegios.

## 23.2.1 Tipos de privilegios discrecionales

En SQL2, el concepto de **identificador de autorización** se utiliza para referirse, a grandes rasgos, a una cuenta de usuario (o a un grupo de cuentas de usuario). Por simplicidad, utilizaremos las palabras *usuario* o *cuenta* de modo indiferente en lugar de *identificador de autorización*. El DBMS debe proporcionar acceso selectivo a cada relación de la base de datos basándose en cuentas determinadas. También podemos controlar las operaciones; de este modo, la posesión de una cuenta no significa necesariamente que el propietario de la cuenta esté autorizado a toda la funcionalidad que permita el DBMS. Informalmente hablando, existen dos niveles de asignación de privilegios de uso del sistema de base de datos:

- **El nivel de cuenta.** En este nivel, el DBA especifica los privilegios en particular que posee cada cuenta, independientemente de las relaciones existentes en la base de datos.
- **El nivel de relación (o tabla).** En este nivel, el DBA puede controlar el privilegio de acceso a cada relación o vista individual en la base de datos.

Los privilegios a **nivel de cuenta** se aplican a las funcionalidades de la propia cuenta y pueden incluir el privilegio sobre CREATE SCHEMA o CREATE TABLE para crear un esquema o relación base; el privilegio sobre CREATE VIEW; el privilegio sobre ALTER, para realizar cambios en el esquema como la agregación o eliminación de atributos en las relaciones; el privilegio sobre DROP, para borrar relaciones o vistas; el privilegio sobre MODIFY, para insertar, borrar o actualizar tuplas; y el privilegio sobre SELECT, para obtener información de la base de datos usando una consulta SELECT. Observe que estos privilegios de cuenta se aplican a las cuentas en general. Si una cuenta determinada no tiene el privilegio sobre CREATE TABLE, no se podrán crear relaciones desde esa cuenta. Los privilegios a nivel de cuenta *no forman* parte de SQL2; deberán ser definidos por los creadores del DBMS. En versiones anteriores de SQL, existía un privilegio denominado CREATETAB para proporcionar a una cuenta el privilegio para crear tablas (relaciones).

El segundo nivel de privilegios se aplica al **nivel de relación**, bien sean relaciones base o relaciones virtuales (vistas). Estos privilegios *se definen* en SQL2. En lo que veremos a continuación, el término *relación* puede referirse a una relación base o a una vista, a menos que especifiquemos explícitamente una o la otra. Los privilegios a nivel de relación especifican, a nivel de usuario, las relaciones individuales sobre las que se puede aplicar cada tipo de comando. Algunos privilegios también hacen referencia a columnas individuales (atributos) de las relaciones. Los comandos de SQL2 proporcionan privilegios *sólo a nivel de relación y atributos* y, aunque sea un tratamiento bastante general, esto dificulta la creación de cuentas con privilegios limitados. La concesión y revocación de privilegios sigue, por lo general, un modelo de autorizaciones para los privilegios discrecionales conocido como modelo **de matriz de acceso**, en el que las filas de una matriz $M$ representan los sujetos (usuarios, cuentas, programas) y las columnas representan objetos (relaciones, registros, columnas, vistas, operaciones). Cada posición $M(i, j)$ de la matriz representa los tipos de permisos (lectura, escritura, actualización) que posee el sujeto $i$ sobre el objeto $j$.

Para controlar la concesión y revocación de privilegios de una relación, a cada relación $R$ de una base de datos se le asigna una **cuenta de propietario** que es, por lo general, la cuenta que se utilizó cuando se creó la relación por primera vez. Al propietario de una relación se le conceden *todos* los privilegios sobre esa relación. En SQL2, el DBA puede asignar un propietario a todo el esquema en su totalidad creando el esquema y asociando a ese esquema el identificador de autorización adecuado mediante el comando CREATE SCHEMA (consulte la Sección 8.1.1). El poseedor de la cuenta de propietario puede traspasar los privilegios sobre cualquiera de las relaciones que posee a otros usuarios mediante la **concesión** de privilegios a sus cuentas. En SQL se pueden conceder los siguientes tipos de privilegios a cada relación individual $R$:

- **Privilegio de selección (recuperación o lectura) sobre *R*.** Concede a la cuenta el privilegio para obtener información. En SQL, esto concede a la cuenta el privilegio de utilizar la instrucción SELECT para obtener tuplas de $R$.
- **Privilegio de modificación sobre *R*.** Concede a la cuenta la capacidad de modificar tuplas de $R$. En SQL, este privilegio se subdivide en privilegios sobre UPDATE, DELETE e INSERT para utilizar el correspondiente comando SQL sobre $R$. Adicionalmente, tanto el privilegio sobre INSERT como sobre UPDATE pueden especificar que sólo ciertos atributos de $R$ pueden ser actualizados por la cuenta.
- **Privilegio de referencia de *R*.** Concede a la cuenta la posibilidad de referenciar la relación $R$ al especificar restricciones de integridad. También se puede restringir este privilegio a determinados atributos de $R$.

Observe que para crear una vista, la cuenta debe tener privilegio de SELECT sobre *todas las relaciones* implicadas en la definición de la vista.

## 23.2.2 Especificación de privilegios mediante vistas

El mecanismo de **vistas** es un importante mecanismo de autorización discrecional por sí mismo. Por ejemplo, si el propietario $A$ de una relación $R$ quiere que otra cuenta $B$ pueda obtener sólo algunos campos de $R$, enton-

ces *A* podría crear una vista *V* de *R* que incluyese sólo esos atributos y después autorizar SELECT sobre *V* a *B*. Lo mismo se aplica a la hora de limitar a *B* para que pueda obtener sólo determinadas tuplas de *R*; se puede crear una vista *V'* definiendo la vista por medio de una consulta que seleccione sólo las tuplas de *R* sobre las que *A* desee permitir el acceso a *B*. Ilustraremos esto mediante el ejemplo que aparece en la Sección 23.2.5.

### 23.2.3 Revocación de privilegios

En algunos casos, es deseable conceder un privilegio a un usuario de manera temporal. Por ejemplo, el propietario de una relación podría querer conceder privilegio a un usuario sobre SELECT para realizar una tarea determinada y, después, revocar ese privilegio una vez que la tarea haya finalizado. Según lo anterior, se necesita un mecanismo para **revocar** privilegios. SQL incluye un comando REVOKE con el propósito de retirar privilegios; en la Sección 23.2.5 veremos cómo se utiliza.

### 23.2.4 Propagación de los privilegios mediante GRANT OPTION

Siempre que el propietario *A* de una relación *R* concede un privilegio a otra cuenta *B*, dicho privilegio se puede conceder a *B con* o *sin* la opción **GRANT OPTION**. Si aparece esta opción, significa que *B* también puede conceder ese privilegio sobre *R* a otras cuentas. Supongamos que *A* concede la opción GRANT OPTION a *B* y que *B* entonces concede privilegio sobre *R* a una tercera cuenta C, también con la opción GRANT OPTION. De este modo, los privilegios sobre *R* se pueden **propagar** a otras cuentas sin el conocimiento del propietario de *R*. Si la cuenta propietaria *A* revoca a continuación el privilegio concedido a *B*, todos los privilegios que *B* propagó basados en ese privilegio deberían ser revocados de forma automática por el sistema.

Es posible que un usuario reciba determinados privilegios de dos o más fuentes. Por ejemplo, A4 podría recibir un determinado privilegio UPDATE *R* de *ambos*, A2 y A3. En este caso, si A2 revoca este privilegio a A4, A4 seguirá disfrutando del privilegio al haber sido concedido también por A3. Si A3 revoca posteriormente el privilegio a A4, A4 lo perderá totalmente. Por tanto, un DBMS que permita la propagación de privilegios debe seguir el rastro del modo en que fueron concedidos los privilegios para que la revocación de privilegios se pueda realizar correcta y convenientemente.

### 23.2.5 Un ejemplo

Supongamos que el DBA crease cuatro cuentas (A1, A2, A3 y A4) y quiere que sólo A1 pueda crear relaciones base; en este caso, el DBA debería ejecutar el siguiente comando GRANT en SQL:

```
GRANT CREATETAB TO A1;
```

El privilegio CREATETAB (crear tabla) concede a la cuenta A1 la posibilidad de crear nuevas tablas en la base de datos (relaciones base) y es, por tanto, un *privilegio de cuenta.* Este privilegio formaba parte de las primeras versiones de SQL, pero ahora se deja su definición a criterio de cada implementación del sistema.

En SQL2 se puede obtener el mismo resultado haciendo que el DBA ejecute el comando CREATE SCHEMA según se muestra a continuación:

```
CREATE SCHEMA EJEMPLO AUTHORIZATION A1;
```

Tras esto, la cuenta de usuario A1 puede crear tablas bajo el esquema de nombre EJEMPLO. Para seguir con nuestro ejemplo, supongamos que A1 crease las dos relaciones base EMPLEADO y DEPARTAMENTO que aparecen en la Figura 23.1; A1 sería entonces el propietario de estas dos relaciones y, por tanto, tendría *todos los privilegios de relación* sobre cada una de ellas.

A continuación, supongamos que la cuenta A1 desea conceder a la cuenta A2 el privilegio para insertar y borrar tuplas en ambas relaciones. Sin embargo, A1 no quiere que A2 pueda propagar estos privilegios a otras cuentas. A1 podría ejecutar el siguiente comando:

**Figura 23.1.** Esquemas para las dos relaciones EMPLEADO y DEPARTAMENTO.

**EMPLEADO**

| Nombre | Dni | FechaNac | Dirección | Sexo | Sueldo | Dno |
|---|---|---|---|---|---|---|

**DEPARTAMENTO**

| NúmeroDpto | NombreDpto | DnDirector |
|---|---|---|

**GRANT** INSERT, DELETE **ON** EMPLEADO, DEPARTAMENTO **TO** A2;

Observe que la cuenta A1 propietaria de una relación posee de forma automática la opción GRANT OPTION, permitiéndole conceder privilegios sobre la relación a otras cuentas. Sin embargo, la cuenta A2 no puede conceder los privilegios INSERT y DELETE sobre las tablas EMPLEADO y DEPARTAMENTO, ya que a A2 no se le concedió la opción GRANT OPTION en el comando anterior.

A continuación, supongamos que A1 desea permitir a la cuenta A3 obtener información de cualquiera de las dos tablas y también poder propagar el privilegio SELECT a otras cuentas. A1 podría ejecutar el siguiente comando:

**GRANT** SELECT **ON** EMPLEADO, DEPARTAMENTO **TO** A3 **WITH GRANT OPTION**;

La cláusula WITH GRANT OPTION significa que A3 puede ahora propagar el privilegio a otras cuentas utilizando GRANT. Por ejemplo, A3 puede conceder el privilegio SELECT sobre la relación EMPLEADO a A4 ejecutando el comando siguiente:

**GRANT** SELECT **ON** EMPLEADO **TO** A4;

Observe que A4 no puede propagar el privilegio SELECT a otras cuentas ya que no se le concedió la opción GRANT OPTION.

Supongamos ahora que A1 decide revocar a A3 el privilegio SELECT sobre la relación EMPLEADO. A1 podría ejecutar este comando:

**REVOKE** SELECT **ON** EMPLEADO **FROM** A3;

El DBMS deberá entonces revocar también de forma automática el privilegio SELECT sobre EMPLEADO a A4, ya que A3 concedió ese privilegio a A4 y A3 ya no dispone de ese privilegio.

A continuación, supongamos que A1 desea devolver a A3 unos permisos limitados de SELECT sobre la relación EMPLEADO y desea permitir a A3 que pueda propagar el privilegio. La limitación consiste en poder obtener sólo los atributos Nombre, FechaNac y Dirección y sólo para las tuplas con el atributo Dno igual a 5. A1 podría entonces crear la siguiente vista:

**CREATE VIEW** A3EMPLEADO **AS**
**SELECT** Nombre, FechaNac, Dirección
**FROM** EMPLEADO
**WHERE** Dno = 5;

Una vez creada la vista, A1 podrá conceder a A3 el permiso para ejecutar SELECT sobre la vista A3EMPLEADO según se muestra a continuación:

**GRANT** SELECT **ON** A3EMPLEADO **TO** A3 **WITH GRANT OPTION;**

Por último, supongamos que A1 desea permitir a A4 que actualice sólo el atributo Sueldo de EMPLEADO. A1 podrá entonces ejecutar el siguiente comando:

**GRANT** UPDATE **ON** EMPLEADO (Sueldo) **TO** A4;

Los privilegios UPDATE o INSERT pueden especificar atributos en particular que puedan ser actualizados o insertados en una relación. Otros tipos de privilegios (SELECT, DELETE) no se aplican a atributos en concreto ya que esta funcionalidad se puede controlar fácilmente mediante la creación de las vistas adecuadas que incluyan sólo los atributos deseados y la concesión de los privilegios correspondientes sobre las vistas. Sin embargo, debido a que la actualización de las vistas no es siempre posible (consulte el Capítulo 8), a los privilegios UPDATE e INSERT se les da la opción de indicar qué atributos en particular de una relación base pueden ser actualizados.

### 23.2.6 Especificación de los límites a la propagación de privilegios

Se han desarrollado técnicas para limitar la propagación de privilegios, aunque aún no han sido implementadas en la mayoría de los DBMSs y tampoco forman parte de SQL. Limitar la **propagación horizontal** a un número entero *i* significa que una cuenta *B* a la que se ha dado la opción GRANT OPTION puede conceder el privilegio a, como mucho, otras *i* cuentas. La **propagación vertical** es más complicada; limita la profundidad de la concesión de los privilegios. Conceder un privilegio con una propagación vertical igual a cero es equivalente a conceder el privilegio sin ninguna opción GRANT OPTION. Si la cuenta *A* concede un privilegio a la cuenta *B* con una propagación vertical igual a un número entero $j > 0$ significa que la cuenta *B* tiene la opción GRANT OPTION sobre ese privilegio, pero *B* sólo puede conceder el privilegio a otras cuentas con una propagación vertical *menor que j*. De hecho, la propagación vertical limita la secuencia de GRANT OPTIONS que pueden ser concedidas de una cuenta a la siguiente en base a la única concesión de privilegios original.

Pondremos un ejemplo (que no está disponible actualmente en SQL u otros sistemas relacionales) para ilustrar los límites de propagación horizontal y vertical. Supongamos que A1 concede SELECT a A2 sobre la relación EMPLEADO con propagación horizontal igual a 1 y propagación vertical igual a 2. A2 podría entonces conceder SELECT a, como mucho, una cuenta porque la limitación de propagación horizontal tiene un valor de 1. Además, A2 no puede conceder el privilegio a otra cuenta excepto las que tienen propagación vertical igual a 0 (sin opción GRANT OPTION) o igual a 1, debido a que A2 debe reducir la propagación vertical en, al menos, 1 al pasar el privilegio a otras cuentas. Según se muestra en este ejemplo, las técnicas de propagación horizontal y vertical están diseñadas para limitar el nivel de profundidad de la propagación de privilegios.

## 23.3 Control de acceso obligatorio y control de acceso basado en roles para la seguridad multinivel[2]

La técnica de control de acceso discrecional consistente en conceder y revocar privilegios sobre las relaciones ha sido tradicionalmente el principal mecanismo de seguridad en los sistemas de bases de datos relacionales. Se trata de un método de tipo "todo o nada". Un usuario, o tiene o no tiene un determinado privilegio. En muchas aplicaciones se necesita una *política de seguridad adicional* que clasifique los datos y los usuarios basándose en clases de seguridad. Este modelo, conocido como **control de acceso obligatorio**, se *combina* habitualmente con los mecanismos de control de acceso discrecional descritos en la sección 23.2. Es importante observar que la mayoría de los DBMSs comerciales sólo proporcionan en la actualidad mecanismos para el control de acceso discrecional. Sin embargo, existe una necesidad de seguridad a varios niveles en las aplicaciones gubernamentales, militares y en las de los servicios de inteligencia, así como en muchas aplicaciones industriales y corporativas.

[2] Se agradece la contribución de Fariborz Farahmand en esta sección y en las siguientes.

Las **clases de seguridad** típicas son alto secreto (TS), secreto (S), confidencial (C) y no clasificado (U), siendo TS el nivel más alto y U el más bajo. Existen otros esquemas de clasificación de seguridad más complejos, en los cuales las clases de seguridad se organizan en forma de rejilla. Por simplicidad utilizaremos el sistema de cuatro niveles de clasificación de seguridad, en el que TS $\geq$ S $\geq$ C $\geq$ U, para ilustrar nuestro debate. El modelo utilizado habitualmente para la seguridad a varios niveles, conocido como modelo Bell-LaPadula, clasifica cada **sujeto** (usuario, cuenta, programa) y **objeto** (relación, tupla, columna, vista, operación) en una de las clasificaciones de seguridad TS, S, C, o U. Llamaremos **clase(*S*)** al **nivel de autorización** (clasificación) de un sujeto *S* y **clase(*O*)** a la clasificación de un objeto *O*. Basándonos en las clasificaciones de sujeto/objeto podemos definir estas dos reglas:

1. A un sujeto *S* no se le permite el acceso de lectura a un objeto *O* a menos que la clase(*S*) $\geq$ clase(*O*). Esto se conoce como **propiedad de seguridad simple**.
2. A un sujeto *S* no se le permite escribir un objeto *O* a menos que la clase(*S*) $\leq$ clase(*O*). Esto se conoce como **propiedad estrella** (o propiedad-*).

La primera restricción es intuitiva y pone en evidencia la regla obvia que dice que ningún sujeto puede leer un objeto cuya clasificación de seguridad sea más alta que el nivel de autorización del sujeto. La segunda restricción es menos intuitiva. Impide que un sujeto escriba un objeto que se encuentre en una clasificación de seguridad menor que el nivel de autorización del sujeto. La violación de esta regla permitiría que la información fluyese desde clasificaciones más altas hacia las más bajas, lo cual viola uno de los principios básicos de la seguridad multinivel. Por ejemplo, un usuario (sujeto) con nivel de autorización TS puede hacer una copia de un objeto con clasificación TS y después volverlo a escribir como un nuevo objeto con clasificación U, haciendo que de este modo pase a ser visible en todo el sistema.

Para añadir conceptos de seguridad multinivel al modelo de base de datos relacional, es habitual considerar a los valores de atributos y a las tuplas como objetos de datos. De este modo, a cada atributo *A* se le asocia un **atributo de clasificación** *C* en el esquema, y a cada valor de atributo en una tupla se le asocia su correspondiente clasificación de seguridad. Además, en algunos modelos se añade un atributo de **clasificación de tupla** *TC* a los atributos de relación para proporcionar una clasificación a cada tupla en su totalidad. Según esto, un esquema de **relación multinivel** *R* con *n* atributos se representaría como

$$R(A_1, C_1, A_2, C_2, \ldots, A_n, C_n, TC)$$

donde cada $C_i$ representa el *atributo de clasificación* asociado a cada atributo $A_i$.

El valor de cada atributo *TC* en cada tupla *t* (que es *el más alto* de todos los valores de clasificación de atributos dentro de *t*) proporciona una clasificación general para la tupla, mientras que cada $C_i$ proporciona una clasificación de seguridad más en detalle dentro de la tupla. La **clave aparente** de una relación multinivel es el conjunto de atributos que habrían formado parte de la clave primaria en una relación normal (a un nivel). Una relación multinivel contendrá aparentemente datos diferentes para los sujetos (usuarios) con distintos niveles de seguridad. En algunos casos, es posible almacenar una única tupla de la relación en un nivel de clasificación más alto y generar las tuplas correspondientes en un nivel de clasificación inferior mediante un proceso llamado **filtrado**. En otros casos, es necesario almacenar dos o más tuplas de niveles de clasificación diferentes con el mismo valor para la *clave aparente*. Esto nos lleva al concepto de **poliinstanciación**,[3] según el cual varias tuplas pueden tener el mismo valor de clave aparente pero con distintos valores de atributo para los usuarios con diferentes niveles de clasificación.

Ilustraremos estos conceptos mediante el ejemplo sencillo de relación multinivel que se muestra en la Figura 23.2(a), en el cual aparecen los valores de atributos de clasificación junto a cada valor de atributo. Supongamos que el atributo Nombre es la clave aparente, y tomemos la consulta **SELECT * FROM** EMPLEADO. Un usuario con nivel de seguridad S vería la misma relación que se muestra en la Figura 23.2(a), ya que todas las clasificaciones de tupla son menores que o iguales a S. Sin embargo, un usuario con nivel de

[3] Esto es similar a la idea de tener varias versiones en la base de datos para representar el mismo objeto del mundo real.

**Figura 23.2.** Una relación multinivel para ilustrar la seguridad multinivel. (a) Las tuplas originales de EMPLEADO. (b) Apariencia de EMPLEADO tras filtrar para los usuarios de clasificación C. (c) Apariencia de EMPLEADO tras filtrar para los usuarios de clasificación U. (d) Poliinstanciación de la tupla Gómez.

**(a) EMPLEADO**

| Nombre | Sueldo | RendimientoTrabajo | TC |
|---|---|---|---|
| Gómez U | 40000 C | Normal S | S |
| Campos C | 80000 S | Bueno C | S |

**(b) EMPLEADO**

| Nombre | Sueldo | RendimientoTrabajo | TC |
|---|---|---|---|
| Gómez U | 40000 C | NULL C | C |
| Campos C | NULL C | Bueno C | C |

**(c) EMPLEADO**

| Nombre | Sueldo | RendimientoTrabajo | TC |
|---|---|---|---|
| Gómez U | NULL U | NULL U | U |

**(d) EMPLEADO**

| Nombre | Sueldo | RendimientoTrabajo | TC |
|---|---|---|---|
| Gómez U | 40000 C | Normal S | S |
| Gómez U | 40000 C | Excelente C | C |
| Campos C | 80000 S | Bueno C | S |

seguridad *C* no estaría autorizado a ver los valores de salario de 'Campos' y la evaluación de rendimiento en el trabajo de 'Gómez', ya que tienen una clasificación más alta. Las tuplas serían filtradas para que apareciesen como se muestra en la Figura 23.2(b), donde Sueldo y RendimientoTrabajo *aparecerían* como valor *nulo*. Para un usuario con nivel de seguridad U, el filtro permite que sólo aparezca el atributo Nombre de 'Gómez', y los otros atributos como valores nulos (véase la Figura 23.2[c]). De este modo, el filtrado introduce valores nulos para los valores de atributo cuya clasificación de seguridad es más alta que el nivel de seguridad del usuario.

En general, la regla de **integridad de entidad** para las relaciones multinivel obliga a que todos los atributos que sean miembros de la clave aparente no puedan ser valores nulos y a que tengan la *misma* clasificación de seguridad en cada tupla individual. Además, el resto de valores de atributo de la tupla deben tener una clasificación de seguridad mayor que o igual a la de la clave aparente. Esta restricción garantiza que un usuario pueda ver la clave si al usuario se le permite ver cualquier parte de la tupla. Otras reglas de integridad, llamadas **integridad nula** e **integridad interinstancia** aseguran, informalmente, que si un valor de tupla en un determinado nivel de seguridad se puede filtrar (derivar) desde una tupla que está a un nivel de seguridad más alto, entonces es suficiente con almacenar en la relación multinivel la tupla que está a mayor nivel.

Para ilustrar la poliinstanciación con más detalle, supongamos que un usuario con nivel de seguridad C intenta actualizar el valor de RendimientoTrabajo de 'Gómez' de la Figura 23.2 a 'Excelente'; esto se corresponde con la ejecución de la siguiente sentencia SQL de actualización:

```
UPDATE EMPLEADO
SET RendimientoTrabajo = 'Excelente'
WHERE Nombre = 'Gómez';
```

Ya que la vista que se muestra a los usuarios con nivel de seguridad C (véase la Figura 23.2[b]) permite una actualización de este tipo, el sistema no debería rechazarla; en caso contrario, el usuario podría deducir que existe algún valor no nulo para el atributo RendimientoTrabajo de 'Gómez' en lugar del valor nulo que aparece. Esto es un ejemplo de la inferencia de información a través de lo que se conoce como **canal oculto**, que no debería estar permitido en sistemas de alta seguridad (consulte la Sección 23.5.1). Sin embargo, al usuario no se le debería permitir sobrescribir el valor existente de RendimientoTrabajo en el nivel de clasificación más alto. La solución es crear una **poliinstanciación** de la tupla 'Gómez' en el nivel de clasificación más bajo C, según se muestra en la Figura 23.2(d). Esto es necesario ya que la nueva tupla no puede ser filtrada a partir de la tupla existente en el nivel de clasificación S.

Las operaciones básicas de actualización del modelo relacional (INSERT, DELETE, UPDATE) deben ser modificadas para tratar con situaciones parecidas a ésta, pero este aspecto del problema está fuera del ámbito de este texto. Los lectores interesados en el tema pueden dirigirse a la Bibliografía Seleccionada que se encuentra al final de este capítulo si desean más detalles.

### 23.3.1 Comparación del control de acceso discrecional y el control de acceso obligatorio

Las políticas de Control de acceso discrecional (DAC) se caracterizan por su alto grado de flexibilidad, lo cual las hace muy indicadas para una gran variedad de dominios de aplicación. El principal inconveniente de los modelos DAC es su vulnerabilidad a ataques maliciosos, como los caballos de Troya incluidos en programas de aplicación. La razón es que los modelos de autorización discrecional no imponen ningún control sobre cómo se propaga y se usa la información una vez que ha sido accedida por los usuarios autorizados a hacerlo. Por el contrario, las políticas obligatorias aseguran un alto grado de protección (dicho de otro modo, previenen cualquier flujo ilegal de información). Por tanto, son las indicadas para las aplicaciones de tipo militar, que requieren un alto grado de protección. Sin embargo, las políticas obligatorias tienen el inconveniente de ser demasiado rígidas, ya que requieren una clasificación estricta de los sujetos y de los objetos en niveles de seguridad y, por tanto, son aplicables a muy pocos entornos. En la práctica, en la mayoría de las situaciones se prefieren las políticas discrecionales, pues ofrecen una mejor relación entre seguridad y aplicabilidad.

### 23.3.2 Control de accesos basado en roles

El Control de acceso basado en roles (RBAC) surgió en muy poco tiempo en los años 90 como tecnología probada para la gestión y el reforzamiento de la seguridad en sistemas empresariales a gran escala. La idea básica es que los permisos están asociados a roles y a los usuarios se les asignan los roles apropiados. Los roles se pueden crear mediante los comandos CREATE ROLE y DESTROY ROLE. Los comandos GRANT y REVOKE que se vieron en el control de acceso discrecional (DAC) pueden ser usados para conceder y revocar privilegios a los roles.

El control de acceso basado en roles parece ser una alternativa viable a los controles de acceso discrecionales y obligatorios tradicionales; garantiza que sólo los usuarios autorizados tienen acceso a determinados datos o recursos. Los usuarios crean sesiones durante las cuales pueden activar un subconjunto de roles a los que pertenecen. Cada sesión se puede asignar a muchos roles, pero se corresponde con un único usuario o sujeto. Muchos DBMSs permiten el concepto de roles que consiste en asignar privilegios a los mismos.

La jerarquía de roles en RBAC es el modo natural de organizar los roles para que reflejen la jerarquía de autoridad y de responsabilidades en la organización. Por convención, los roles de menor responsabilidad de la parte inferior se conectan con roles de responsabilidad cada vez mayor a medida que se sube por la jerarquía. Los diagramas de jerarquía son órdenes parciales, por tanto son reflexivos, transitivos y antisimétricos.

Otra idea importante en los sistemas RBAC es la posibilidad de que existan restricciones temporales en los roles, como el tiempo y la duración de las activaciones de los roles, y el disparo temporizado de un rol median-

te la activación de otro rol. La utilización de un modelo RBAC es una meta a alcanzar para conseguir algo altamente deseable que es el tratamiento de los requisitos clave de seguridad en las aplicaciones web. Se pueden asignar roles a las tareas del flujo de trabajo para que se pueda autorizar a un usuario con uno de los roles relacionados con una tarea la ejecución de ésta y la actuación con un rol durante un determinado periodo de tiempo.

Los modelos RBAC tienen varias características interesantes, como la flexibilidad, la neutralidad en las políticas, un mejor soporte para la gestión de la seguridad y de la administración, y otros aspectos que los hacen candidatos adecuados para el desarrollo de aplicaciones web seguras. Por el contrario, a los modelos DAC y otros de control de acceso obligatorio (MAC) les faltan las capacidades necesarias para proporcionar los requisitos de seguridad de compañías emergentes y de aplicaciones web. Además, los modelos RBAC pueden representar políticas DAC y MAC tradicionales, así como políticas definidas por el usuario o específicas de una organización. De este modo, RBAC se convierte en un gran modelo que puede, a su vez, imitar el comportamiento de los sistemas DAC y MAC. Además, un modelo RBAC proporciona un mecanismo natural para el tratamiento de los temas de seguridad relacionados con la ejecución de tareas y flujos de trabajo. Otra razón para el éxito de los modelos RBAC ha sido la facilidad para el desarrollo en Internet.

### 23.3.3 Control de acceso XML

Debido a la extensión del uso de XML en las aplicaciones científicas y comerciales, se están realizando esfuerzos orientados al desarrollo de estándares de seguridad. Entre estos esfuerzos se encuentran las firmas digitales y los estándares de criptografía para XML. La especificación de sintaxis y procesamiento de firmas en XML describe una sintaxis XML para la representación de las asociaciones entre firmas criptográficas y documentos XML u otros recursos electrónicos. La especificación también incluye procedimientos para el cálculo y la verificación de firmas XML. Una firma digital XML se diferencia de otros protocolos de firma de mensajes como PGP (*Pretty Good Privacy*, un servicio de autenticación y confidencialidad que puede ser utilizado en el correo electrónico y en las aplicaciones de almacenamiento de archivos), en que proporciona la posibilidad de firmar sólo partes determinadas del árbol XML (consulte el Capítulo 26) en lugar del documento completo. Además, la especificación de firma XML define mecanismos de contrafirma y transformaciones (llamadas canonizaciones para asegurar que las dos instancias del mismo texto producen el mismo valor resumen para la firma, incluso si sus representaciones difieren ligeramente). La especificación de sintaxis y procesamiento de cifrado en XML define el vocabulario XML y las reglas de procesamiento para proteger la confidencialidad de los documentos XML en su totalidad o en parte y también la de los datos no XML. El contenido cifrado y la información de procesamiento adicional para el receptor se representan en XML "bien formado" de forma que el resultado pueda ser procesado posteriormente utilizando herramientas XML. A diferencia de otras tecnologías utilizadas habitualmente para la confidencialidad como SSL (*Secure Socket Layer*, un protocolo importante en la seguridad en Internet) y las redes privadas virtuales, el cifrado XML también se aplica a partes de los documentos que se encuentran en almacenamiento persistente.

### 23.3.4 Políticas de control de acceso para el comercio electrónico y la Web

Los entornos de comercio electrónico (**E-commerce**) se caracterizan por transacciones que se realizan de forma electrónica. Requieren políticas de control de acceso detalladas que van más allá de las de los DBMSs tradicionales. En los entornos de bases de datos convencionales, el control de acceso se realiza normalmente mediante un conjunto de autorizaciones definidas por los encargados de la seguridad o por los usuarios de acuerdo a determinadas políticas de seguridad. Este paradigma no se ajusta demasiado bien a un entorno dinámico como el comercio electrónico. Además, en un entorno de comercio electrónico los recursos a proteger no son sólo datos de tipo tradicional, sino que incluyen también el conocimiento y la experiencia. Estas peculiaridades requieren más flexibilidad al especificar políticas de control de acceso. El mecanismo de control de

acceso debe ser lo suficientemente flexible para proporcionar un amplio espectro de objetos heterogéneos a proteger.

Un segundo requisito relacionado con esto es la posibilidad de controlar el acceso basado en contenidos. El control de acceso basado en contenidos permite expresar políticas de control de acceso que tomen en cuenta el contenido del objeto a proteger. Para proporcionar el control de acceso basado en contenidos, las políticas de control de acceso deben permitir la inclusión de condiciones basadas en el contenido del objeto.

Un tercer requisito está relacionado con la heterogeneidad de los sujetos, lo cual necesita de políticas de control basadas en las características del usuario y sus cualificaciones, en lugar de basarse en características específicas e individuales (por ejemplo, los identificadores de usuario). Una posible solución para tener en cuenta los perfiles de usuario en la formulación de políticas de control de acceso es la existencia de la noción de credenciales. Una **credencial** es un conjunto de propiedades en relación con un usuario que son relevantes para los propósitos de seguridad (por ejemplo, la edad o el puesto dentro de la organización). Por ejemplo, al usar credenciales podemos simplemente formular políticas del tipo *Sólo los empleados fijos con cinco o más años de antigüedad pueden acceder a documentos relacionados con la organización interna del sistema*.

Se piensa que el lenguaje XML puede jugar un papel clave en el control de acceso en las aplicaciones de comercio electrónico[4] ya que XML se está convirtiendo en el lenguaje de representación habitual en el intercambio de documentos en la Web y también en el lenguaje para el comercio electrónico. Según esto, por una parte existe la necesidad de hacer que las representaciones XML sean seguras proporcionando mecanismos de control de acceso diseñados específicamente para la protección de los documentos XML. Por otra parte, la información de control de acceso (es decir, las políticas de control de acceso y las credenciales de usuario) se puede expresar utilizando XML. El Lenguaje de marcado de servicios de directorio (DSML) proporciona una base para lo siguiente: un estándar para la comunicación con los servicios de directorio que serán responsables de proporcionar y autenticar las credenciales de usuario. La presentación uniforme de los objetos a proteger y de las políticas de control de acceso se puede aplicar a las propias políticas y credenciales. Por ejemplo, algunas propiedades de las credenciales (como el nombre de usuario) pueden ser accesibles a todo el mundo, mientras que otras propiedades pueden ser visibles sólo a ciertas clases restringidas de usuarios. Además, el uso de un lenguaje basado en XML para especificar credenciales y políticas de control de acceso facilita el envío seguro de las credenciales y la exportación de las políticas de control de acceso.

## 23.4 Introducción a la seguridad en bases de datos estadísticas

Las bases de datos estadísticas se utilizan principalmente para obtener estadísticas sobre poblaciones diversas. La base de datos puede contener información confidencial sobre individuos que debería ser protegida de su acceso por parte de los usuarios. Sin embargo, a los usuarios se les permite obtener información estadística sobre la población, como medias de valores, totales, contadores, máximos, mínimos y desviaciones estándar. Las técnicas que se han desarrollado para proteger la privacidad de la información de las personas se encuentran fuera del ámbito de este libro. Ilustraremos el problema mediante un ejemplo muy sencillo, basado en la relación que se muestra en la Figura 23.3. Se trata de una relación PERSONA con los atributos Nombre, Dni, Ingresos, Direcc, Ciudad, Estado, CP, Sexo y Estudios.

**Figura 23.3.** El esquema de la relación PERSONA para ilustrar la seguridad en bases de datos estadísticas.

**PERSONA**

| Nombre | Dni | Ingresos | Direcc | Ciudad | Provincia | CP | Sexo | Estudios |
|---|---|---|---|---|---|---|---|---|

[4] Véase Thuraisingham y otros (2001).

Una **población** es un conjunto de tuplas de una relación (tabla) que satisfacen una condición de selección. Según esto, cada condición de selección en la relación PERSONA especificará una población en particular de tuplas de PERSONA. Por ejemplo, la condición Sexo = 'H' especifica la población de varones; la condición ((Sexo = 'F') AND (Estudios = 'Licenciado' OR Estudios = 'Doctor')) especifica la población femenina que tiene una titulación de Licenciado o Doctor; y la condición Ciudad = 'Madrid' especifica la población que vive en Madrid.

Las consultas estadísticas implican aplicar funciones estadísticas a una población de tuplas. Por ejemplo, podríamos querer obtener el número de individuos de una población o los ingresos medios de la población. Sin embargo, a los usuarios estadísticos no se les permite obtener datos individuales, como los ingresos de una persona determinada. Las técnicas de **seguridad en bases de datos estadísticas** deben prohibir la obtención de datos individuales. Esto se puede conseguir mediante la prohibición de consultas que obtengan valores de atributos y permitiendo sólo las consultas que impliquen funciones estadísticas de agregación como COUNT, SUM, MIN, MAX, AVERAGE y STANDARD DEVIATION. A las consultas de este tipo se les llama a veces **consultas estadísticas**.

Es responsabilidad de un sistema de gestión de bases de datos asegurar la confidencialidad de la información relacionada con los individuos y, a la vez, proporcionar a los usuarios resúmenes estadísticos de los datos acerca de esos individuos. Garantizar la **protección de la privacidad** de los usuarios en una base de datos estadística es lo más importante; su incumplimiento aparece documentado en el siguiente ejemplo.

En algunos casos es posible **deducir** los valores de tuplas individuales a partir de una secuencia de consultas estadísticas. Esto es particularmente cierto cuando las condiciones dan como resultado una población formada por un número pequeño de tuplas. Como muestra, supongamos las siguientes consultas estadísticas:

**C1:** **SELECT COUNT** (*∗) **FROM** PERSONA
**WHERE** <condición>;

**C2:** **SELECT AVG** (Ingresos) **FROM** PERSONA
**WHERE** <condición>;

Supongamos ahora que estamos interesados en encontrar el Sueldo de Ana Gómez y que sabemos que tiene el título de Doctora y que vive en la ciudad de Getafe, Madrid. Ejecutaríamos la consulta estadística C1 con la siguiente condición

(Estudios='Doctor' AND Sexo='F' AND Ciudad='Getafe' AND Provincia='Madrid')

Si obtenemos un resultado igual a 1 en esta consulta, podremos ejecutar la consulta C2 con la misma condición y descubrir el sueldo de Ana Gómez. Incluso si el resultado de C1 con la condición anterior no es 1 pero es un valor pequeño (por ejemplo, 2 ó 3), podremos ejecutar consultas estadísticas usando las funciones MAX, MIN y AVERAGE para identificar el posible rango de valores para el Sueldo de Ana Gómez.

La posibilidad de deducir información individual a partir de consultas estadísticas se reduce si no se permiten las consultas estadísticas siempre que el número de tuplas de la población especificada por la condición de selección esté por debajo de determinado nivel. Otra técnica para prohibir la obtención de información individual es imposibilitar las secuencias de consultas que hagan referencia repetidamente a la misma población de tuplas. También es posible introducir deliberadamente ligeras imperfecciones o *ruido* en los resultados de las consultas estadísticas para hacer que sea difícil deducir información individual a partir de los resultados. Otra técnica consiste en particionar la base de datos. El particionamiento implica que los registros se almacenan en grupos de un tamaño mínimo; las consultas pueden hacer referencia a cualquier grupo o conjunto de grupos completos, pero nunca a subconjuntos de registros dentro de un grupo. Los lectores interesados en el tema pueden dirigirse a la Bibliografía Seleccionada que se encuentra al final de este capítulo si desean una revisión de estas técnicas.

# 23.5 Introducción al control de flujo

El control de flujo regula la distribución o flujo de información entre objetos accesibles. Un flujo entre el objeto *X* y el objeto *Y* sucede cuando un programa lee valores de *X* y escribe valores en *Y*. Los **controles de flujo** comprueban que la información contenida en algunos objetos no fluye explícita o implícitamente hacia objetos menos protegidos. De este modo, un usuario no puede poner de forma indirecta en *Y* lo que no pueda obtener de forma directa en *X*. El control de flujo activo surgió a principios de los años 70. La mayoría de los flujos de control emplean algún concepto de clase de seguridad; la transferencia de información desde un emisor hasta un receptor sólo se permite si la clase de seguridad del receptor tiene al menos tantos privilegios como la del emisor. Entre los ejemplos de control de flujo podemos incluir el evitar que un programa de servicio guarde datos confidenciales de un cliente y el bloqueo de la transmisión de datos secretos militares a un usuario desconocido.

Una **política de flujo** especifica los canales a través de los cuales se permite que se mueva la información. La política de flujo más sencilla especifica sólo dos clases de información: confidencial (*C*) y no confidencial (*N*), y permite todos los flujos excepto aquellos que se dirigen desde la clase C hasta la clase N. Esta política puede resolver el problema de aislamiento que surge cuando un programa trata datos como información sobre clientes, algunos de los cuales pueden ser confidenciales. Por ejemplo, un programa de servicio para cálculo de impuestos podría estar autorizado a guardar la dirección del cliente y el importe del servicio prestado, pero no los ingresos de ese cliente o sus deducciones de impuestos.

Los mecanismos de control de acceso son los encargados de comprobar las autorizaciones de los usuarios para acceder a los recursos: sólo se ejecutan las operaciones autorizadas. Se pueden reforzar los controles de flujo mediante un mecanismo ampliado de control de acceso, lo cual implica la asignación de una clase de seguridad (llamada normalmente *autorización*) a cada programa en ejecución. Al programa se le permite leer un segmento de memoria en particular sólo si su clase de seguridad es tan alta como la del segmento. Se le permite escribir en un segmento sólo si su clase es de tan bajo nivel como la del segmento. Esto nos asegura, de forma automática, que la información transmitida por el usuario se puede mover desde una clase más alta hacia una clase más baja. Por ejemplo, un programa militar con una autorización de tipo secreto podría leer sólo objetos de tipo no clasificado o confidencial y escribir sólo en objetos de tipo secreto o alto secreto.

Se pueden distinguir dos tipos de flujo: *flujos explícitos*, que tienen lugar como consecuencia de instrucciones de asignación como $Y:= f(X_1,X_n,)$; y *flujos implícitos* generados por instrucciones condicionales, como if $f(X_{m+1}, \ldots, X_n)$ then $y:= f(X_1,X_m)$.

Los mecanismos de control de flujo deben comprobar que sólo se ejecutan los flujos autorizados, sean de tipo explícito o implícito. Para garantizar la seguridad en los flujos de información se deben cumplir un conjunto de reglas. Las reglas pueden ser expresadas mediante relaciones de flujo entre clases y pueden ser asignadas a la información para declarar los flujos autorizados dentro de un sistema. (Un flujo de información de A a B se produce cuando la información asociada a A afecta al valor de la información asociada a B. El flujo aparece como resultado de las operaciones que hacen que la información pase de un objeto al otro). Este tipo de relaciones puede definir, para el caso de una clase, el conjunto de clases en donde puede fluir la información (clasificada en esa clase), o puede especificar las relaciones específicas entre dos clases que tienen que ser comprobadas entre dos clases para permitir que la información fluya de la una a la otra. En general, los mecanismos de control de flujo implementan los controles asignando una etiqueta a cada objeto y especificando la clase de seguridad del objeto. Las etiquetas serán utilizadas posteriormente para verificar las relaciones de flujo definidas en el modelo.

## 23.5.1 Canales ocultos

Un canal oculto permite las transferencias de información que violan la seguridad o la política de seguridad. En concreto, un **canal oculto** permite que la información pase desde un nivel de clasificación más alto hacia

un nivel de clasificación más bajo a través de algún medio que no es el normal. Los canales ocultos se pueden clasificar en dos grandes grupos: canales de temporización y de almacenamiento. Lo que distingue a ambos es que en un **canal de temporización** la información se transmite tras una temporización de eventos o procesos mientras que los **canales de almacenamiento** no necesitan una sincronización temporal, ya que la información se transmite accediendo a información del sistema o a información que sería inaccesible al usuario de otro modo.

Como ejemplo sencillo de canal oculto, tomemos un sistema de base de datos distribuida en la cual dos nodos tienen niveles de seguridad de usuario igual a secreto (S) y no clasificado (U). Para que se ejecute una transacción, ambos nodos tienen que ponerse de acuerdo. Ambos en conjunto sólo pueden realizar operaciones que sean consistentes con la propiedad-*, que indica que en cualquier transacción, el elemento S no puede escribir o pasar información al elemento U. Sin embargo, si ambos sitios se ponen de acuerdo para establecer un canal oculto entre ellos, una transacción que contenga datos secretos podrá ser ejecutada incondicionalmente por el sitio U, pero el sitio S sólo podrá hacer lo mismo de algún modo predefinido y convenido para que determinada información pueda ser transferida desde el elemento S al elemento U, violando la propiedad-*. Esto se puede conseguir en los casos en los que la transacción se ejecute repetidamente y las acciones llevadas a cabo por el elemento S comuniquen información implícitamente al elemento U. Ciertas medidas, como el bloqueo, que vimos en los Capítulos 17 y 18, impiden la escritura concurrente de la información por parte de usuarios con distintos niveles de seguridad en los mismos objetos, impidiendo los canales ocultos de tipo almacenamiento. Los sistemas operativos y las bases de datos distribuidas proporcionan control sobre la multiprogramación de operaciones que permiten compartir recursos sin la posibilidad de que un proceso o programa se apropie de la memoria de otro o de los recursos del sistema, previniendo de este modo los canales ocultos de tipo temporización. En general, los canales ocultos no son un gran problema cuando las implementaciones de las bases de datos están bien construidas y son robustas. Sin embargo, ciertos esquemas pueden ser engañados por usuarios inteligentes que realicen transferencias de información de manera implícita.

Algunos expertos en seguridad creen que una forma de evitar los canales ocultos es impedir a los programadores obtener acceso a datos confidenciales que serán procesados por un programa una vez se encuentre en ejecución. Por ejemplo, un programador que trabaje para un banco no tiene necesidad de acceder a los nombres o al saldo de la cuenta de los clientes. Los programadores que trabajen para compañías de bolsa no necesitan saber qué órdenes de compra o de venta van a realizar los clientes. Durante las pruebas del programa podría estar justificado el acceso a partes de los datos reales o a datos de ejemplo, pero no una vez que se ha aceptado el programa para su uso en el entorno real.

## 23.6 Cifrado e infraestructuras de clave pública

Los anteriores métodos de acceso y de control de flujo, a pesar de ser medidas de control bastante fuertes, podrían no ser capaces de proteger a las bases de datos contra algunas amenazas. Supongamos que vamos a transmitir datos, pero estos caen en las manos de un usuario no autorizado. En esa situación, utilizando el cifrado podemos disfrazar el mensaje para que incluso en el caso de que la transmisión sea interceptada, el contenido del mensaje no sea descubierto. El cifrado es un medio de mantener la seguridad de los datos en un entorno no seguro. El cifrado consiste en aplicar un algoritmo de cifrado a los datos utilizando una clave de cifrado predefinida. Los datos resultantes tienen que ser descifrados mediante una clave de descifrado para recuperar los datos originales.

### 23.6.1 Los datos y estándares avanzados de cifrado

El estándar de cifrado de datos (DES) es un sistema desarrollado por el gobierno de Estados Unidos para su uso por el público en general. Es ampliamente aceptado como estándar de cifrado dentro y fuera de EE.UU. DES proporciona cifrado extremo a extremo en el canal existente entre el emisor A y el receptor B. El algo-

ritmo DES es una cuidadosa y compleja combinación de dos de los bloques fundamentales usados en la construcción del cifrado: la sustitución y la permutación (transposición). La fuerza del algoritmo deriva de la aplicación repetida de estas dos técnicas durante un total de 16 ciclos. El texto plano (la forma original del mensaje) se cifra en bloques de 64 bits. Aunque la clave tiene una longitud de 64 bits, en realidad puede ser cualquier número de 56 bits. Tras cuestionar la validez de DES, el Instituto Nacional de Estándares (NIST, *National Institute of Standards*) introdujo el algoritmo AES (*Advanced Encryption Standards*). Este algoritmo tiene un tamaño de bloque de 128 bits, a diferencia de los 56 bits del bloque de DES, y puede utilizar claves de 128, 192 o 256 bits, a diferencia de los 56 bits de la clave de DES. AES introduce más claves posibles, a diferencia de DES y, por tanto, el tiempo necesario para romperlo es mayor.

### 23.6.2 Cifrado con clave pública

En 1976, Diffie y Hellman propusieron un nuevo tipo de sistema de cifrado, al cual llamaron **cifrado con clave pública**. Los algoritmos de clave pública se basan en funciones matemáticas en lugar de patrones de bits. También requieren el uso de dos claves independientes, a diferencia del cifrado convencional que utiliza una única clave. El uso de dos claves puede tener grandes consecuencias en el ámbito de la confidencialidad, la distribución de claves y la autenticación. A las dos claves que se usan para el cifrado con clave pública se les llama **clave pública** y **clave privada**. Invariablemente, la clave privada se mantiene en secreto, pero se le denomina *clave privada* en lugar de *clave secreta* (la clave que se usa en el cifrado convencional) para evitar confusiones con el cifrado convencional.

Un esquema, o infraestructura, de cifrado con clave pública tiene seis componentes:

1. **Texto plano o sin formato.** Son los datos o el mensaje legible que se toma como entrada al algoritmo.
2. **Algoritmo de cifrado.** Este algoritmo realiza diversas transformaciones al texto plano.
3. y **4. Claves pública y privada.** Son un par de claves que han sido elegidas de forma que si una de ellas se utiliza para cifrar, la otra se podrá usar para descifrar. Las transformaciones realizadas por el algoritmo de cifrado dependen de la clave pública o privada que se suministre como entrada.
5. **Texto cifrado.** El mensaje ininteligible generado como resultado. Depende del texto plano y de la clave. Para un mensaje en concreto, dos claves diferentes producirán dos textos cifrados diferentes.
6. **Algoritmo de descifrado.** Este algoritmo toma como entrada el texto cifrado y la clave correspondiente y genera el texto plano original.

Como indica su nombre, la clave pública del par se hace pública a los demás para que la utilicen, mientras que la clave privada sólo la conoce su propietario. Un algoritmo de cifrado con clave pública de propósito general se basa en una clave para el cifrado y otra diferente, pero relacionada con la anterior, para descifrar. Los pasos más importantes se muestran a continuación:

1. Cada usuario genera un par de claves a utilizar en el cifrado y descifrado de mensajes.
2. Cada usuario coloca una de las dos claves en un registro público u otro archivo accesible. Ésta es la clave pública. La otra clave se mantendrá en privado.
3. Si un emisor desea enviar un mensaje privado a un receptor, el emisor cifrará el mensaje utilizando la clave pública del receptor.
4. Cuando el receptor reciba el mensaje, lo descifrará utilizando la clave privada del receptor. Ningún otro destinatario puede descifrar el mensaje, ya que sólo el receptor conoce su clave privada.

**Algoritmo RSA de cifrado con clave pública.** Uno de los primeros esquemas de clave pública fue introducido en 1978 por Ron Rivest, Adi Shamir y Len Adleman del MIT y se llama esquema RSA siguiendo sus iniciales. El esquema RSA ha sido desde entonces uno de los más extendidos y uno de los modelos de cifrado con clave pública más implementados. El algoritmo de cifrado RSA incorpora nociones de la teoría numé-

rica combinadas con la dificultad de obtención de los factores primos de un elemento. El algoritmo RSA también trabaja con la aritmética modular (mod $n$).

Se utilizan dos claves, $d$ y $e$ para el cifrado y descifrado. Una propiedad importante es que son intercambiables entre sí. Se elige $n$ como el entero más grande que sea el producto de dos números primos grandes y distintos, $a$ y $b$. La clave de cifrado $e$ es un número elegido al azar entre 1 y $n$ que sea primo relativo a $(a - 1) \times (b - 1)$. El bloque de texto plano $P$ se cifra como $P^e$ mod $n$. Ya que la exponenciación se realiza en módulo $n$, resulta difícil factorizar $P^e$ para descubrir el texto plano que ha sido cifrado. Sin embargo, la clave de cifrado $d$ se elige cuidadosamente para que $(P^e)^d$ mod $n = P$. La clave de cifrado $d$ se puede calcular a partir de la condición $d \times e = 1 \bmod ((a - 1) \times (b - 1))$. Por tanto, al receptor autorizado que conozca $d$ le basta con calcular $(P^e)^d$ mod $n = P$ y recuperará $P$ sin tener que factorizar $P^e$.

### 23.6.3 Firmas digitales

Una firma digital es un ejemplo del uso de las técnicas de cifrado para proporcionar servicios de autenticación en las aplicaciones de comercio electrónico. Al igual que en el caso de la firma escrita a mano, una **firma digital** es un mecanismo para asociar una identificación única para un individuo a un cuerpo de texto. La marca debe ser perdurable, en el sentido de que otros deberían ser capaces de comprobar que la firma procede de su emisor.

Una firma digital está formada por una secuencia de símbolos. Si la firma digital de una persona fuese siempre la misma para todos los mensajes, alguien podría falsificarla fácilmente con sólo copiar la secuencia de símbolos. Por tanto, las firmas deben ser diferentes en cada uso. Esto se puede conseguir haciendo que cada firma digital se genere en función del mensaje que se está firmando, junto con una marca de tiempo. Para que sea única para cada usuario y a prueba de falsificaciones, cada firma digital debe depender también de algún número secreto que sea único para cada firmante. Según esto, en general, una firma digital a prueba de falsificaciones debe depender del mensaje y de un número secreto único para el firmante. Sin embargo, a la hora de comprobar la firma, no debe ser necesario conocer ningún número secreto. Las técnicas de clave pública son el mejor medio de creación de firmas digitales que cumplan estas propiedades.

## 23.7 Mantenimiento de la privacidad

Mantener la privacidad es un desafío cada vez mayor para la seguridad en las bases de datos y para los expertos en privacidad. En cierto modo, para mantener la privacidad de los datos deberíamos incluso limitar la ejecución de los análisis y obtención de datos a gran escala. Las técnicas utilizadas más habitualmente para tratar estos temas son evitar la construcción de grandes almacenes de datos centralizados en un único repositorio con toda la información importante. Otra posible medida es la modificación o la distorsión intencionada de los datos.

Si todos los datos se encuentran disponibles en un único almacén, la violación de la seguridad en sólo un único repositorio podría dejar accesibles todos los datos. Si se evitan los almacenes de datos centralizados y se utilizan algoritmos distribuidos de obtención de datos se minimiza el intercambio de datos necesarios para el desarrollo de modelos válidos a nivel global. Si se modifican, distorsionan o se hacen anónimos los datos, podemos también mitigar los riesgos en la privacidad asociados a la minería de datos. Esto se puede conseguir eliminando la información de identidad en los datos obtenidos e inyectando ruido en ellos. Sin embargo, al utilizar estas técnicas deberíamos prestar atención a la calidad de los datos resultantes en la base de datos, ya que podrían quedar sometidos a demasiadas modificaciones. Debemos ser capaces de calcular los errores que puedan ser introducidos por estas modificaciones.

La privacidad es un área importante en el desarrollo de las investigaciones en gestión de bases de datos. Resulta complicado debido a su naturaleza multidisciplinar y a los temas relacionados con la subjetividad en la interpretación de la privacidad, fiabilidad, etc. Como ejemplo, tomemos los registros y las transacciones

médicas y de tipo legal, que deben mantener ciertos requisitos de privacidad cuando son definidos y reforzados. Actualmente, también se están dedicando esfuerzos a proporcionar control de acceso y privacidad en los dispositivos móviles. Los DBMS tienen la necesidad de técnicas sólidas para el almacenamiento eficaz de información relevante en cuanto a seguridad sobre dispositivos de pequeño tamaño, y también de técnicas de negociación de relaciones de confianza. Aún resulta un problema importante saber dónde almacenar información relacionada con identidades de usuario, perfiles, credenciales y permisos, y cómo utilizarla para la identificación fiable de los usuarios. Ya que en esos entornos se genera un gran flujo de datos, es necesario desarrollar técnicas eficientes para el control de accesos que se encuentren integradas con las técnicas de procesamiento continuo de consultas. Por último, se debe asegurar la privacidad de datos relacionados con la localización física de usuarios que hayan sido adquiridos a través de detectores y redes de comunicaciones.

# 23.8 Retos en la seguridad en las bases de datos

Pensemos en el gran crecimiento en volumen y en la rapidez de aparición de las amenazas a las bases de datos y a la información de valor; los esfuerzos en investigación se deben orientar a los siguientes temas: calidad de los datos, derechos de propiedad intelectual y supervivencia en bases de datos.

## 23.8.1 Calidad de los datos

La comunidad de bases de datos necesita técnicas y soluciones organizativas para evaluar y demostrar la calidad de los datos. Estas técnicas pueden incluir mecanismos sencillos como los sellos de calidad que se insertan en los sitios web. Necesitamos también técnicas que proporcionen reglas semánticas de integridad más efectivas y herramientas para la evaluación de la calidad de los datos basadas en técnicas como el enlace de registros. Se necesitan también técnicas de recuperación a nivel de aplicación para la reparación automática de datos incorrectos. Actualmente, para estos temas se está intentando el uso de las herramientas ETL (extraer, transformar, cargar) usadas ampliamente en la carga de datos en los almacenes de datos (consulte la Sección 29.4).

## 23.8.2 Derechos de propiedad intelectual

Con la extensión del uso de Internet y de las tecnologías Intranet, los aspectos legales e informativos de los datos se están convirtiendo en una de las principales preocupaciones de las organizaciones. Para tratar estos temas, se han propuesto recientemente unas técnicas de marca de agua digital para bases de datos. El propósito principal de las marcas de agua digitales es proteger el contenido frente a la duplicación y distribución no autorizadas mediante la creación de una prueba sobre la propiedad del contenido. Tradicionalmente, esto se ha conseguido basándose en la disponibilidad de un amplio dominio de ruido dentro del cual se puede alterar el objeto manteniendo, a su vez, sus propiedades esenciales. Sin embargo, es necesario seguir trabajando para evaluar la robustez de estas técnicas y para descubrir diferentes modelos orientados a prevenir la violación de los derechos de propiedad intelectual.

## 23.8.3 Supervivencia en las bases de datos

Los sistemas de bases de datos necesitan mantenerse operativos y seguir desarrollando sus funciones a pesar de que se produzcan sucesos que les afecten negativamente, como los ataques contra la información que contienen. Un DBMS, además de estar preparado contra un ataque y de poder detectarlo en caso de que suceda, debería ser capaz de realizar lo siguiente:

- **Aislamiento.** Llevar a cabo acciones inmediatas para eliminar el acceso del atacante al sistema y aislar o contener el problema para evitar que se extienda.
- **Evaluación de los daños.** Determinar el alcance del problema, incluyendo las funciones afectadas y los datos corrompidos.
- **Reconfiguración.** Reconfigurar para permitir que continúe en funcionamiento aún estando dañado mientras se lleva a cabo la recuperación.
- **Reparación.** Reparación de los datos perdidos o corrompidos y reparación o reinstalación de las funciones dañadas del sistema para establecer un nivel normal de funcionamiento.
- **Tratamiento de los fallos.** Hasta el punto que sea posible, identificar las debilidades que han sido aprovechadas por el ataque y tomar medidas para prevenir que vuelva a ocurrir.

El objetivo de un atacante de un sistema es dañar el funcionamiento de la organización y cumplir su misión mediante la interrupción de sus sistemas de información. El objetivo específico de un ataque puede ser bien el sistema o bien sus datos. Aunque los ataques que dejan el sistema fuera de servicio son muy graves, deben ser planificados para poder conseguir el objetivo del atacante, ya que esos ataques recibirán una atención inmediata y enérgica para volver a poner el sistema en estado operativo, para diagnosticar cómo se produjo el ataque y para instalar medidas preventivas.

A fecha de hoy, los temas relacionados con la supervivencia de las bases de datos aún no han sido investigados suficientemente. Se necesita mucha más investigación dedicada a las técnicas y metodologías que aseguren la supervivencia de las bases de datos.

## 23.9 Resumen

En este capítulo hemos visto varias técnicas para reforzar la seguridad en las bases de datos. Hemos presentado diferentes amenazas a las bases de datos en términos de pérdida de integridad, disponibilidad y confidencialidad. Revisamos los tipos de medidas de control que tratan con estos problemas: control de accesos, control de inferencias, control de flujo y cifrado.

El reforzamiento de la seguridad tiene relación con el control de los accesos al sistema de base de datos en su totalidad y con el control de la autorización para acceder a determinadas partes de una base de datos. Lo primero se lleva a cabo habitualmente mediante la asignación de cuentas con contraseñas a los usuarios. Lo último se puede realizar utilizando un sistema de concesión y revocación de privilegios a cuentas individuales para acceder a partes concretas de la base de datos. A este modelo se le denomina, por lo general, control de acceso discrecional. Mostramos algunos comandos SQL para conceder y revocar privilegios, e ilustramos su uso mediante ejemplos. Posteriormente, hicimos un repaso a los mecanismos de control de acceso obligatorio que refuerzan la seguridad multinivel. Éstos requieren la clasificación de los usuarios y de los valores de los datos en clases de seguridad y refuerzan las reglas que prohíben el flujo de información desde niveles de seguridad altos hacia los más bajos. Se presentaron algunos de los conceptos clave que subyacen al modelo relacional multinivel, incluidos el filtrado y la poliinstanciación. Se mostró el control de acceso basado en roles, en el cual se asignan privilegios basados en roles desempeñados por los usuarios. Repasamos brevemente el problema del control de acceso a las bases de datos estadísticas para proteger la privacidad de la información de los individuos a la vez que se proporciona acceso estadístico a las poblaciones de registros. Los temas relacionados con el control de flujo y los problemas asociados a los canales ocultos se revisaron a continuación. También vimos el cifrado de datos, incluyendo la infraestructura de clave pública y las firmas digitales. Resaltamos la importancia de los asuntos relacionados con la privacidad y dimos unas ideas sobre técnicas de salvaguardia de la privacidad. Revisamos varios retos a la seguridad entre los que incluimos la calidad de los datos, los derechos de propiedad intelectual y la supervivencia de los datos.

## Preguntas de repaso

**23.1.** Comente qué se entiende por cada uno de los siguientes términos: autorización en bases de datos, control de acceso, cifrado de datos, cuentas privilegiadas (de sistema), auditorías de bases de datos y registro de auditoría.

Comente los tipos de privilegios a nivel de cuenta y a nivel de relación.

**23.2.** ¿Qué cuenta es la que se designa como propietaria de la relación? ¿Qué privilegios tiene el propietario de una relación?

**23.3.** ¿Cómo se usa el mecanismo de vista como mecanismo de autorización?

**23.4.** ¿Qué significa conceder un privilegio?

**23.5.** ¿Qué significa revocar un privilegio?

**23.6.** Comente el sistema de propagación de privilegios y las restricciones impuestas por los límites de propagación horizontal y vertical.

**23.7.** Enumere los tipos de privilegios disponibles en SQL.

**23.8.** ¿Cuál es la diferencia entre control de acceso *obligatorio* y *discrecional*?

**23.9.** ¿Cuáles son las clasificaciones habituales de seguridad? Comente la propiedad de seguridad simple y la propiedad-$*$, y explique qué justificación hay tras estas reglas para reforzar la seguridad multinivel.

**23.10.** Describa el modelo de datos relacional multinivel. Defina los siguientes términos: *clave aparente*, *poliinstantiación* y *filtrado*.

**23.11.** ¿Cuáles son los beneficios de utilizar DAC o MAC?

**23.12.** ¿Qué es el control de acceso basado en roles? ¿De qué modo es superior a DAC y MAC?

**23.13.** ¿Qué es una base de datos estadística? Comente los problemas de seguridad en las bases de datos estadísticas.

**23.14.** ¿Qué relación tiene la privacidad con la seguridad en bases de datos estadísticas? ¿Qué medidas se pueden tomar para asegurar cierto grado de privacidad en las bases de datos estadísticas?

**23.15.** ¿Qué es el control de flujo como medida de seguridad? ¿Qué tipos de control de flujo existen?

**23.16.** ¿Qué son los canales ocultos? Ponga un ejemplo de canal oculto.

**23.17.** ¿Cuál es el objetivo del cifrado? ¿Qué proceso está involucrado en el cifrado de los datos y su posterior recuperación en el otro extremo?

**23.18.** Proporcione un ejemplo de un algoritmo de cifrado y explique cómo funciona.

**23.19.** Repita la pregunta anterior para el conocido algoritmo RSA.

**23.20.** ¿Qué es el esquema de infraestructura de clave pública? ¿Cómo proporciona seguridad?

**23.21.** ¿Qué son las firmas digitales? ¿Cómo funcionan?

## Ejercicios

**23.22.** ¿Cómo se puede mantener la privacidad de los datos en una base de datos?

**23.23.** ¿Cuáles son los mayores retos actuales para la seguridad en las bases de datos?

**23.24.** Observe el esquema de base de datos relacional de la Figura 5.5. Suponga que todas las relaciones fueron creadas por (y, por tanto, propiedad de) el usuario *X*, que desea conceder los siguientes privilegios a las cuentas de usuario *A*, *B*, *C*, *D* y *E*:

**a.** La cuenta *A* puede modificar o acceder a cualquier relación excepto SUBORDINADO y puede otorgar cualquiera de estos privilegios a otros usuarios.

**b.** La cuenta *B* puede acceder a todos los atributos de EMPLEADO y DEPARTAMENTO excepto a Sueldo, DniDirector, y FechaIngresoDirector.

**c.** La cuenta *C* puede modificar o acceder a TRABAJA_EN, pero sólo puede acceder a los atributos Nombre, Apellido1, Apellido2 y Dni de EMPLEADO y a los atributos NombreProyecto y NumProyecto de PROYECTO.

**d.** La cuenta *D* puede acceder a cualquier atributo de EMPLEADO o SUBORDINADO y puede modificar SUBORDINADO.

**e.** La cuenta *E* puede acceder a cualquier atributo de EMPLEADO, pero sólo a las tuplas de EMPLEADO que tengan Dno = 3.

**f.** Escriba sentencias SQL para conceder estos privilegios. Utilice vistas en donde sea adecuado.

**23.25.** Supongamos que se va a dar el privilegio (a) del Ejercicio 23.1 con la opción GRANT OPTION pero sólo para que la cuenta *A* pueda concederlo a, como mucho, cinco cuentas, y cada una de estas cuentas podrá propagar el privilegio a otras cuentas pero *sin* el privilegio GRANT OPTION. ¿Cuáles serían los límites de propagación horizontal y vertical en este caso?

**23.26.** Observe la relación que aparece en la Figura 23.2(d). ¿Cómo se mostraría a un usuario con clasificación U? Supongamos que un usuario de clasificación U intenta actualizar el salario de 'Gómez' a 50.000; ¿cuál sería el resultado de esta acción?

## Bibliografía seleccionada

La autorización basada en la concesión y revocación de privilegios se propuso en el DBMS experimental SYSTEM R y se muestra en Griffiths y Wade (1976). Varios libros revisan la seguridad en bases de datos y en sistemas de computadores en general, entre los cuales se encuentran los libros de Leiss (1982a) y Fernández y otros (1981).

En muchos documentos se revisan distintas técnicas para el diseño y la protección de las bases de datos estadísticas. Entre ellos podemos citar McLeish (1989), Chin y Ozsoyoglu (1981), Leiss (1982), Wong (1984), y Denning (1980). Ghosh (1984) revisa el uso de bases de datos estadísticas para el control de la calidad. También hay muchos documentos en los que se discute la criptografía y el cifrado de datos, entre los que están Diffie y Hellman (1979), Rivest y otros (1978), Akl (1983), Pfleeger (1997), Omura y otros (1990), Stalling (2000), y Iyer y otros (2004).

La seguridad multinivel se repasa en Jajodia y Sandhu (1991), Denning y otros (1987), Smith y Winslett (1992), Stachour y Thuraisingham (1990), Lunt y otros (1990), y Bertino y otros (2001). Una visión general de temas de investigación en seguridad en bases de datos se muestra en Lunt y Fernández (1990), Jajodia y Sandhu (1991), Bertino y otros (1998), Castano y otros (1995), y Thuraisingham y otros (2001). Los efectos de la seguridad multinivel en el control de la concurrencia se muestran en Atluri y otros (1997). La seguridad en las bases de datos de la siguiente generación, la semántica y las bases de datos orientadas a objetos aparece en Rabbiti y otros (1991), Jajodia y Kogan (1990), y Smith (1990). Oh (1999) presenta un modelo para la seguridad discrecional y obligatoria. Los modelos de seguridad para las aplicaciones web y el control de acceso basado en roles se muestra en Joshi y otros (2001). Temas de seguridad para gestores en el contexto de las aplicaciones de comercio electrónico y la necesidad de modelos de evaluación de riesgos para la selección de medidas de control de seguridad adecuadas aparecen en Farahmand y otros (2005).

Los avances más recientes, así como los desafíos futuros en seguridad y privacidad de bases de datos, se discuten en Bertino y Sandhu (2005). El lenguaje XML y el control de accesos se revisan en Naedele (2003). Se puede encontrar más detalles sobre técnicas de mantenimiento de la privacidad en Vaidya y Clifton (2005), derechos de propiedad intelectual en Sion y otros (2004), y supervivencia de bases de datos en Jajodia y otros (1999).

CAPÍTULO 24

# Modelos de datos mejorados para aplicaciones avanzadas

A medida que crece el uso de sistemas de bases de datos, los usuarios demandan funcionalidades adicionales de los paquetes de software, con el propósito de facilitar la implementación de aplicaciones de usuario más avanzadas y complejas. Las bases de datos orientadas a objetos y los sistemas objeto-relacional proporcionan características que permiten a los usuarios extender sus sistemas especificando tipos de datos abstractos adicionales para cada aplicación. Sin embargo, es muy útil identificar ciertas características comunes para algunas de estas aplicaciones avanzadas y para crear modelos que puedan representarlas. Además, es posible implementar estructuras de almacenamiento especializadas y métodos de indexación para mejorar el rendimiento de dichas características comunes. Después, las características se pueden implementar como tipos de datos abstractos o librerías de clases, y se pueden adquirir de forma separada con el paquete de software DBMS básico. Hemos utilizado el término **data blade** (pala de datos) en Informix y *cartridge* (**cartucho**) en Oracle (consulte el Capítulo 22) para referirnos a estos submódulos opcionales que pueden incluirse en un paquete DBMS. Los usuarios pueden utilizar directamente estas características si son adecuadas para sus aplicaciones, sin tener que reinventar, reimplementar y reprogramar dichas características.

Este capítulo introduce conceptos de bases de datos para algunas de las características comunes que las aplicaciones avanzadas necesitan y que empiezan a difundirse. Hablaremos de las *reglas activas*, que se utilizan en las aplicaciones de bases de datos activas, los *conceptos temporales*, que se utilizan en las aplicaciones de bases de datos temporales o de tiempos, y, brevemente, de algunos de los problemas que surgen con las *bases de datos multimedia*. También hablaremos de las *bases de datos deductivas*. Es importante saber que cada uno de estos temas es muy amplio, y aquí sólo ofrecemos una breve introducción de cada uno. De hecho, cada una de estas áreas puede servir por sí sola como tema para un libro entero.

En la Sección 24.1 hacemos una introducción de las bases de datos activas, que proporcionan funcionalidad adicional para especificar **reglas activas**. Estas reglas pueden ser activadas automáticamente por los eventos que se producen, como las actualizaciones de la base de datos, y pueden iniciar ciertas acciones que se han especificado en la declaración de la regla para que ocurran si se dan ciertas condiciones. Muchos paquetes comerciales incluyen parte de la funcionalidad que ofrecen las bases de datos activas, en forma de ***triggers*** o disparadores. Los *triggers* forman ahora parte del estándar SQL-99.

En la Sección 24.2 introducimos el concepto de **base de datos temporal**, que permite al sistema de bases de datos almacenar un histórico de cambios, de modo que el usuario puede consultar tanto un estado pasado

como el estado actual de la base de datos. Algunos modelos de bases de datos temporales permiten almacenar la información futura que se espera, como las agendas planificadas. Es importante saber que muchas aplicaciones de bases de datos son temporales, pero a menudo se implementan sin tener mucho soporte temporal por parte del paquete DBMS; es decir, los conceptos temporales se implementan en los programas de aplicación que acceden a la base de datos.

La Sección 24.3 ofrece una breve visión general de las bases de datos espaciales y multimedia. Las **bases de datos espaciales** suministran conceptos a modo de seguimiento de los objetos en un espacio multidimensional. Por ejemplo, las bases de datos cartográficas (almacenamiento de mapas) incluyen el posicionamiento bidimensional de sus objetos: países, estados, ríos, ciudades, carreteras, mares, etcétera. Otras bases de datos, como las meteorológicas, son tridimensionales, puesto que las temperaturas y otras informaciones meteorológicas están relacionadas con puntos espaciales tridimensionales. Las **bases de datos multimedia** ofrecen características que permiten almacenar y consultar diferentes tipos de información multimedia, como **imágenes** (fotografías o dibujos), **clips de vídeo** (películas, o vídeos caseros), **clips de audio** (sonido, mensajes telefónicos, o discursos) y **documentos** (libros o artículos).

En la Sección 24.4 explicamos las bases de datos deductivas,[1] un área que se encuentra en la intersección entre las bases de datos, la lógica y la inteligencia artificial o bases del conocimiento. Un **sistema de base de datos deductivo** incluye la posibilidad de definir **reglas (deductivas)**, que pueden deducir o inferir información adicional a partir de los hechos almacenados en una base de datos. Como parte de los fundamentos teóricos de algunos sistemas de bases de datos deductivos es lógica matemática, dichas reglas se conocen a veces como **bases de datos lógicas**. Otros tipos de sistemas, denominados **sistemas de bases de datos expertos** o **sistemas basados en el conocimiento**, también incorporan capacidades de razonamiento e inferencia; dichos sistemas utilizan técnicas que se desarrollaron en el campo de la inteligencia artificial, incluyendo redes semánticas, marcos, sistemas de producción o reglas para capturar el conocimiento específico de un dominio.

El lector puede estudiar atentamente los temas particulares de su interés, ya que las secciones de este capítulo son prácticamente independientes unas de otras.

# 24.1 Conceptos de bases de datos activas y *triggers*

Durante algún tiempo, se han considerado mejoras importantes de los sistemas de bases de datos las reglas que especifican acciones que se activan automáticamente a causa de ciertos eventos. De hecho, el concepto de ***triggers*** (una técnica para especificar determinados tipos de reglas activas) existía en versiones anteriores de la especificación SQL para las bases de datos relacionales, y los *triggers* forman ahora parte del estándar SQL-99. Los DBMSs relacionales comerciales (como Oracle, DB2 y SYBASE) tienen disponibles determinadas versiones de *triggers*. Sin embargo, parece haberse investigado mucho respecto al modelo general para las bases de datos activas desde que se propusieran los primeros modelos de *triggers*. En la Sección 24.1.1 presentaremos los conceptos generales que se han propuesto para especificar reglas para las bases de datos activas. Utilizaremos la sintaxis del DBMS relacional comercial Oracle para ilustrar estos conceptos con ejemplos específicos, ya que los *triggers* de Oracle se acercan a cómo se especifican las reglas en el estándar SQL. En la Sección 24.1.2 explicaremos algunos temas generales de diseño e implementación para las bases de datos activas. En la Sección 24.1.3 ofreceremos ejemplos de cómo dichas bases de datos se implementan en el DBMS experimental STARBURST, puesto que éste proporciona dentro de su estructura muchos de los conceptos de las bases de datos activas generales. En la Sección 24.1.4 explicamos las posibles aplicaciones de las bases de datos activas. Por último, en la Sección 24.1.5 describimos cómo se declaran los *triggers* en el estándar SQL-99.

[1] La Sección 24.4 es un resumen del Capítulo 25, Bases de datos deductivas, de la tercera edición. El capítulo completo está disponible en el sitio web del libro.

### 24.1.1 Modelo generalizado para las bases de datos activas y los *triggers* de Oracle

El modelo que se ha utilizado para especificar las reglas de las bases de datos activas se conoce como **evento-condición-acción**, o modelo **ECA**. Una regla en el modelo ECA tiene tres componentes:

1. El (los) **evento(s)** que provoca(n) la acción: normalmente, estos eventos son las operaciones de modificación de la base de datos que se aplican explícitamente a la misma. No obstante, en el modelo general, podrían ser también eventos temporales[2] u otros tipos de eventos externos.
2. La **condición** que determina si debe ejecutarse la acción de la regla: una vez que se ha producido el evento de activación, puede evaluarse una condición *opcional*. Si no se especifica una condición, la acción se ejecutará una vez que se produce el evento. Si se especifica una condición, primero se evalúa ésta y, sólo si es *verdadera*, se ejecutará la acción de la regla.
3. La **acción** que se tomará: la acción es normalmente una secuencia de sentencias SQL, pero también puede ser una transacción de base de datos o un programa externo que se ejecutará automáticamente.

A fin de ilustrar estos conceptos, vamos a considerar algunos ejemplos. Los ejemplos están basados en una variante muy simplificada de la base de datos EMPRESA que vimos en la Figura 5.7 (véase la Figura 24.1), según la cual cada empleado tiene un nombre (Nombre), un número de documento nacional de identidad (Dni), un salario (Sueldo), el departamento al que está actualmente asignado (Dno, una clave externa a DEPARTAMENTO) y un supervisor directo (SuperDni, una clave externa [recursiva] a EMPLEADO). Para este ejemplo, asumimos que para Dno está permitido el valor NULL, lo que representaría un empleado que actualmente no está asignado a ningún departamento. Cada departamento tiene un nombre (NombreDpto), un número (Dno), el salario total de todos los empleados asignados al departamento (SueldoTotal) y un director (DniDirector, una clave externa a EMPLEADO).

El atributo SueldoTotal es realmente un atributo derivado, cuyo valor debe ser la suma de los sueldos de todos los empleados que están asignados a un departamento específico. El mantenimiento del valor correcto de dicho atributo derivado se puede lograr mediante una regla activa. Primero tenemos que determinar los **eventos** que pueden causar un cambio en el valor de SueldoTotal, que son los siguientes:

1. Inserción (uno o más) de nuevas tuplas de empleado.
2. Modificación del sueldo de los empleados existentes (uno o más).
3. Modificación de la asignación de los empleados existentes de un departamento a otro.
4. Eliminación de tuplas de empleado (una o más).

En el caso del evento 1, sólo tenemos que recalcular SueldoTotal si el empleado nuevo es asignado inmediatamente a un departamento; es decir, si el valor del atributo Dno para el empleado nuevo no es NULL (asumiendo que para Dno permitimos NULL). Por tanto, ésta sería la **condición** que tendríamos que comprobar.

**Figura 24.1.** Una base de datos EMPRESA simplificada que utilizaremos para los ejemplos de reglas.

**EMPLEADO**

| Nombre | Dni | Sueldo | Dno | SuperDni |
|---|---|---|---|---|

**DEPARTAMENTO**

| NombreDpto | Dno | SueldoTotal | DniDirector |
|---|---|---|---|

[2] Un ejemplo puede ser un evento temporal especificado como una hora periódica: por ejemplo, activar esta regla cada día a las 5:30 A.M.

**Figura 24.2.** Especificación de reglas activas como *triggers* en la notación de Oracle. (a) *Triggers* para el mantenimiento automático de la coherencia de SueldoTotal de DEPARTAMENTO. (b) *Trigger* para comparar el sueldo de un empleado con el de su supervisor.

**a.** **R1:**
```
CREATE TRIGGER SueldoTotal1
AFTER INSERT ON EMPLEADO
FOR EACH ROW
WHEN ( NEW.Dno IS NOT NULL )
    UPDATE DEPARTAMENTO
    SET SueldoTotal = SueldoTotal + NEW.Sueldo
    WHERE Dno = NEW.Dno;
```

**R2:**
```
CREATE TRIGGER SueldoTotal2
AFTER UPDATE OF Sueldo ON EMPLEADO
FOR EACH ROW
WHEN ( NEW.Dno IS NOT NULL )
    UPDATE DEPARTAMENTO
    SET SueldoTotal = SueldoTotal + NEW.Sueldo – OLD.Sueldo
    WHERE Dno = NEW.Dno;
```

**R3:**
```
CREATE TRIGGER SueldoTotal3
AFTER UPDATE OF Dno ON EMPLEADO
FOR EACH ROW
    BEGIN
    UPDATE DEPARTAMENTO
    SET SueldoTotal = SueldoTotal + NEW.Sueldo
    WHERE Dno = NEW.Dno;
    UPDATE DEPARTAMENTO
    SET SueldoTotal = SueldoTotal – OLD.Sueldo
    WHERE Dno = OLD.Dno;
    END;
```

**R4:**
```
CREATE TRIGGER SueldoTotal4
AFTER DELETE ON EMPLEADO
FOR EACH ROW
WHEN ( OLD.Dno IS NOT NULL )
    UPDATE DEPARTAMENTO
    SET SueldoTotal = SueldoTotal – OLD.Sueldo
    WHERE Dno = OLD.Dno;
```

**b.** **R5:**
```
CREATE TRIGGER Inform_supervisor1
BEFORE INSERT OR UPDATE OF Sueldo, SuperDni
        ON EMPLEADO
FOR EACH ROW
WHEN ( NEW.Sueldo > (       SELECT Sueldo FROM EMPLEADO
                            WHERE Dni = NEW.SuperDni ) )
        inform_supervisor(NEW.SuperDni, NEW.Dni );
```

Para el evento 2 (y 4) podríamos evaluar una condición parecida, a fin de determinar si el empleado cuyo sueldo ha cambiado (o empleado que hemos eliminado) está actualmente asignado a un departamento. En el caso del evento 3, siempre tenemos que ejecutar una acción para mantener el valor correcto de SueldoTotal, por lo que no necesitamos condición alguna (la acción siempre se ejecuta).

La **acción** para los eventos 1, 2 y 4 es para actualizar automáticamente el valor de SueldoTotal para el departamento del empleado, a fin de reflejar el sueldo del empleado recién insertado, actualizado o eliminado. En el caso del evento 3, necesitamos una acción doble; una para actualizar el SueldoTotal del departamento antiguo del empleado y otra para actualizar el SueldoTotal del departamento nuevo del empleado.

Las cuatro reglas activas (o *triggers*) R1, R2, R3 y R4 (correspondientes a la situación anterior) se pueden especificar con la notación del DBMS Oracle como se muestra en la Figura 24.2(a). Vamos a considerar la regla R1 para ilustrar la sintaxis de creación de *triggers* en Oracle.

La sentencia CREATE TRIGGER especifica un nombre de *trigger* (o regla activa): SueldoTotal1 para R1. La cláusula AFTER especifica que la regla se activará *después* de que se produzcan los eventos que activan la regla. Los eventos de activación (la inserción de un empleado nuevo en este ejemplo) se especifican a continuación de la palabra clave AFTER.[3]

La cláusula ON especifica la relación en la que la regla está especificada: EMPLEADO para R1. Las palabras clave *opcionales* FOR EACH ROW especifican que la regla se activará *una vez por cada fila* que se vea afectada por el evento de activación.[4]

La cláusula *opcional* WHEN se utiliza para especificar las condiciones que sea preciso evaluar después de haberse activado la regla, pero antes de que la acción se ejecute. Por último, la o las acciones que deban tomarse se especifican como un bloque PL/SQL, que normalmente contiene una o más sentencias SQL o llamadas para ejecutar procedimientos externos.

Los cuatro *triggers* (reglas activas) R1, R2, R3 y R4 ilustran varias características de las reglas activas. En primer lugar, los **eventos** básicos que pueden especificarse para activar las reglas son comandos de actualización SQL estándar: insertar, eliminar y actualizar, que se especifican mediante las palabras clave **INSERT**, **DELETE** y **UPDATE** en la notación de Oracle. En este caso de actualización (**UPDATE**) podemos especificar los atributos que van a actualizarse: por ejemplo, escribiendo **UPDATE OF** Sueldo, Dno. En segundo lugar, el diseñador de la regla debe poder referirse a las tuplas que el evento de activación ha insertado, eliminado o modificado. En la notación de Oracle también se utilizan las palabras clave **NEW** y **OLD**; NEW se utiliza para hacer referencia a una tupla recién insertada o actualizada, mientras que OLD se utiliza para hacer referencia a una tupla eliminada o a una tupla antes de ser actualizada.

Por tanto, la regla R1 se activa después de haberse aplicado una operación INSERT a la relación EMPLEADO. En R1 se comprueba la condición (**NEW**.Dno **IS NOT NULL**), y si se evalúa como verdadera, significa que el empleado recién insertado está relacionado con un departamento, por lo que se ejecuta la acción. La acción actualiza la tupla o tuplas DEPARTAMENTO relacionadas con el empleado recién insertado añadiendo su sueldo (**NEW**.Sueldo) al atributo SueldoTotal de su departamento relacionado.

La regla R2 es parecida a la R1, pero es activada por una operación UPDATE que actualiza el SUELDO de un empleado, y no por una operación INSERT. La regla R3 es activada por una actualización del atributo Dno de EMPLEADO, que significa cambiar la asignación de un empleado de un departamento a otro. En R3 no hay una condición, por lo que la acción se ejecuta siempre que se produce el evento de activación. La acción actualiza tanto el departamento antiguo como el nuevo de los empleados reasignados, añadiendo su sueldo al SueldoTotal de su *nuevo* departamento y sustrayendo su sueldo del SueldoTotal de su departamento *antiguo*.

---

[3] Como veremos, también es posible especificar BEFORE en lugar de AFTER, que indica que la regla se activa *antes de que se ejecute el evento de activación*.

[4] Una vez más, veremos que una alternativa es activar la regla *sólo una vez* aun cuando varias filas (tuplas) se vean afectadas por el evento de activación.

Esto debe funcionar aun en el caso de que el valor de Dno fuera NULL, porque en este caso ningún departamento se seleccionará para la acción de la regla.[5]

Es importante observar el efecto de la cláusula FOR EACH ROW opcional, que significa que la regla se activa separadamente *para cada tupla*. Es lo que se conoce como ***trigger* a nivel de fila**. Si omitimos esta cláusula, el *trigger* se denominaría ***trigger* a nivel de sentencia** y se activaría una vez por cada sentencia de activación. Para ver la diferencia, consideremos la siguiente operación de actualización, que aumenta un 10% el sueldo de todos los empleados asignados al departamento 5. Esta operación sería un evento que activaría la regla R2:

```
UPDATE    EMPLEADO
SET       Sueldo = 1.1 * Sueldo
WHERE     Dno = 5;
```

Como la sentencia anterior podría actualizar varios registros, se activaría una regla que utiliza la semántica a nivel de fila, como R2 en la Figura 24.2, *por cada fila*, mientras que una regla que utiliza la semántica a nivel de sentencia *sólo se activaría una vez*. El sistema Oracle permite elegir cuál de las opciones anteriores vamos a utilizar para cada regla. Con la cláusula opcional FOR EACH ROW creamos un *trigger* a nivel de fila, y si la omitimos creamos un *trigger* a nivel de sentencia. Las palabras clave NEW y OLD sólo se pueden utilizar con *triggers* a nivel de fila.

Como segundo ejemplo, suponga que queremos comprobar si el sueldo de un empleado es mayor que el de su supervisor(a) directo(a). Varios eventos pueden activar esta regla: la inserción de un empleado nuevo, la modificación del sueldo de un empleado, o la modificación del supervisor de un empleado. Suponga que la acción a tomar fuera llamar a un procedimiento externo inform_supervisor,[6] que notificará al supervisor. La regla podría escribirse como R5 (véase la Figura 24.2[b]).

La Figura 24.3 muestra la sintaxis para especificar algunas de las opciones principales disponibles en los *triggers* de Oracle. En la sección 24.1.5 describiremos la sintaxis de los *triggers* en el estándar SQL-99.

## 24.1.2 Diseño e implementación de temas para bases de datos activas

La sección anterior ofrecía una panorámica de algunos de los conceptos principales para especificar las reglas activas. En esta sección, explicamos algunos temas relativos al diseño y la implementación de reglas. El

**Figura 24.3.** Resumen de la sintaxis para especificar *triggers* en el sistema Oracle (sólo opciones principales).

```
<trigger>                 ::= CREATE TRIGGER <nombre del trigger>
                              ( AFTER | BEFORE ) <eventos de activación> ON <nombre de la tabla>
                              [ FOR EACH ROW ]
                              [ WHEN <condición> ]
                              <acciones del trigger> ;
<eventos de activación>    ::= <evento trigger> {OR <evento trigger> }
<evento trigger>          ::= INSERT | DELETE | UPDATE [ OF <nombre columna> { ,
                              <nombre columna> } ]
<acción trigger>          ::= <bloque PL/SQL>
```

[5] R1, R2 y R4 también se pueden escribir sin una condición. Sin embargo, será más eficaz ejecutarlas con la condición, pues la acción no se invoca a menos que sea necesaria.

[6] Asumiendo que se ha declarado un procedimiento externo apropiado. Es una característica que ahora está disponible en SQL.

primer tema atañe a la activación, desactivación y agrupamiento de reglas. Además de crear reglas, un sistema de base de datos activa debe permitir a los usuarios *activar*, *desactivar* y *eliminar* reglas. Una **regla desactivada** no será activada por el evento de activación. Esta característica permite a los usuarios desactivar selectivamente las reglas durante determinados periodos de tiempo cuando no son necesarias. El **comando de activación** (*activate*) hará que la regla se active de nuevo. El **comando de eliminación** (*drop*) elimina la regla del sistema. Otra opción es agrupar reglas en **conjuntos de reglas** con nombre, para que sea posible activar, desactivar o eliminar un conjunto de reglas. También es útil tener un comando que pueda activar una regla o un conjunto de reglas a través de un comando **PROCESS RULES** explícito emitido por el usuario.

Un segundo tema tiene que ver con si la acción activada debe ejecutarse *antes*, *después* o *a la vez* que el evento de activación. Un asunto relacionado es si la acción que está ejecutándose debe ser considerada como una *transacción independiente* o si forma parte de la misma transacción que activó la regla. Intentaremos clasificar las distintas opciones. Es importante saber que no todas las opciones estarán disponibles en un sistema de base de datos activa particular. De hecho, la mayoría de los sistemas comerciales están *limitados a una o dos de las opciones* que ahora explicaremos.

Vamos a asumir que el evento de activación ocurre como parte de la ejecución de una transacción. Primero debemos considerar las distintas opciones de cómo el evento de activación está relacionado con la evaluación de la condición de la regla. La *evaluación de la condición* de la regla también se denomina **consideración de la regla**, porque la acción sólo se ejecutará después de considerar si la condición se evaluará como verdadera o como falsa. Hay tres posibilidades principales en cuanto a la consideración de la regla:

1. **Consideración inmediata.** La condición se evalúa como parte de la misma transacción que el evento de activación, y se evalúa *inmediatamente*. Este caso se puede clasificar, a su vez, en tres opciones:
   - ☐ Evaluar la condición *antes* de ejecutar el evento de activación.
   - ☐ Evaluar la condición *después* de ejecutar el evento de activación.
   - ☐ Evaluar la condición *en lugar de* ejecutar el evento de activación.
2. **Consideración diferida.** La condición se evalúa al término de la transacción que incluye el evento de activación. En este caso, podría haber muchas reglas activadas esperando a la evaluación de sus condiciones.
3. **Consideración aislada o separada.** La condición se evalúa como una transacción separada, independientemente de la transacción de activación.

El siguiente conjunto de opciones tienen que ver con la relación entre la evaluación de la condición de la regla y la *ejecución* de la acción de la regla. Aquí, una vez más, hay tres opciones posibles: **inmediata**, **diferida** y **aislada**. La mayoría de los sistemas utilizan la primera opción. Es decir, tan pronto como se evalúa la condición, si devuelve verdadero, la acción se ejecuta *inmediatamente*.

El sistema Oracle (véase la Sección 24.1.1) utiliza el modelo de *consideración inmediata*, pero le permite al usuario especificar para cada regla si desea utilizar la opción *antes* o *después* con la evaluación inmediata de la condición. También utiliza el modelo de *ejecución inmediata*. El sistema STARBURST (consulte la Sección 24.1.3) utiliza la opción de *consideración diferida*, de modo que todas las reglas activadas por una transacción esperan hasta que la transacción de activación alcanza su final y emite su comando COMMIT WORK antes de que se evalúen las condiciones de la regla.[7]

Otro tema relacionado con las reglas de las bases de datos activas es la distinción entre las *reglas a nivel de fila* y las *reglas a nivel de sentencia*. Como las sentencias de actualización SQL (que actúan como eventos de activación) pueden especificar un conjunto de tuplas, debemos distinguir entre si la regla debe considerarse una vez para la *sentencia completa* o si debe considerarse de forma separada *para cada fila* (es decir, cada

[7] STARBURST también permite al usuario iniciar la consideración de la regla explícitamente a través de un comando PROCESS RULES.

**Figura 24.4.** Ejemplo para ilustrar el problema de la terminación de las reglas activas.

```
R1:  CREATE TRIGGER T1
     AFTER INSERT ON TABLA1
     FOR EACH ROW
          UPDATE TABLA2
          SET Atributo1 = . . . ;

R2:  CREATE TRIGGER T2
     AFTER UPDATE OF Atributo1 ON TABLA2
     FOR EACH ROW
          INSERT INTO TABLA1 VALUES ( . . . ) ;
```

tupla) afectada por la sentencia. El estándar SQL-99 (véase la Sección 24.1.5) y el sistema Oracle (consulte la Sección 24.1.1) permiten elegir la opción que vamos a utilizar para cada regla, mientras que STARBURST sólo utiliza la semántica a nivel de sentencia. En la Sección 24.1.3 ofreceremos ejemplos de cómo pueden especificarse *triggers* a nivel de sentencia.

Una de las dificultades que pueden limitar el uso generalizado de las reglas activas, a pesar de su potencial para simplificar las bases de datos y el desarrollo de software, es que no hay técnicas fáciles de utilizar para diseñar, escribir y verificar reglas. Por ejemplo, es muy difícil comprobar que un conjunto de reglas es coherente, es decir, que dos o más reglas del conjunto no se contradicen entre sí. También es complejo garantizar la terminación de un conjunto de reglas bajo todas las circunstancias. Para ilustrar brevemente el problema de la terminación, consideremos las reglas de la Figura 24.4. Aquí, la regla R1 es activada por un evento INSERT en TABLA1 y su acción incluye un evento de actualización sobre Atributo1 de TABLA2. Sin embargo, el evento de activación de R2 es un evento UPDATE sobre Atributo1 de TABLA2, y su acción incluye un evento INSERT en TABLA1. En este ejemplo, es fácil ver que estas dos reglas se pueden activar entre sí indefinidamente, no terminándose nunca. Sin embargo, si se escriben docenas de reglas, es muy difícil determinar si la terminación está o no garantizada.

Para alcanzar el potencial de las reglas activas, es necesario desarrollar herramientas para diseñar, depurar y monitorizar reglas activas que pueda ayudar a los usuarios a diseñar y depurar sus propias reglas.

### 24.1.3 Ejemplos de reglas activas a nivel de sentencia en STARBURST

Ahora vamos a ofrecer algunos ejemplos para ilustrar cómo podemos especificar reglas en el DBMS experimental STARBURST. Esto nos permitirá mostrar cómo pueden escribirse las reglas a nivel de sentencia, puesto que es el único tipo de regla que STARBURST permite.

Las tres reglas activas R1S, R2S y R3S de la Figura 24.5 se corresponden a las tres primeras reglas de la Figura 24.2, pero utilizan la notación y la semántica a nivel de sentencia de STARBURST. Con la regla R1S podemos explicar la estructura de la regla. La sentencia CREATE RULE especifica el nombre de una regla (SueldoTotal1 para R1S). La cláusula ON especifica la relación sobre la que se especifica la regla (EMPLEADO para R1S). La cláusula WHEN se utiliza para especificar los **eventos** que activan la regla.[8] La cláusula IF *opcional* se utiliza para especificar las **condiciones** que deban evaluarse. Por último, la cláusula THEN se

[8] La palabra clave WHEN especifica los *eventos* de STARBURST, pero se utiliza para especificar la *condición* de la regla en SQL y en los *triggers* de Oracle.

**Figura 24.5.** Reglas activas utilizando la semántica a nivel de sentencia en la notación de STARBURST.

```
R1S: CREATE   RULE SueldoTotal1 ON EMPLEADO
     WHEN     INSERTED
     IF       EXISTS     (SELECT * FROM INSERTED WHERE Dno IS NOT NULL )
     THEN     UPDATE     DEPARTAMENTO AS D
              SET        D.SueldoTotal = D.SueldoTotal +
                           (SELECT SUM (I.Sueldo) FROM INSERTED AS I WHERE D.Dno = I.Dno )
              WHERE      D.Dno IN ( SELECT Dno FROM INSERTED );

R2S: CREATE   RULE SueldoTotal2 ON EMPLEADO
     WHEN     UPDATED    ( Sueldo )
     IF       EXISTS     (SELECT * FROM NEW-UPDATED WHERE Dno IS NOT NULL )
              OR EXISTS  (SELECT * FROM OLD-UPDATED WHERE Dno IS NOT NULL )
     THEN     UPDATE     DEPARTAMENTO AS D
              SET        D.SueldoTotal = D.SueldoTotal +
                         (SELECT SUM (N.Sueldo) FROM NEW-UPDATED AS N
                           WHERE D.Dno = N.Dno ) –
                         (SELECT SUM (O.Sueldo) FROM OLD-UPDATED AS O
                           WHERE D.Dno = O.Dno )
              WHERE      D.Dno IN ( SELECT Dno FROM NEW-UPDATED ) OR
                         D.Dno IN ( SELECT Dno FROM OLD-UPDATED );

R3S: CREATE   RULE SueldoTotal3 ON EMPLEADO
     WHEN     UPDATED    ( Dno )
     THEN     UPDATE     DEPARTAMENTO AS D
              SET        D.SueldoTotal = D.SueldoTotal +
                         (SELECT SUM (N.Sueldo) FROM NEW-UPDATED AS N
                           WHERE D.Dno = N.Dno )
              WHERE      D.Dno IN ( SELECT Dno FROM NEW-UPDATED );

              UPDATE     DEPARTAMENTO AS D
              SET        D.SueldoTotal = SueldoTotal –
                     (   SELECT SUM (O.Sueldo) FROM OLD-UPDATED AS O
                         WHERE D.Dno = O.Dno )
              WHERE      D.Dno IN ( SELECT Dno FROM OLD-UPDATED );
```

utiliza para especificar la **acción** (o acciones) que han de tomarse, que normalmente son una o más sentencias SQL.

En STARBURST, los eventos básicos que podemos especificar para activar las reglas son los comandos de actualización SQL estándar: insertar, eliminar y actualizar, que se especifican mediante las palabras clave **INSERTED**, **DELETED** y **UPDATED** en la notación de STARBURST. En segundo lugar, el diseñador de la regla necesita una forma de referirse a las tuplas que se han modificado. Las palabras clave **INSERTED**, **DELETED**, **NEW-UPDATED** y **OLD-UPDATED** se utilizan en la notación de STARBURST para referirnos a las **tablas de transición** (relaciones) que incluyen las tuplas recién insertadas, recién eliminadas y las tuplas actualizadas *antes* de que fueran actualizadas, y las tuplas actualizadas *después* de que fueran actualizadas,

respectivamente. Obviamente, en función de los eventos de activación, sólo algunas de estas tablas de transición pueden estar disponibles. Al escribir la regla puede hacerse referencia a estas tablas en las partes de la condición y la acción de la regla. Las tablas de transición contienen tuplas del mismo tipo que las de la relación especificada en la cláusula ON de la regla (para R1S, R2S y R3S, se trata de la relación EMPLEADO).

En la semántica a nivel de la sentencia, el diseñador de la regla sólo puede referirse a las tablas de transición como un todo y la regla sólo se activa una vez, por lo que las reglas deben escribirse de forma diferente a como se hace en la semántica a nivel de fila. Como con una sola sentencia de inserción se pueden insertar varias tuplas de empleado, tenemos que comprobar si *al menos una* de las tuplas de empleado insertadas está relacionada con un departamento. En R1S, comprobamos la condición:

**EXISTS** ( **SELECT** * **FROM INSERTED WHERE** Dno **IS NOT NULL** )

y, si se evalúa como verdadera, entonces se ejecuta la acción. La acción actualiza en una sola sentencia la o las tuplas DEPARTAMENTO relacionadas con el o los empleados recién insertados añadiendo sus sueldos al atributo SueldoTotal de cada departamento relacionado. Como más de uno de los empleados recién insertados pueden pertenecer al mismo departamento, utilizamos la función agregada SUM para asegurarnos de que se han añadido todos sus sueldos.

La regla R2S es parecida a la regla R1S, pero es activada por una operación UPDATE que actualiza el sueldo de uno o más empleados, en lugar de ser activada por INSERT. La regla R3S es activada por una actualización del atributo Dno de EMPLEADO, que significa cambiar una o más asignaciones de empleados de un departamento a otro. En R3S no hay condición, por lo que la acción se ejecuta siempre que se produce el evento de activación.[9] La acción actualiza el departamento antiguo y el departamento nuevo de los empleados reasignados, añadiendo sus sueldos al SueldoTotal de cada *nuevo* departamento y sustrayendo sus sueldos del SueldoTotal de cada departamento *antiguo*.

En nuestro ejemplo, es más complicado escribir las reglas a nivel de sentencia que a nivel de fila, como puede comprobar comparando las Figuras 24.2 y 24.5. Sin embargo, no es una norma general, y otros tipos de reglas activas son más fáciles de escribir con la notación a nivel de sentencia que con la notación a nivel de fila.

El modelo de ejecución para las reglas activas en STARBURST utiliza la **consideración diferida**. Es decir, todas las reglas que se activan dentro de una transacción se colocan en un conjunto, denominado **conjunto de conflicto**, que no se tiene en consideración para la evaluación de las condiciones y la ejecución hasta el final de la transacción (emitiendo su comando COMMIT WORK). STARBURST también permite al usuario iniciar explícitamente la consideración de la regla en medio de una transacción a través de un comando PROCESS RULES explícito. Como deben evaluarse varias reglas, es preciso especificar un orden entre las reglas. La sintaxis para la declaración de una regla en STARBURST permite la especificación de la *ordenación* entre las reglas para instruir al sistema acerca del orden en que debe considerarse un conjunto de reglas.[10] Además, las tablas de transición (INSERTED, DELETED, NEW-UPDATED y OLD-UPDATED) contienen el *efecto neto* de todas las operaciones dentro de la transacción que afectaron a cada tabla, puesto que pueden haberse aplicado varias operaciones a cada tabla durante la transacción.

### 24.1.4 Aplicaciones potenciales para las bases de datos activas

Ahora explicaremos brevemente algunas de las aplicaciones potenciales de las reglas activas. Obviamente, una aplicación importante es permitir la **notificación** de ciertas condiciones que pueden darse. Por ejemplo, una base de datos activa se puede utilizar para monitorizar la temperatura de un horno industrial. La aplica-

[9] Como en los ejemplos de Oracle, las reglas R1S y R2S se pueden escribir sin una condición. Sin embargo, puede resultar más eficaz ejecutarlas con la condición, ya que la acción no se realiza a menos que sea necesaria.

[10] Si no se especifica un orden entre dos reglas, el orden predeterminado del sistema es colocar primero la regla que se declaró primero, y después colocar la siguiente.

ción puede insertar periódicamente en la base de datos la temperatura leyendo directamente los registros de los sensores de temperatura, y pueden escribirse reglas activas que se activan cuando se inserta un registro de temperatura, con una condición que comprueba si la temperatura excede el nivel de peligro, en cuyo caso se llevaría a cabo una acción para dar la alarma.

Las reglas activas también se pueden utilizar para **implementar restricciones de integridad** mediante la especificación de los tipos de eventos que pueden provocar la violación de las restricciones y la posterior evaluación de las condiciones apropiadas que comprueban si las restricciones han sido realmente violadas o no por el evento. Por tanto, de este modo pueden implementarse las restricciones de aplicación complejas, denominadas a menudo **reglas comerciales**. Por ejemplo, en la base de datos UNIVERSIDAD, una regla puede monitorizar la nota media de los estudiantes siempre que se introduzca una calificación nueva, y puede alertar al tutor si la nota media de un estudiante está por debajo de un determinado umbral; otra regla puede comprobar que los prerrequisitos de un curso son satisfechos antes de permitir que un estudiante se matricule en ese curso; etcétera.

Otras aplicaciones son el **mantenimiento de los datos derivados**, como los ejemplos de las reglas R1 a R4 que mantienen el atributo derivado SueldoTotal siempre que se modifican tuplas de empleado individuales. Una aplicación parecida es el uso de reglas activas para mantener la coherencia de las **vistas materializadas** (consulte la Sección 8.8) siempre que se modifican las relaciones de la base. Esta aplicación también es importante para las nuevas tecnologías de almacenamiento de datos (consulte el Capítulo 29). Una aplicación relacionada mantiene la coherencia de estas **tablas duplicadas** especificando reglas que modifican las réplicas siempre que se modifica la tabla maestra.

### 24.1.5 *Triggers* en SQL-99

Los *triggers* en el estándar SQL-99 son muy parecidos a los ejemplos que hemos visto en la Sección 24.1.1, con algunas diferencias sintácticas menores. Los **eventos** básicos que podemos especificar para activar las reglas son los comandos de actualización SQL estándar: INSERT, DELETE y UPDATE. En el caso de UPDATE, podemos especificar los atributos que se van a modificar. Están permitidos los *triggers* a nivel de fila y a nivel de sentencia, que se indican en el *trigger* con las cláusulas FOR EACH ROW y FOR EACH STATEMENT, respectivamente. Una diferencia sintáctica es que el *trigger* puede especificar nombres de variable de tupla particulares para las tuplas antigua y nueva, en lugar de utilizar las palabras clave NEW y OLD, como mostramos en la Figura 24.1. El *trigger* T1 de la Figura 24.6 muestra cómo podemos especificar en SQL-99 el *trigger* a nivel de fila R2 de la Figura 24.1(a). Dentro de la cláusula REFERENCING, utilizamos las variables de tupla con nombre (alias) O y N para referirnos a la tupla OLD (antes de la modificación) y a la tupla NEW (después de la modificación), respectivamente. El *trigger* T2 de la Figura 24.6 muestra cómo podemos especificar en SQL-99 el *trigger* a nivel de sentencia R2S de la Figura 24.5. En el caso de un *trigger* a nivel de sentencia, utilizamos la cláusula REFERENCING para referirnos a la tabla de todas las tuplas nuevas (recién insertadas o modificadas) como N, mientras que a la tabla de todas las tuplas antiguas (tuplas eliminadas o tuplas antes de que fueran actualizadas) nos referimos como O.

## 24.2 Conceptos de bases de datos de tiempo (temporales)

Las bases de datos temporales, en el sentido más amplio, engloban todas las aplicaciones de bases de datos que requieren algún aspecto de tiempo para organizar su información. Por tanto, suponen un buen ejemplo para ilustrar la necesidad de desarrollar un conjunto de conceptos unificado para que lo utilicen los desarrolladores de aplicaciones. Las aplicaciones de bases de datos temporales existen desde que empezaron a utilizarse las bases de datos. No obstante, al crear estas aplicaciones, normalmente se deja a los diseñadores y los desarrolladores descubrir, diseñar, programar e implementar los conceptos temporales (de tiempo) que nece-

sitan. Hay muchos ejemplos de aplicaciones en los que se necesita algún aspecto del tiempo para mantener la información en una base de datos: el *cuidado de la salud*, donde es necesario conservar los historiales de los pacientes; las *aseguradoras*, donde se necesitan los historiales de las reclamaciones y los accidentes, así como información sobre el momento en que una póliza tiene efecto; los *sistemas de reservas* en general (hoteles, aerolíneas, alquiler de automóviles, trenes, etc.), en los que se necesitan las fechas y las horas de las reservas; las *bases de datos científicas*, en las que los datos recopilados de los experimentos incluyen la hora en que se mide cada dato; etcétera. Incluso los dos ejemplos utilizados en este libro se pueden ampliar fácilmente como aplicaciones temporales. En la base de datos EMPRESA, podemos querer conservar historiales de SUELDO, TRABAJO y PROYECTO por cada empleado. En la base de datos UNIVERSIDAD, el tiempo ya se incluye en SEMESTRE y AÑO de cada NIVEL de un CURSO; el historial de anotaciones de un ESTUDIANTE; y la información sobre las becas de investigación. De hecho, es realista concluir que la mayoría de las aplicaciones de bases de datos tienen algo de información sobre tiempos. Sin embargo, los usuarios a menudo intentan simplificar o ignorar los aspectos del tiempo debido a la complejidad que suponen para sus aplicaciones.

En esta sección, introduciremos algunos de los conceptos que se han desarrollado para tratar con la complejidad de las aplicaciones de bases de datos de tiempo. La Sección 24.2.1 ofrece una visión general de cómo se representa el tiempo en las bases de datos, los diferentes tipos de información temporal y algunas de las diferentes dimensiones de tiempo que necesitamos. La Sección 24.2.2 explica cómo puede incorporarse el tiempo a las bases de de datos relacionales. La Sección 24.2.3 ofrece algunas de las opciones adicionales para representar el tiempo que son posibles con los modelos de bases de datos y que permiten objetos estructurados complejos. La Sección 24.2.4 introduce las operaciones para consultar las bases de datos temporales, y ofrece una visión general del lenguaje TSQL2, que extiende SQL con conceptos relacionados con el tiempo. La Sección 24.2.5 se centra en los datos de series de tiempo, que es un tipo de datos de tiempos muy importante en la práctica.

**Figura 24.6.** El *trigger* T1 ilustra la sintaxis para definir *triggers* en SQL-99.

```
T1:  CREATE TRIGGER SueldoTotal1
     AFTER UPDATE OF Sueldo ON EMPLEADO
     REFERENCING OLD ROW AS O, NEW ROW AS N
     FOR EACH ROW
     WHEN      ( N.Dno IS NOT NULL )
     UPDATE DEPARTAMENTO
     SET SueldoTotal = SueldoTotal + N.sueldo – O.sueldo
     WHERE Dno = N.Dno;

T2:  CREATE TRIGGER SueldoTotal2
     AFTER UPDATE OF Sueldo ON EMPLEADO
     REFERENCING OLD TABLE AS O, NEW TABLE AS N
     FOR EACH STATEMENT
     WHEN      EXISTS ( SELECT * FROM N WHERE N.Dno IS NOT NULL ) OR
               EXISTS ( SELECT * FROM O WHERE O.Dno IS NOT NULL )
     UPDATE DEPARTAMENTO AS D
     SET D.SueldoTotal = D.SueldoTotal
     + ( SELECT SUM (N.Sueldo) FROM N WHERE D.Dno=N.Dno )
     – ( SELECT SUM (O.Sueldo) FROM O WHERE D.Dno=O.Dno )
        WHERE Dno IN ( ( SELECT Dno FROM N ) UNION ( SELECT Dno FROM O ) );
```

### 24.2.1 Representación del tiempo, calendarios y dimensiones del tiempo

El tiempo, para una base de datos de tiempos, se considera como una *secuencia ordenada* de **puntos** de alguna **granularidad** determinada por la aplicación. Por ejemplo, supongamos que una aplicación nunca va a necesitar unidades de tiempo menores que el segundo. Por tanto, cada punto de tiempo representa un segundo según esta granularidad. En realidad, cada segundo es una (breve) *duración de tiempo,* no un punto, ya que puede dividirse en milisegundos, microsegundos, etc. Los investigadores de las bases de datos de tiempo han usado el término ***chronon*** en lugar de punto para describir la granularidad mínima para una aplicación particular. La consecuencia principal de elegir esta granularidad mínima (digamos, un segundo) es que los eventos que tengan lugar dentro del mismo segundo serán considerados como *eventos simultáneos,* aunque pensemos en realidad que no lo son.

Ya que no existe un comienzo y un final de tiempo conocidos, es preciso contar con una referencia desde la que medir puntos de tiempo específicos. Cada cultura ha utilizado su propio calendario (como el Gregoriano [occidental], el chino, el islámico, el hindú, el judío, el copto, etc.) con puntos de referencia diferentes. Un **calendario**, por convenio, organiza el tiempo en diferentes unidades de tiempo. La mayoría de ellos agrupan 60 segundos en un minuto, 60 minutos en una hora, 24 horas en un día (en base al tiempo físico de la rotación de la tierra sobre su eje) y 7 días en una semana. Otras agrupaciones mayores, como la de días en meses y meses en años, siguen los fenómenos lunares o solares naturales, y suelen ser irregulares. En el calendario Gregoriano, usado en la mayoría de países occidentales, los días se agrupan en meses de 28, 29, 30 ó 31 días, y se agrupan 12 meses para formar un año. Para emparejar las distintas unidades se emplean complejas fórmulas.

En SQL2, la información referente al tiempo (consulte el Capítulo 8) puede ser DATE (que especifica año, mes y día como YYYY-MM-DD), TIME (donde hora, minuto y segundo aparecen como HH:MM:SS), TIMESTAMP (o combinación fecha/hora en la que existen opciones para incluir divisiones inferiores al segundo en caso de necesitarse), INTERVAL (una duración de tiempo relativa, como 10 días o 250 minutos) y PERIOD (una duración de tiempo fija que cuenta con un punto de inicio, como los 10 días transcurridos entre el 1 y el 10 de enero de 2007, ambos inclusive).[11]

**Información de evento frente a información de duración (o estado).** Una base de datos de tiempo almacenará información referente al momento en que se producen ciertos eventos, o cuando ciertos hechos son considerados como verdaderos. Existen distintos tipos de información de tiempo. Los **eventos o hechos de punto** suelen estar asociados en la base de datos con un **punto de tiempo único** de alguna granularidad. Por ejemplo, un deposito bancario podría estar asociado con la marca de tiempo de cuándo se concretó, y las ventas totales mensuales de un producto (hecho) podrían estarlo a un mes particular (digamos, febrero de 1999). Observe que aunque eventos o hechos de este tipo podrían tener diferentes granularidades, cada uno de ellos esta asociado a un *valor de tiempo único* en la base de datos. Este tipo de información suele representarse como **datos de series de tiempos** tal y como veremos en la Sección 24.2.5. En el otro lado, los **eventos o hechos de duración** están asociados a **periodos de tiempo** específicos en la base de datos.[12] Por ejemplo, un empleado podría haber trabajado en una empresa desde el 15 de agosto de 1993 hasta el 20 de noviembre de 1998.

Un **periodo de tiempo** está identificado por sus **puntos de tiempo inicial** y **final** [START-TIME, ENDTIME]. Por ejemplo, el periodo anterior está representado como [15-08-1993, 20-11-1998]. De este modo, un perio-

[11] Por desgracia, la terminología no se utiliza de forma coherente. Por ejemplo, el término *intervalo* suele usarse con frecuencia para especificar una duración fija. Por coherencia, utilizaremos la terminología SQL.

[12] Esto es lo mismo que una duración fija. Con frecuencia se la conoce como un **intervalo de tiempo**, aunque para evitar confusiones usaremos **periodo** para mantener la coherencia con la terminología SQL.

do de tiempo suele interpretarse como el *conjunto de todos los puntos de tiempo* existentes entre su inicio y su final, ambos inclusive, en la granularidad especificada. Así pues, y asumiendo una granularidad diaria, el periodo [15-08-1993, 20-11-1998] representa el conjunto de todos los días comprendidos entre el 15 de agosto de 1993 y el 20 de noviembre de 1998, ambos inclusive.[13]

**Dimensiones de tiempo de transacción y tiempo válido.** Dado un evento o hecho particulares asociados a un punto o periodo de tiempo concretos en la base de datos, la asociación puede interpretarse de distintas maneras. La más natural de ellas es que el tiempo asociado es aquél en el que se produce el evento, o el periodo durante el que el hecho se consideró como verdadero *en el mundo real.* Si se emplea esta interpretación, el tiempo asociado suele conocerse con frecuencia como el **tiempo válido**. Una base de datos de tiempo que utiliza esta interpretación recibe el nombre de **base de datos de tiempo válido**.

Sin embargo, puede usarse una interpretación diferente en la que el tiempo asociado hace referencia al momento en el que la información se almacenó en la base de datos; esto es, es el valor del reloj del sistema cuando la información es validada *en dicho sistema.*[14] En este caso, el tiempo asociado recibe el nombre de **tiempo de transacción**, y la base de datos que usa esta interpretación, **base de datos de tiempo de transacción**.

Existen otras interpretaciones, pero las dos anteriores se consideran las más comunes, y están referidas como **dimensiones de tiempo**. En algunas aplicaciones sólo se precisa una de estas dimensiones, mientras que en otras se precisan las dos, en cuyo caso la base de datos de tiempo se llama **base de datos bitemporal**. Si se desean otras interpretaciones para el tiempo, el usuario puede definir las semánticas y programar las aplicaciones de forma adecuada, lo que se conoce como un **tiempo definido por el usuario**.

La siguiente sección muestra con ejemplos cómo pueden incorporarse estos conceptos a las bases de datos relacionales, mientras que la Sección 24.2.3 muestra un planteamiento para la incorporación de conceptos de tiempo en bases de datos de objetos.

### 24.2.2 Incorporación del tiempo a bases de datos relacionales usando el versionado de tuplas

**Relaciones de tiempo válido.** Vamos a ver ahora cómo pueden representarse los distintos tipos de bases de datos de tiempo en un modelo relacional. En primer lugar, supongamos que queremos incluir la historia de cambios tal y como ocurren en el mundo real. Consideremos de nuevo la base de datos de la Figura 24.1, y asumamos que, para esta aplicación, la granularidad es el día. Por tanto, podríamos convertir las dos relaciones EMPLEADO y DEPARTAMENTO en **relaciones de tiempo válido** añadiendo los atributos Tiv (Tiempo de inicio válido) y Tfv (Tiempo de finalización válido), cuyo tipo de datos es DATE para conseguir la granularidad de día. Todo esto puede verse en la Figura 24.7(a), en la que las relaciones han sido renombradas como EMP_TV y DEPT_TV, respectivamente.

Consideremos en qué se diferencian la relación EMP_TV de la relación EMPLEADO no temporal (véase la Figura 24.1).[15] En EMP_TV, cada tupla V representa una **versión** de la información de un empleado que es válida (en el mundo real) sólo durante el periodo de tiempo [V.Tiv, V.Tfv], mientras que en EMPLEADO, cada tupla únicamente representa el estado, o versión, actuales de cada empleado. En EMP_TV, la **versión actual**

[13] La representación [15-08-1993, 20-11-1998] se conoce como una representación de *intervalo cerrado*. Puede usarse también un *intervalo abierto*, designado como [15-08-1993, 20-11-1998), cuando el conjunto de puntos no incluye el de finalización. Aunque la última representación es a veces más conveniente, usaremos intervalos cerrados para evitar confusiones.

[14] La explicación es más compleja, tal y como veremos en la Sección 24.2.3.

[15] Una relación no temporal también recibe el nombre de **relación fija** (o ***snapshot***) porque sólo muestra la *instantánea actual*, o *estado actual*, de la base de datos.

**Figura 24.7.** Diferentes tipos de bases de datos relacionales de tiempo. (a) Esquema de base de datos de tiempo válido. (b) Esquema de base de datos de tiempo de transacción. (c) Esquema de base de datos bitemporal.

**(a)** **EMP_TV**

| Nombre | Dni | Sueldo | Dno | SuperDni | Tiv | Tfv |
|---|---|---|---|---|---|---|

**DEPT_TV**

| NombreDpto | Dno | SueldoTotal | DniDirector | Tiv | Tfv |
|---|---|---|---|---|---|

**(b)** **EMP_TT**

| Nombre | Dni | Sueldo | Dno | SuperDni | Tit | Tft |
|---|---|---|---|---|---|---|

**DEPT_TT**

| NombreDpto | Dno | SueldoTotal | DniDirector | Tit | Tft |
|---|---|---|---|---|---|

**(c)** **EMP_BT**

| Nombre | Dni | Sueldo | Dno | SuperDni | Tiv | Tfv | Tit | Tft |
|---|---|---|---|---|---|---|---|---|

**DEPT_BT**

| NombreDpto | Dno | SueldoTotal | DniDirector | Tiv | Tfv | Tit | Tft |
|---|---|---|---|---|---|---|---|

de cada empleado cuenta con un valor especial, *now,* como su tiempo de finalización válido. Este valor especial, ***now***, es una **variable de tiempo** que representa implícitamente el momento actual a medida que el tiempo progresa. La relación no temporal EMPLEADO podría incluir sólo las tuplas de EMP_TV cuyo Tfv es *now*.

La Figura 24.8 muestra algunas versiones de tupla de las relaciones de tiempo EMP_TV y DEPT_TV. Existen dos versiones de Pérez, tres de Campos, una de Torres y una de Ojeda. Ahora podemos ver cómo debería comportarse una relación de tiempo válido cuando la información cambia. Siempre que se **actualizan** uno o más atributos de un empleado, en lugar de sobrescribir los valores antiguos como ocurre con las bases de datos no temporales, el sistema debería crear una nueva versión y **cerrar** la actual cambiando su TFV al tiempo final. Así, cuando el usuario ejecuta el comando para actualizar el salario de Pérez efectivo el 1 de junio de 2003 a 30.000 euros, se creó la segunda versión de Pérez (véase la Figura 24.8). En el momento de realizar esta actualización, la primera versión de Pérez era la versión actual, con *now* como su Tfv. Pero después de realizarla, *now* se cambió a 31 de mayo de 2003 (uno menos que el 1 de junio de 2003, con una granularidad diaria), para indicar que esta información se ha convertido en una **versión cerrada**, o **histórica**, y que la nueva (segunda) versión de Pérez es ahora la actual.

Es importante indicar que, en una relación de tiempo válido, el usuario debe proporcionar el tiempo válido de una actualización. Por ejemplo, el sueldo actualizado de Pérez puede haberse introducido en la base de datos el 15 de mayo de 2003 a las 8:52:12 A.M., aunque esta modificación sería efectiva el 1 de junio de 2003. Esto se conoce como **actualización preventiva**, ya que se aplica a la base de datos *antes* de que se haga efectiva en el mundo real. Si la aplicación es posterior a la actualización, lo que tenemos es una **actualización retroactiva**, mientras que si se aplica en el mismo momento de hacerse efectiva, contamos con una **actualización simultánea**.

La acción correspondiente al **borrado** de un empleado en una base de datos no temporal podría aplicarse en una de tiempo válido *cerrando la versión actual* de ese empleado. Por ejemplo, si Pérez deja la empresa de forma efectiva el 19 de enero de 2004, esto podría aplicarse cambiando el valor de TFV de la versión actual

**Figura 24.8.** Algunas versiones de tupla en las relaciones de tiempo válido EMP_TV y DEPT_TV.

**EMP_TV**

| Nombre | Dni | Sueldo | Dno | SuperDni | Tiv | Tfv |
|---|---|---|---|---|---|---|
| Pérez | 123456789 | 25000 | 5 | 333445555 | 15-06-2002 | 31-05-2003 |
| Pérez | 123456789 | 30000 | 5 | 333445555 | 01-06-2003 | Now |
| Campos | 333445555 | 25000 | 4 | 999887777 | 20-08-1999 | 31-01-2001 |
| Campos | 333445555 | 30000 | 5 | 999887777 | 01-02-2001 | 31-03-2002 |
| Campos | 333445555 | 40000 | 5 | 888665555 | 01-04-2002 | Now |
| Torres | 222447777 | 28000 | 4 | 999887777 | 01-05-2001 | 10-08-2002 |
| Ojeda | 666884444 | 38000 | 5 | 333445555 | 01-08-2003 | Now |

. . .

**DEPT_TV**

| NombreDpto | Dno | SuperDni | Tiv | Tfv |
|---|---|---|---|---|
| Investigación | 5 | 888665555 | 20-09-2001 | 31-03-2002 |
| Investigación | 5 | 333445555 | 01-04-2002 | Now |

. . .

de Pérez desde *now* a 19-01-2004. En la Figura 24.8, no existe versión actual para López ya que, presumiblemente, dejó la empresa el 10-08-2002 y fue *borrado de forma lógica.* Sin embargo, ya que la base de datos es de tiempo, la información antigua de López aún se mantiene.

La operación de **insertar** un empleado nuevo podría corresponderse con la *creación de la primera versión de la tupla* de ese empleado y convirtiéndola en versión actual, siendo el Tiv el momento efectivo en el que ese empleado comenzó a trabajar. En la Figura 24.7, la tupla de Ojeda ilustra este caso, ya que la primera versión aún no ha sido actualizada.

Observe que en una relación de tiempo válida, la *clave no temporal,* como el Dni en EMPLEADO, no es única en cada tupla (versión). La nueva clave de relación para EMP_TV es una combinación de la clave no temporal y el atributo de tiempo de inicio válido Tiv,[16] de manera que utilizamos (Dni, Tiv) como clave primaria. Esto es así porque, en cualquier momento, debe existir *al menos una versión válida* de cada entidad. Por consiguiente, la restricción de que dos versiones de tupla cualesquiera que representan a la misma entidad deben tener *periodos de tiempo válidos que no intersecten* debe persistir en relaciones de tiempo válidas. Tenga en cuenta que si el valor de clave primaria no temporal puede cambiar a lo largo del tiempo, es importante contar con un **atributo de clave suplente** único cuyo valor nunca cambie por cada entidad del mundo real, lo que nos permite relacionar todas las versiones de dicha entidad.

Las relaciones de tiempo válidas suelen mantener la historia de los cambios a medida que se hacen efectivos en el *mundo real.* Por tanto, si se aplican todos estos cambios, la base de datos almacenará un histórico de los *estados del mundo real* representados. Sin embargo, ya que las actualizaciones, inserciones y borrados pueden aplicarse de forma retroactiva o preventiva, no existe un registro del *estado actual de la base de datos* en un momento concreto. En caso de que esta información sea relevante para la aplicación, será preciso utilizar *relaciones de tiempo de transacción.*

**Relaciones de tiempo de transacción.** En una base de datos de tiempo de transacción, siempre que se le aplica una modificación se registra la **marca de tiempo** (*timestamp*) actual de la transacción que aplicó el

[16] También puede usarse una combinación de la clave no temporal y el atributo Tfv.

cambio (inserción, borrado o actualización). Este tipo de base de datos es más útil cuando los cambios se aplican *simultáneamente* en la mayoría de los casos (por ejemplo, ventas en tiempo real o transacciones bancarias). Si convertimos la base de datos no temporal de la Figura 24.1 en otra de tiempo de transacción, las dos relaciones EMPLEADO y DEPARTMENTO se transforman en **relaciones de tiempo de transacción** añadiendo los atributos Tit (Tiempo de Inicio de Transacción) y Tft (Tiempo Final de Transacción) cuyo tipo de datos es, por lo general, TIMESTAMP. Todo esto puede verse en la Figura 24.7(b), en la que las relaciones se han renombrado como EMP_TT y DEPT_TT, respectivamente.

En EMP_TT, cada tupla $V$ representa una *versión* de la información de un empleado que fue creada en el momento actual $V$.Tit y fue eliminada (de forma lógica) en el momento $V$.Tft (porque dichos datos no eran correctos). En EMP_TT, la *versión actual* de cada empleado tiene un valor especial ***uc*** **(Hasta que se cambie, *Until Changed*)** como su tiempo de final de transacción, el cual indica que la tupla representa la información correcta *hasta que sea modificada* por alguna otra transacción.[17] Una base de datos de tiempo de transacción recibe también el nombre de **base de datos *rollback***,[18] ya que un usuario puede, de forma lógica, volver atrás a cualquier momento en el tiempo $T$ recuperando todas las versiones de la tupla $V$ cuyo período de tiempo de transacción [$V$.Tit, $V$.Tft] incluya el momento $T$.

**Relaciones bitemporales.** Algunas aplicaciones precisan tanto del tiempo válido como del tiempo de transacción, lo que nos conduce a las **relaciones bitemporales**. En nuestro ejemplo, la Figura 24.7(c) muestra cómo serían las relaciones no temporales EMPLEADO y DEPARTAMENTO de la Figura 24.1 al expresarlas como relaciones bitemporales EMP_BT y DEPT_BT, respectivamente. La Figura 24.9 contiene algunas tuplas de estas relaciones. En estas tablas, las tuplas cuyo Tft es *uc* son las únicas que contienen la información correcta, mientras que aquéllas cuyo Tft es una marca de tiempo absoluta son tuplas que eran válidas hasta (justamente) esa marca de tiempo. Por tanto, las tuplas con *uc* de la Figura 24.9 se corresponden con las tuplas de tiempo válido de la Figura 24.7. El atributo Tit de cada tupla es la marca de tiempo de la transacción que la creó.

Ahora vamos a considerar el modo de implementar una **operación de actualización** en una relación bitemporal. En este modelo de bases de datos,[19] *ningún atributo de ninguna tupla se modifica de forma física* excepto el Tft, que almacenará *uc*.[20] Para ilustrar cómo se crearon las tuplas, consideremos la relación EMP_BT. La *versión actual* $V$ de un empleado tiene *uc* en su atributo Tft y *now* en Tfv. Si se modifica algún otro atributo (por ejemplo, Sueldo), la transacción $T$ que lleva a cabo esa modificación debe contar con dos parámetros: el nuevo Sueldo y el tiempo válido VT en el que dicho salario se hará efectivo (en el mundo real). Asumiendo que VT- es el momento justo anterior a VT en la granularidad dada y que la transacción $T$ tiene una marca de tiempo TS($T$), los siguientes son los cambios físicos que se aplicarán a la tabla EMP_BT:

1. Realizar una copia $V_2$ de la versión $V$ actual; almacenar en $V_2$.Tfv el valor VT-, TS($T$) en $V_2$.Tit, *uc* en $V_2$.Tft e insertar $V_2$ en EMP_BT; $V_2$ es una copia de la versión anterior de *V después de cerrarse* en el tiempo válido VT−.
2. Realizar una copia $V_3$ de la versión actual de $V$; almacenar en $V_3$.Tiv el valor VT, *now* en $V_3$.Tfv, el nuevo salario en $V_3$.Sueldo, TS($T$) en $V_3$.Tit, *uc* en $V_3$.Tft e insertar $V_3$ en EMP_BT; $V_3$ es la nueva versión actual.

---

[17] La variable *uc* en relaciones de tiempo de transacción se corresponde con la variable *now* en las de tiempo válido, aunque las semánticas son ligeramente diferentes.

[18] Aquí, el término *rollback* no es lo mismo que una *transacción rollback o de anulación* (consulte el Capítulo 19) durante una recuperación en la que las actualizaciones son *canceladas físicamente*. En nuestro caso, las actualizaciones pueden ser *canceladas de forma lógica*, permitiendo que el usuario examine la base de datos en un momento anterior.

[19] Se han propuesto muchos modelos de bases de datos de tiempo. Aquí se describen modelos específicos como ejemplos para ilustrar los conceptos.

[20] Algunos modelos bitemporales permiten cambiar también el atributo Tfv, aunque la interpretación de las tuplas en estos modelos son diferentes.

**3.** Almacenar en $V$.Tft el valor TS($T$), ya que la versión actual nunca contendrá la información correcta.

Para ilustrar todo este proceso, consideremos las tres primeras tuplas $V_1$, $V_2$ y $V_3$ de EMP_BT de la Figura 24.9. Antes de actualizar el sueldo de Pérez de 25.000 a 30.000, en EMP_BT sólo existía $V_1$, que era la versión actual y que tenía *uc* como valor para Tft. Después, una transacción $T$ cuya marca de tiempo TS($T$) es '04-06-2003,08:56:12' actualiza el sueldo a 30.000 e indica '01-06-2003' como tiempo válido efectivo. Se crea la tupla $V_2$, que es una copia de $V_1$ excepto por el hecho de que Tfv contiene el valor '31-05-2003', un día menos que el nuevo tiempo válido y su Tit es la marca de tiempo de la transacción de actualización. Se crea también la tupla $V_3$, que contiene el nuevo salario, y cuyo Tiv se ajusta al valor '01-06-2003' y su Tit contiene también la marca de tiempo de la transacción de actualización. Para concluir, se almacena en el Tft de $V_1$ la marca de tiempo de la transacción de actualización, '04-06-2003,08:56:12'. Observe que esto es una *actualización retroactiva,* ya que la modificación se pone en marcha el 4 de junio de 2003, pero el cambio del salario es efectivo el 1 de junio de 2003.

De forma análoga, cuando el salario y el departamento de Campos se actualizan (a la vez) a 30.000 y 5, la marca de tiempo de la transacción de actualización es '07-01-2001,14:33:02', mientras que el tiempo válido efectivo de la actualización es '01-02-2001'. Por consiguiente, ésta es una *actualización preventiva* porque la transacción entra en juego el 7 de enero de 2001, aunque la fecha efectiva era 1 de febrero de 2001. En este caso, la tupla $V_4$ es reemplazada de forma lógica por $V_5$ y $V_6$.

A continuación vamos a ver cómo puede implementarse una **operación de borrado** en una relación bitemporal considerando las tuplas $V_9$ y $V_{10}$ de la relación EMP_BT en la Figura 24.9. Aquí, el empleado Torres deja la compañía de forma efectiva el 10 de agosto de 2002, mientras que el borrado lógico lo lleva a cabo una transacción $T$ con TS($T$) = 12-08-2002,10:11:07. Antes de esto, V9 era la versión actual de Torres y su Tft era *uc.* El borrado lógico se implementa asignando a $V_9$.Tft el valor 12-08-2002,10:11:07 para invalidarlo, y creando la *versión final* $V_{10}$ para Torres cuyo Tfv = 10-08-2002 (véase la Figura 24.9). Para terminar, se implementa una **operación de inserción** creando la *versión inicial* tal y como muestra $V_{11}$ en la tabla EMP_BT.

**Consideraciones de la implementación.** Existen varias opciones a la hora de almacenar las tuplas en una relación temporal. Una de ellas es guardar todas las tuplas en la misma tabla, tal y como muestran las Figuras 24.8 y 24.9. Otra posibilidad es crear dos tablas: una para contener la información que es correcta en la actualidad y otra para el resto de las tuplas. Por ejemplo, en la relación bitemporal EMP_BT, las tuplas cuyos valores Tft y Tfv fueran *uc* y *now*, respectivamente, podrían estar en una relación, la *tabla actual,* ya que son las únicas actualmente correctas (esto es, representan la instantánea actual), mientras que el resto de tuplas podrían ir a otra relación. Esto permite que el administrador de la base de datos tenga rutas de acceso diferentes, así como índices por cada relación, y mantenga el tamaño de la misma dentro de lo razonable. Otra posibilidad es crear una tercera tabla para las tuplas corregidas cuyo Tft no es *uc.*

Otra opción es la *partición vertical* de los atributos de la relación temporal en relaciones separadas ya que, si una de ellas tiene muchos atributos, se creará una nueva versión completa de la tupla siempre que cualquiera de esos atributos se modifiquen. Si dicha actualización se realiza de forma asíncrona, cada nueva versión podría diferir de la anterior en un único atributo, repitiéndose los demás. Si se crea una relación separada que sólo contenga los atributos que *siempre cambian sincronizadamente,* repitiendo la clave primaria en cada relación, la base de datos se dice que está en **forma normal temporal**. Sin embargo, para combinar la información, será necesaria una variación de la concatenación conocida como **concatenación de intersección temporal**, cuya implementación suele ser costosa.

Es importante indicar que las bases de datos bitemporales permiten un registro completo de los cambios. Incluso es posible un registro de correcciones. Por ejemplo, es posible que dos versiones de tupla del mismo empleado tengan el mismo tiempo válido pero con valores de atributo diferentes siempre y cuando sus tiempos de transacción sean disjuntos. En este caso, la tupla con el tiempo de transacción más tardío será una **corrección** de otra versión de la tupla. De esta forma se puede corregir incluso tiempos válidos introducidos de forma incorrecta. Para las consultas, el estado incorrecto de la base de datos estará disponible como

**Figura 24.9.** Algunas versiones de tupla en las relaciones bitemporales EMP_BT y DEPT_BT.

**EMP_BT**

| Nombre | Dni | Sueldo | Dno | SuperDni | Tiv | Tfv | Tit | Tft |
|---|---|---|---|---|---|---|---|---|
| Pérez | 123456789 | 25000 | 5 | 333445555 | 15-06-2002 | Now | 08-06-2002, 13:05:58 | 04-06-2003, 08:56:12 |
| Pérez | 123456789 | 25000 | 5 | 333445555 | 15-06-2002 | 31-05-2003 | 04-06-2003, 08:56:12 | uc |
| Pérez | 123456789 | 30000 | 5 | 333445555 | 01-06-2003 | Now | 04-06-2003, 08:56:12 | uc |
| Campos | 333445555 | 25000 | 4 | 999887777 | 20-08-1999 | Now | 20-08-1999, 11:18:23 | 07-01-2001, 14:33:02 |
| Campos | 333445555 | 25000 | 4 | 999887777 | 20-08-1999 | 31-01-2001 | 07-01-2001, 14:33:02 | uc |
| Campos | 333445555 | 30000 | 5 | 999887777 | 01-02-2001 | Now | 07-01-2001, 14:33:02 | 28-03-2002, 09:23:57 |
| Campos | 333445555 | 30000 | 5 | 999887777 | 01-02-2001 | 31-03-2002 | 28-03-2002, 09:23:57 | uc |
| Campos | 333445555 | 40000 | 5 | 888667777 | 01-04-2002 | Now | 28-03-2002, 09:23:57 | uc |
| Torres | 222447777 | 28000 | 4 | 999887777 | 01-05-2001 | Now | 27-04-2001, 16:22:05 | 12-08-2002, 10:11:07 |
| Torres | 222447777 | 28000 | 4 | 999887777 | 01-05-2001 | 10-08-2002 | 12-08-2002, 10:11:07 | uc |
| Ojeda | 666884444 | 38000 | 5 | 333445555 | 01-08-2003 | Now | 28-07-2003, 09:25:37 | uc |

. . .

**DEPT_BT**

| NombreDpto | Dno | DniDirector | Tiv | Tfv | Tit | Tft |
|---|---|---|---|---|---|---|
| Investigación | 5 | 888665555 | 20-09-2001 | Now | 15-09-2001, 14:52:12 | 28-03-2001, 09:23:57 |
| Investigación | 5 | 888665555 | 20-09-2001 | 31-03-1997 | 28-03-2002, 09:23:57 | uc |
| Investigación | 5 | 333445555 | 01-04-2002 | Now | 28-03-2002, 09:23:57 | uc |

. . .

un estado de base de datos previo. Una base de datos que mantiene un registro completo de cambio y correcciones recibe a veces el nombre de **base de datos para sólo añadir**.

### 24.2.3 Incorporación del tiempo en las bases de datos orientadas a objetos utilizando el versionado de atributos

La sección anterior explicó el **método del versionado de tuplas** para implementar las bases de datos de tiempo. En este método, siempre que cambia el valor de un atributo, se crea una nueva versión de la tupla entera, aunque el resto de los valores de atributo sigan siendo idénticos a la versión anterior de la tupla. En los sistemas de bases de datos que soporten **objetos estructurados complejos**, como las bases de datos de objetos (consulte los Capítulos 20 y 21) o los sistemas objeto-relacional (consulte el Capítulo 22), es posible utilizar un método alternativo, conocido como **versionado de atributos**.[21]

En el versionado de atributos, se utiliza un solo objeto complejo para almacenar todos los cambios temporales del objeto. Cada atributo que cambia con el tiempo se denomina **atributo que varía con el tiempo**, y tiene sus valores versionados sobre el tiempo añadiendo periodos temporales al atributo. Los periodos temporales pueden representar un tiempo válido, un tiempo de transacción o ser bitemporal, en función de los requisitos de la aplicación. Los **atributos que no cambian con el tiempo** no están asociados con los periodos temporales. Para ilustrar esto, consideremos el ejemplo de la Figura 24.10, que es un atributo versionado de la repre-

[21] El versionado de atributos también puede utilizarse en el modelo relacional anidado (consulte el Capítulo 22).

**Figura 24.10.** Esquema ODL posible para una clase de objeto EMPLEADO_VT de tiempo válido temporal utilizando el versionado de atributos.

```
class TEMPORAL_SUELDO
{ attribute   Date            Tiempo_inicio_válido;
  attribute   Date            Tiempo_final_válido;
  attribute   float           Sueldo;
};
class TEMPORAL_DPTO
{ attribute   Date                    Tiempo_inicio_válido;
  attribute   Date                    Tiempo_final_válido;
  attribute   DEPARTAMENTO_VT   Dept;
};
class TEMPORAL_SUPERVISOR
{ attribute   Date                    Tiempo_inicio_válido;
  attribute   Date                    Tiempo_final_válido;
  attribute   EMPLEADO_VT         Supervisor;
};
class TEMPORAL_TIEMPOVIDA
{ attribute   Date                    Tiempo_inicio_válido;
  attribute   Date                    Tiempo_final_válido;
};
class EMPLEADO_VT
( extent EMPLEADOS )
{ attribute   list<TEMPORAL_TIEMPOVIDA>     lifespan;
  attribute   string                        Nombre;
  attribute   string                        Dni;
  attribute   list<TEMPORAL_SUELDO>         Histórico_sueldo;
  attribute   list<TEMPORAL_DPTO>           Histórico_dpto;
  attribute   list <TEMPORAL_SUPERVISOR>    Histórico_supervisor;
};
```

sentación de un tiempo válido de EMPLEADO utilizando la notación ODL de las bases de datos de objetos (consulte el Capítulo 21). Aquí, asumimos que el nombre y el documento nacional de identidad son atributos que no varían con el tiempo, mientras que el sueldo, el departamento y el supervisor son atributos que sí lo hacen (pueden cambiar con el transcurso del tiempo). Cada atributo que varía con el tiempo está representado como una lista de tuplas <Tiempo_inicio_válido, Tiempo_final_válido, Valor>, ordenada por el tiempo de inicio válido.

Siempre que un atributo cambia en este modelo, la versión actual del atributo está cerrada y a la lista se añade una **nueva versión del atributo** para este atributo. Esto permite que los atributos cambien asíncronamente. El valor actual para cada atributo tiene *now* (el presente) como Tiempo_final_válido. Al utilizar el versionado de atributos, resulta útil incluir un **atributo temporal de tiempo de vida** asociado con el objeto entero cuyo valor es uno o más periodos de tiempo válidos que indican el tiempo válido de existencia para todo el objeto.

La eliminación lógica del objeto se implementa cerrando el tiempo de vida. Es preciso implementar la restricción de que cualquier periodo de tiempo de un atributo dentro de un objeto debe ser un subconjunto del tiempo de vida del objeto.

En el caso de las bases de datos bitemporales, cada versión de atributo tendría una tupla con cinco componentes:

<Tiempo_inicio_válido, Tiempo_final_válido, Tiempo_inicio_trans, Tiempo_final_trans, Valor>

El tiempo de vida del objeto también debe incluir las dimensiones de tiempo válido y de transacción. Por consiguiente, las capacidades completas de las bases de datos bitemporales pueden estar disponibles con el versionado de atributos. Mecanismos parecidos a los explicados anteriormente para actualizar las versiones de las tuplas se pueden aplicar para actualizar las versiones de los atributos.

### 24.2.4 Construcciones de consultas de tiempo y el lenguaje TSQL2

Hasta ahora, hemos explicado cómo pueden extenderse los modelos de datos con construcciones temporales. Ahora vamos a ofrecer una visión general y breve de cómo las operaciones de consulta tienen que extenderse para las consultas de tiempos. También explicaremos brevemente el lenguaje TSQL2, que extiende SQL para poder consultar tiempos válidos, tiempos de transacción y bases de datos relacionales bitemporales.

En las bases de datos relacionales no temporales, las condiciones de selección típicas implican condiciones de atributos, y del conjunto de *tuplas actuales* se seleccionan las tuplas que satisfacen esas condiciones. A continuación de esto debemos especificar los atributos de interés para la consulta mediante una *operación de proyección* (consulte el Capítulo 5). Por ejemplo, en la consulta para recuperar los nombres de todos los empleados que trabajan en el departamento 5 cuyo sueldo es superior a 30.000, la condición de selección sería la siguiente:

((Sueldo > 30000) AND (Dno = 5))

El atributo proyectado sería Nombre. En una base de datos temporal, en las condiciones pueden estar implicados el tiempo y los atributos. Una **condición de tiempo pura** sólo involucra al tiempo; por ejemplo, para seleccionar todas las versiones de tuplas de empleado que eran válidas en un *momento de tiempo determinado T* o que fueron válidas *durante un cierto periodo de tiempo* $[T_1, T_2]$. En este caso, el periodo de tiempo especificado se compara con el periodo de tiempo válido de cada versión de tupla [*T*.Tiv, *T*.Tfv], y sólo se seleccionan las tuplas que satisfacen la condición. En estas operaciones, se considera que un periodo es equivalente al conjunto de puntos de tiempo de $T_1$ a $T_2$ inclusive, por lo que podemos utilizar las operaciones de comparación de conjuntos estándar. También necesitamos otras operaciones adicionales, como si un periodo de tiempo termina *antes* de que otro empiece.[22]

Algunas de las operaciones más comunes que se utilizan en las consultas son las siguientes:

| | |
|---|---|
| [*T*.Tiv, *T*.Tfv] **INCLUDES** $[T_1, T_2]$ | equivale a $T_1 \geqslant T$.Tiv AND $T_2 \leqslant T$.Tfv |
| [*T*.Tiv, *T*.Tfv] **INCLUDED_IN** $[T_1, T_2]$ | equivale a $T_1 \leqslant T$.Tiv AND $T_2 \geqslant T$.Tfv |
| [*T*.Tiv, *T*.Tfv] **OVERLAPS** $[T_1, T_2]$ | equivale a ($T_1 \leqslant T$.Tfv AND $T_2 \geqslant T$.Tiv)[23] |
| [*T*.Tiv, *T*.Tfv] **BEFORE** $[T_1, T_2]$ | equivale a $T_1 \geqslant T$.Tfv |
| [*T*.Tiv, *T*.Tfv] **AFTER** $[T_1, T_2]$ | equivale a $T_2 \leqslant T$.Tiv |

[22] Se ha definido un completo conjunto de operaciones, conocido como **álgebra de Allen**, para la comparación de periodos de tiempo.

[23] Esta operación devuelve verdadero si la *intersección* de los dos periodos no está vacía; también se ha conocido como INTERSECTS_WITH.

[*T*.Tiv, *T*.Tfv] **MEETS_BEFORE** [$T_1$, $T_2$] equivale a $T_1 = T.\text{Tfv} + 1$[24]

[*T*.Tiv, *T*.Tfv] **MEETS_AFTER** [$T_1$, $T_2$] equivale a $T_2 + T_1 = T.\text{Tiv}$

Además, necesitamos operaciones para manipular los periodos de tiempo, como el cálculo de la unión o la intersección de dos periodos de tiempo. Es posible que los resultados de estas operaciones no sean a su vez periodos, sino más bien **elementos temporales** (una colección de uno o más periodos de tiempo disjuntos tal que no hay dos periodos de tiempo en un elemento temporal que sean directamente adyacentes). Es decir, para cualesquiera dos periodos de tiempo [$T_1$, $T_2$] y [$T_3$, $T_4$] de un elemento temporal, deben cumplirse estas tres condiciones:

- La intersección de [$T_1$, $T_2$] y [$T_3$, $T_4$] está vacía.
- $T_3$ no es el punto de tiempo que sigue a $T_2$ en la granularidad dada.
- $T_1$ no es el punto de tiempo que sigue a $T_4$ en la granularidad dada.

Las últimas condiciones son necesarias para garantizar unas representaciones únicas de los elementos temporales. Si dos periodos de tiempo [$T_1$, $T_2$] y [$T_3$, $T_4$] son adyacentes, se combinan o **fusionan** en un solo periodo de tiempo [$T_1$, $T_4$]. Esta fusión también combina los periodos de tiempo intersectados.

Para mostrar cómo se pueden utilizar las condiciones de tiempos puros, pensemos en un usuario que quiere seleccionar todas las versiones de empleado que fueron válidas en cualquier punto durante 2002. La condición de selección apropiada aplicada a la relación de la Figura 24.8 sería:

[*T*.Tiv, *T*.Tfv] **OVERLAPS** [2002-01-01, 2002-12-31]

Normalmente, la mayoría de las selecciones temporales se aplican a la dimensión de tiempo válida. En una base de datos bitemporal, normalmente se aplican las condiciones a las tuplas actualmente correctas, con *uc* como sus tiempos de finalización de transacción. Sin embargo, si es preciso aplicar la consulta a un estado anterior de la base de datos, se añade una cláusula AS_OF *T* a la consulta, lo que se traduce en que la consulta se aplica a las tuplas de tiempo válidas que fueron correctas en la base de datos en el momento *T*.

Además de las condiciones de tiempo puras, otras selecciones implican **atributos y condiciones de tiempo**. Por ejemplo, suponga que queremos recuperar todas las versiones de tupla EMP_TV T de los empleados que trabajaron en el departamento 5 en cualquier momento durante 2002. En este caso, la condición es la siguiente:

[*T*.Tiv, *T*.Tfv] **OVERLAPS** [2002-01-01, 2002-12-31] AND (*T*.Dno = 5)

Por último, ofrecemos una visión general breve del lenguaje de consulta TSQL2, que extiende SQL con estructuras para las bases de datos de tiempo. La idea principal tras TSQL2 es permitir al usuario especificar si una relación es o no es temporal (en este último caso, sería una relación SQL estándar). La sentencia CREATE TABLE está extendida con una cláusula AS *opcional* para permitir al usuario declarar diferentes opciones temporales. Están disponibles las siguientes opciones:

- AS VALID STATE <GRANULARITY>
  (relación de tiempo válido con periodo de tiempo válido)
- AS VALID EVENT <GRANULARITY>
  (relación de tiempo válido con punto de tiempo válido)
- AS TRANSACTION
  (relación de tiempo de transacción con periodo de tiempo de transacción)
- AS VALID STATE <GRANULARITY> AND TRANSACTION
  (relación bitemporal, periodo de tiempo válido)

[24] Aquí, 1 se refiere a un punto de tiempo en la granularidad especificada. Las operaciones MEETS especifican básicamente si un periodo empieza inmediatamente después de que termine otro periodo.

- AS VALID EVENT <GRANULARITY> AND TRANSACTION
  (relación bitemporal, punto de tiempo válido)

Las palabras clave STATE y EVENT se utilizan para especificar si un *periodo* de tiempo o un *punto* de tiempo está asociado con la dimensión de tiempo válida. En TSQL2, en lugar de que el usuario tenga que ver cómo están implementadas las tablas temporales (como explicamos en las secciones anteriores), el lenguaje TSQL2 añade estructuras de consulta para especificar distintos tipos de selecciones temporales, proyecciones temporales, agregaciones temporales, transformaciones entre granularidades y muchos otros conceptos. El libro Snodgrass y otros (1995) describe este lenguaje.

### 24.2.5 Datos de series de tiempo

Los datos de series de tiempo se utilizan con frecuencia en las aplicaciones financieras, de ventas y económicas. Implican valores de datos que se graban según una secuencia específica de puntos de tiempo. Por consiguiente, hay un tipo especial de **datos de evento válidos**, donde los puntos de tiempo del evento están predeterminados de acuerdo con un calendario fijo. Consideremos el ejemplo del precio de las acciones de una determinada empresa al cierre de la Bolsa. Aquí, la granularidad es de un día, pero se conocen los días de apertura del Mercado. Por tanto, es normal especificar un procedimiento de cálculo que calcule el **calendario** particular asociado con las series de tiempo. Las consultas de tiempo en las series de tiempo implican la **agregación temporal** sobre intervalos de granularidad más altos (por ejemplo, descubrir el precio máximo y el promedio de la acción durante la semana, o los precios máximo y mínimo de la acción en Bolsa durante el mes, a partir de una información diaria).

Otro ejemplo serían las ventas diarias de cada una de las tiendas de una cadena propiedad de una empresa concreta. Una vez más, las agregaciones temporales típicas serían la recuperación de ventas semanales, mensuales o anuales a partir de las ventas diarias (utilizando la función de agregación suma); también se pueden comparar las ventas mensuales de una tienda con las ventas de meses anteriores, etcétera.

Debido a la naturaleza especializada de los datos de series de tiempo y a la carencia de soporte en los DBMSs más antiguos, es normal utilizar **sistemas de administración de series de tiempo** en lugar de DBMSs de propósito general para gestionar dicha información. En estos sistemas, es frecuente almacenar valores de series de tiempo en orden secuencial dentro de un fichero, y aplicar procedimientos de series de tiempo especializados para analizar la información. El problema de este método es que en estos sistemas no contamos con toda la potencia de los lenguajes de consulta de alto nivel, como SQL.

Más recientemente, algunos paquetes DBMS comerciales ofrecen extensiones de series de tiempo, como las series *data blade* de Informix Universal Server (consulte el Capítulo 22). Además, el lenguaje TSQL2 proporciona algo de soporte para las series de tiempo en forma de tablas de eventos.

## 24.3 Bases de datos multimedia y espaciales

Debido a que los dos temas tratados en esta sección son muy amplios, sólo ofreceremos una breve introducción a ellos. La Sección 24.3.1 presenta las bases de datos espaciales, mientras que la 24.3.2 se encarga brevemente de las de tipo multimedia.

### 24.3.1 Introducción a los conceptos de base de datos espacial

Las **bases de datos espaciales** ofrecen el soporte necesario para gestionar las bases de datos que siguen la pista de objetos en un espacio multidimensional. Por ejemplo, las bases de datos cartográficas que almacenan mapas incluyen descripciones espaciales en dos dimensiones acerca de sus objetos (desde países y estados a ríos, ciudades, carreteras, mares, etc.). Estas aplicaciones son conocidas también como GIS (Sistemas de

información geográfica, *Geographical Information Systems*)[25], y se emplean en áreas muy diversas como el medioambiente, las emergencias y la dirección de batallas. Otras bases de datos, como las meteorológicas que mantienen información acerca del clima, son tridimensionales, ya que la temperatura y otro tipo de información meteorológica están relacionados con puntos espaciales en tres dimensiones. En general, una base de datos espacial contiene objetos que cuentan con características espaciales que los describen. Las relaciones espaciales existentes entre los objetos son importantes, y con frecuencia requeridos cuando se consulta la base de datos. Aunque, en general, una base de datos espacial puede hacer referencia a un espacio *n*-dimensional para cualquier valor de *n,* nosotros vamos a limitarnos a dos dimensiones.

Las principales extensiones necesarias para las bases de datos espaciales son modelos capaces de interpretar características espaciales. Además, y para mejorar el rendimiento, a veces se hace necesario el uso de indexación y estructuras de almacenamiento. En primer lugar trataremos algunas de las extensiones modelo para las bases de datos espaciales en dos dimensiones. Las extensiones básicas necesarias incluyen algunos conceptos geométricos bidimensionales, como puntos, líneas, segmentos, círculos, polígonos y arcos, que permiten especificar las características espaciales de los objetos. Por otro lado, son necesarias también operaciones espaciales para operar sobre dichas características (por ejemplo, para obtener la distancia entre dos objetos), así como condiciones lógicas espaciales (por ejemplo, para determinar si dos objetos están solapados en el espacio). Para ilustrar todo esto vamos a considerar una base de datos usada en aplicaciones de gestión de emergencias. Será necesaria una descripción de las posiciones espaciales de muchos tipos de objetos. Algunos de estos objetos suelen contar con características espaciales estáticas, como calles o autopistas, bombas de agua (para el control de incendios), comisarías, parques de bomberos y hospitales, mientras que en otros, esas características son dinámicas y varían a lo largo del tiempo (como ocurre con los vehículos policiales, ambulancias y camiones de bomberos).

Las siguientes categorías ilustran tres tipos de consultas espaciales típicas:

- **Consulta de rango.** Localiza los objetos de un tipo particular que se encuentran dentro de un área espacial concreta o a una distancia particular de una localización dada (por ejemplo, localiza todos los hospitales dentro del área metropolitana de Madrid o las ambulancias que se encuentran a 10 kilómetros de un accidente).
- **Consulta del vecino más próximo.** Encuentra un objeto de un tipo particular que está cerca de una localización (por ejemplo, el vehículo policial que está más próximo a la localización de un delito).
- **Concatenaciones espaciales u *overlays*.** Generalmente, concatena los objetos de dos tipos basándose en una condición espacial, tales como los que intersectan o se solapan espacialmente o los que están separados una cierta distancia (por ejemplo, localiza todas las ciudades que se encuentren en una carretera principal o todas las casas que estén a tres kilómetros de un lago).

Para que estas y otras consultas espaciales sean ejecutadas eficazmente, son precisas una serie de técnicas especiales para la indexación espacial. Una de las mejores elecciones es el uso de **árboles-R** y sus variantes. Los árboles-R agrupan los objetos que están próximos al mismo nodo hoja de un índice de árbol estructurado. Ya que uno de estos nodos sólo puede apuntar a un cierto número de objetos, son precisos algoritmos para dividir el espacio en subespacios rectangulares que incluyan los objetos. Entre los criterios para llevar a cabo esta división espacial está minimizar las áreas del rectángulo, ya que esto conduce a un estrechamiento del espacio de búsqueda. Problemas como tener objetos con áreas espaciales solapadas son gestionados de forma diferente dependiendo de las variaciones de los árboles-R. Los nodos internos de estos árboles están asociados con rectángulos cuyas áreas abarcan todos los rectángulos en su subárbol. Por tanto, los árboles-R pueden responder fácilmente a preguntas como la localización de todos los objetos de un área determinada limitando el árbol de búsqueda a aquellos subárboles cuyos rectángulos intersecten con el área especificada en la consulta.

[25] La Sección 30.3 está dedicada en exclusiva a tratar sobre los problemas de administración de datos relacionados con el GIS.

Otras estructuras de almacenamiento especial incluyen los **árboles cuadrados** y sus variaciones. Estos árboles, generalmente, dividen cada espacio o subespacio en áreas del mismo tamaño, y proceden con las subdivisiones de cada subespacio para identificar las posiciones de los distintos objetos. Recientemente, se han propuesto muchas nuevas estructuras de acceso espacial, y éste sigue siendo un campo activo de investigación.

### 24.3.2 Introducción a los conceptos de base de datos multimedia[26]

Las **bases de datos multimedia** proporcionan características que permiten a los usuarios almacenar y consultar diferentes tipos de información multimedia, entre la que se incluye *imágenes* (como fotografías o dibujos), *clips de vídeo* (como películas, noticiarios o vídeos caseros), *clips de audio* (como canciones, mensajes telefónicos o alocuciones) y *documentos* (como libros o artículos). Los tipos principales de consultas de base de datos implican la localización de las fuentes multimedia que contengan ciertos objetos, por ejemplo, localizar todos los clips de vídeo de una base de datos de vídeos que incluyan a una cierta persona (digamos, Bill Clinton). Otra posibilidad sería obtener clips de vídeo basándonos en ciertas actividades incluidas en ellos, como los goles marcados por cierto jugador o equipo.

Las consultas anteriores son conocidas como **recuperaciones basadas en el contenido**, ya que la fuente multimedia se obtiene en base a que contenga ciertos objetos u actividades. Por tanto, una base de datos multimedia debe utilizar algún modelo para organizar e indexar las fuentes en función de sus contenidos. La *identificación de los contenidos* de las fuentes multimedia es una tarea complicada y que consume mucho tiempo. Existen dos métodos fundamentales. El primero está basado en el **análisis automático** de dichas fuentes para identificar ciertas características matemáticas de sus contenidos. Esta técnica emplea diferentes algoritmos dependiendo del tipo de origen multimedia (imagen, vídeo, audio o texto). El segundo método depende de la **identificación manual** de los objetos y actividades de interés de cada fuente multimedia y del uso de esta información para indexar esas fuentes. Este enfoque puede aplicarse a todas ellas, pero requiere de una fase de preprocesamiento manual en la que una persona tiene que estudiar cada una de las fuentes, identificar y catalogar los objetos y actividades que contiene, y usar esta información para indexarlas.

A lo largo de esta sección comentaremos brevemente algunas de las características de cada tipo de origen multimedia (imágenes, vídeo, audio y texto).

Una **imagen** puede estar almacenada en bruto como un conjunto de valores de píxel o celda, o en un formato comprimido para ahorrar espacio. El *descriptor de forma* de la imagen describe la forma geométrica de la imagen en bruto, la cual suele ser un rectángulo de **celdas** de una cierta altura y anchura. Por tanto, cada imagen puede estar representada por una rejilla de celdas de $m$ por $n$. Cada celda contiene un píxel que describe el contenido de la misma. En imágenes en blanco y negro, los píxeles pueden ser un bit. En imágenes en escala de grises o en color, un píxel se corresponde con múltiples bits. Ya que las imágenes pueden precisar de grandes cantidades de espacio de almacenamiento, suelen almacenarse en formato comprimido. Los estándares de compresión, como GIF, JPEG o MPEG, usan diferentes transformaciones matemáticas para reducir el número de celdas almacenadas pero manteniendo las características principales de la imagen. Entre dichas técnicas se incluyen DFT (Transformación discontinua de Fourier, *Discrete Fourier Transform*), DCT (Transformación discontinua de coseno, *Discrete Cosine Transform*) y las transformaciones de pequeña ondulación.

Para identificar los objetos interesantes de una imagen, ésta suele dividirse en segmentos homogéneos usando un *predicado de homogeneidad*. Por ejemplo, en una imagen a color, las celdas adyacentes que tienen píxeles similares se agrupan en un segmento. El predicado de homogeneidad define condiciones para un agrupamiento automático de esas celdas. De este modo, la segmentación y la compresión pueden identificar las características principales de una imagen.

---

[26] La Sección 30.2 trata con detalle las bases de datos multimedia y la administración de la información multimedia.

Una típica consulta a una base de datos de imágenes podría ser localizar las imágenes que son similares a otra dada. Esta imagen podría ser un segmento aislado que contiene, digamos, un patrón, mientras que la consulta es la localización de otras imágenes que contienen ese mismo patrón. Para realizar este tipo de búsqueda existen dos tipos de técnicas principales. La primera utiliza una **función de distancia** para comparar la imagen buscada con las que están almacenadas y con sus segmentos. Si el valor de distancia devuelto es pequeño, la posibilidad de una coincidencia es alta. Se pueden crear índices para agrupar las imágenes que están próximas en la distancia métrica, de modo que se limite el espacio de búsqueda. El segundo método, llamado **aproximación por transformación**, mide las similitudes de la imagen haciendo el menor número de transformaciones para que las celdas de una imagen coincidan con las de otra. Estas transformaciones incluyen rotaciones, traslaciones y escalados. Aunque este último método es más general, es también el más complejo y el que consume más tiempo.

Un **vídeo** está representado normalmente como una secuencia de *frames* o fotogramas, donde cada uno de ellos es una imagen estática. Sin embargo, en lugar de identificar los objetos y actividades de cada *frame* individual, el vídeo está dividido en **segmentos de vídeo**, donde cada uno de ellos abarca una secuencia de *frames* contiguos que incluyen los mismos objetos/actividades. Cada segmento está identificado por sus *frames* inicial y final. Los objetos y actividades identificados en cada segmento de vídeo pueden utilizarse para indexar dichos segmentos. Para la indexación de video existe una técnica llamada *árboles de segmento de frame*. El índice incluye tanto objetos, como personas, casas, coches y actividades, como una persona *dando* un discurso o dos personas *hablando*. Los vídeos también están comprimidos usando estándares como el MPEG.

Un **texto/documento** es, básicamente, el texto completo de un artículo, libro o revista. Estas fuentes están indexadas normalmente identificando las palabras clave que aparecen en el texto y sus frecuencias relativas. Sin embargo, las palabras de relleno se eliminan del proceso. Debido a que pueden existir muchas de estas palabras clave a la hora de intentar indexar una colección de documentos, se han desarrollado una serie de técnicas para reducir su número a las más relevantes de la colección. Para este objetivo, existe una técnica llamada SVD (Descomposiciones de valor singular, *Singular Value Decompositions*), la cual se basa en transformaciones de matriz. Después puede emplearse otra técnica llamada árboles-TV (árboles de vector telescópico, *Telescoping Vector trees*) para agrupar documentos similares.

El **audio** incluye mensajes grabados, como discursos, presentaciones e incluso grabaciones telefónicas de seguridad o conversaciones por orden judicial. Aquí pueden usarse transformaciones discretas para identificar las características principales de la voz de ciertas personas para tener un patrón para su indexación y recuperación. Entre estas características de audio se incluyen el volumen, la intensidad, el tono y la pureza.

# 24.4 Introducción a las bases de datos deductivas

## 24.4.1 Panorámica de las bases de datos deductivas

En un sistema de bases de datos deductivas se suelen especificar las reglas mediante un **lenguaje declarativo** (un lenguaje en el que se indica qué se quiere conseguir en lugar de cómo hacerlo). Un **motor de inferencia** (o **mecanismo de deducción**) dentro del sistema puede deducir nuevos hechos interpretando esas reglas. El modelo usado por las bases de datos deductivas está íntimamente relacionado con el modelo de datos relacional, y particularmente con el formalismo de cálculo relacional de dominio (consulte la Sección 6.6). También está relacionado con el campo de la **lógica de programación** y el lenguaje **Prolog**. El trabajo de las bases de datos deductivas basado en la lógica ha usado Prolog como su punto de inicio. **Datalog** es una variación de Prolog usada para definir reglas de forma declarativa en conjunción con un conjunto de relaciones existente, las cuales son tratadas a su vez como literales en el lenguaje. Aunque la estructura de Datalog se asemeja mucho al de Prolog, sus semánticas operativas (es decir, el modo en que se ejecuta un programa Datalog) son muy diferentes.

Una base de datos deductiva utiliza dos tipos de especificaciones principales: hechos y reglas. Los **hechos** están especificados de una forma similar a como lo están las relaciones, excepto por el hecho de que no es necesario incluir los nombres de atributo. Recuerde que una tupla en una relación describe algún hecho del mundo real cuyo significado está parcialmente determinado por los nombres de atributo. En una base de datos deductiva, el significado de un atributo en una tupla está determinado únicamente por su *posición* dentro de la misma. Las **reglas** son algo parecido a las vistas. Especifican relaciones virtuales que no están almacenadas pero que pueden formarse a partir de los hechos aplicando mecanismos de inferencia basados en las especificaciones de reglas. La diferencia principal entre reglas y vistas es que las primeras pueden comportar recursividad y, por tanto, producir relaciones virtuales que no pueden obtenerse en términos de vistas relacionales básicas.

La evaluación de los programas Prolog está basada en una técnica llamada *encadenamiento hacia atrás* (*backward chaining*), la cual implica una evaluación de objetivos de arriba hacia abajo. En las bases de datos deductivas que usan Datalog, la atención ha sido consagrada a manipular grandes volúmenes de datos almacenados en una base de datos relacional. Por tanto, se han diseñado técnicas que las asemeja a una evaluación de abajo hacia arriba. Prolog sufre de la limitación de que el orden de especificación de los hechos y las reglas es importante a la hora de llevar a cabo la evaluación; además, el orden de los literales (definidos en la Sección 24.4.3) dentro de una regla es significativo. Las técnicas de ejecución para los programas Datalog intentan soslayar estos problemas.

## 24.4.2 Notación Prolog/Datalog

La notación usada en Prolog/Datalog está basada en proporcionar predicados con nombres únicos. Un **predicado** tiene un significado implícito, el cual está sugerido por su nombre, y un número fijo de **argumentos**. Si todos ellos son valores constantes, el predicado simplemente declara que un cierto hecho es verdadero. Si, por el contrario, el predicado tiene variables como argumentos, éste puede considerarse como una consulta o como parte de una regla o una restricción. A lo largo de este capítulo adoptamos el convenio Prolog de que todos los **valores constantes** de un predicado pueden ser cadenas *numéricas* o de caracteres que están representadas por identificadores (o nombres) que empiezan con *letras minúsculas*, mientras que los **nombres de variables** siempre empiezan con una *letra mayúscula.*

Considere el ejemplo mostrado en la Figura 24.11, la cual está basada en la base de datos relacional de la Figura 5.6, pero mucho más simplificada. Existen tres nombres de predicado: *supervisar, superior* y *subordinado.* El predicado supervisar está definido mediante un conjunto de hechos, cada uno de ellos con dos argumentos: un nombre de supervisor seguido por el nombre de un supervisado *directo* (subordinado) de ese supervisor. Estos hechos se corresponden con los datos actuales almacenados en la base de datos, y pueden ser considerados como constituyentes de un conjunto de tuplas en una relación SUPERVISAR con dos atributos cuyo esquema es:

SUPERVISAR(Supervisor, Supervisado)

Por tanto, supervisado(*X*, *Y*) indica el hecho de que *X supervisa a Y*. Observe la omisión del nombre del atributo en la notación Prolog. Los nombres de atributo sólo están representados por virtud de la posición de cada argumento en un predicado: el primero representa al supervisor y el segundo al subordinado directo.

Los otros dos nombres de predicado están definidos mediante reglas. La principal contribución de las bases de datos deductivas es la capacidad para especificar reglas recursivas y para proporcionar un entorno de trabajo para la deducción de nueva información basada en las reglas especificadas. Una regla tiene la forma **cabecera :– cuerpo**, donde los caracteres :– indican *si y sólo si*. Una regla tiene, por lo general, un **único predicado** a la izquierda del símbolo :– (llamado **cabecera,** LHS [**lado izquierdo, *Left-Hand Side***] o **conclusión** de la regla) y **uno o más predicados** a su derecha (el **cuerpo**, RHS [**lado derecho, *Right-Hand Side***] o **premisa(s)** de la regla). Un predicado con constantes se dice que es ***ground*** o de **predicado instanciado**. Los argumentos de los predicados que aparecen en una regla suelen incluir un número de símbolos variables, aun-

que también pueden contener constantes como argumentos. Una regla especifica que si una asignación particular, o ***binding***, de valores constantes a las variables del cuerpo (predicados RHS) hace que *todos* estos predicados RHS sean **verdaderos**, también hace verdadera la cabecera (predicado LHS) usando la misma asignación de valores constantes a las variables. Por consiguiente, una regla nos ofrece una forma de generar nuevos hechos que son instanciaciones de la cabecera de dicha regla. Estos nuevos hechos están basados en otros que ya existen y que se corresponden con las instanciaciones (o *bindings*) de predicados en el cuerpo de la regla. Observe que el listado de múltiples predicados en el cuerpo de una regla supone implícitamente la aplicación del operador **lógico AND** a esos predicados. Por tanto, las comas existentes entre los predicados RHS pueden leerse como *y*.

Consideremos la definición del predicado superior de la Figura 24.11, cuyo primer argumento es el nombre de un empleado, mientras que el segundo es un empleado que puede ser un subordinado *directo* o *indirecto* del primero. Por *subordinado indirecto* entendemos aquel que depende de otro a cualquier nivel inferior. De esta manera, superior(*X*, *Y*) especifica el hecho de que *X es un superior de Y* ya sea por supervisión directa o indirecta. Podemos escribir dos reglas que, en conjunto, especifican el significado del nuevo predicado. La primera de las reglas que existen bajo Reglas en la Figura indica que, por cada valor de *X* e *Y*, si supervisado(*X*, *Y*) (el cuerpo de la regla) es verdadero, entonces superior(*X*, *Y*) (la cabecera de la regla) también lo es, ya que *Y* podría ser un subordinado directo de *X* (en un nivel inferior). Esta regla puede emplearse para generar todas las relaciones superior/subordinado directas a partir de los hechos que definen el predicado supervisar. La segunda regla recursiva especifica que, si supervisar(*X, Z*) *y* superior(*Z, Y*) son *ambas verdaderas*, entonces superior(*X*, *Y*) también lo es. Esto es un ejemplo de **regla recursiva**, donde uno de los predicados del cuerpo de la regla en el RHS es el mismo que el de la cabecera en el LHS. En general, el cuerpo de la regla define un número de premisas como que, si todas son verdaderas, podemos deducir que la conclusión en la cabecera de la misma también lo es. Observe que si tenemos dos (o más) reglas con la misma cabecera (predicado LHS), podemos concluir que el predicado es verdadero (esto es, que puede ser instanciado) si *cualquiera de* los cuerpos lo es; por tanto, es equivalente a una operación **lógica OR**. Por ejemplo, si tenemos estas dos reglas (*X*:– *Y, X*:– *Z*), podemos deducir que son equivalentes a *X* :– *Y* OR *Z*. Sin embargo, esta última forma no se emplea en sistemas deductivos porque no está en un formato estándar de regla, llamado cláusula Horn, como ya vimos en la Sección 24.4.4.

**Figura 24.11.** (a) Notación Prolog. (b) El árbol de supervisión.

**(a)** **Hechos**

SUPERVISAR(alberto, josé).
SUPERVISAR(alberto, fernando).
SUPERVISAR(alberto, aurora).
SUPERVISAR(juana, alicia).
SUPERVISAR(juana, luis).
SUPERVISAR(eduardo, alberto).
SUPERVISAR(eduardo, juana).
. . .

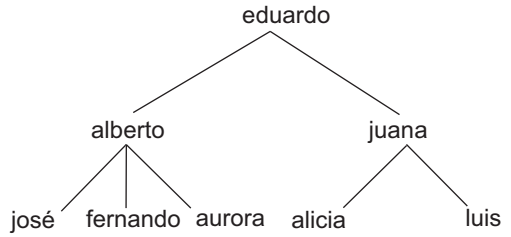
**Reglas**

SUPERIOR(*X*, *Y*) :- SUPERVISAR(*X*, *Y*).
SUPERIOR(*X*, *Y*) :- SUPERVISAR(*X*, Z), SUPERIOR(*Z*, *Y*).
SUBORDINADO(*X*, *Y*) :- SUPERIOR(*Y*, *X*).

**Consultas**

SUPERIOR(eduardo, *Y*)?
SUPERIOR(eduardo, aurora)?

**(b)**



Un sistema Prolog contiene una serie de predicados **incorporados** que el sistema puede interpretar directamente. Entre ellos se incluyen el operador de comparación de igualdad =(*X*, *Y*), que devuelve verdadero si *X* e *Y* son idénticos y que puede escribirse también como *X* = *Y* usando la notación estándar.[27] Otros operadores de comparación de números, como <, <=, > y >=, pueden considerarse como predicados binarios. En Prolog pueden usarse las funciones aritméticas +, –, * y / como argumentos en los predicados. Por el contrario, Datalog (en su forma básica) no lo permite; ciertamente, ésta es una de las principales diferencias que existen entre Prolog y Datalog. Sin embargo, versiones posteriores de Datalog si que lo permiten.

Una **consulta** suele involucrar un símbolo de predicado con un cierto número de argumentos variables, y su significado (o *respuesta*) es deducir todas las combinaciones constantes diferentes que, al ser **asignadas** a las variables, hacen que el predicado sea verdadero. Por ejemplo, la primera consulta de la Figura 24.11 solicita los nombres de todos los subordinados de *Eduardo* a cualquier nivel. Una consulta que sólo tenga constantes como argumentos devuelve tanto un resultado verdadero como falso, dependiendo de si los argumentos pueden ser deducidos de los hechos y las reglas. Por ejemplo, la segunda consulta de la misma figura devuelve verdadero, ya que puede deducirse superior(eduardo, aurora).

## 24.4.3 Notación Datalog

En Datalog, como en cualquier otro lenguaje basado en la lógica, un programa está constituido por objetos básicos llamados **fórmulas atómicas**. Es costumbre definir la sintaxis de este tipo de lenguajes describiendo la sintaxis de las fórmulas atómicas e identificando el modo en que pueden combinarse para formar un programa. En Datalog, estas fórmulas son **literales** que tienen la forma $p(a_1, a_2, ..., a_n)$, donde $p$ es el nombre del predicado y $n$ es su número de argumentos. Diferentes predicados pueden contar con distinto número de argumentos, por tanto $n$ de $p$ suele recibir a veces el nombre de **grado** de $p$. Los argumentos pueden ser constantes o nombres de variables. Como ya se mencionó anteriormente, usamos el convenio de que los valores constantes son numéricos o comenzar con una letra *minúscula*, mientras que los nombres de variable siempre empiezan con una letra *mayúscula*.

Datalog cuenta con una serie de **predicados incorporados**, los cuales pueden usarse también para construir fórmulas atómicas. Estos predicados son de dos tipos principalmente: los de comparación binaria [< (menor), <= (menor_o_igual), > (mayor) y >= (mayor_o_igual)] sobre dominios ordenados y los de comparación [= (igual) y /= (distinto)] en dominios ordenados o desordenados. Ambos pueden emplearse como predicados binarios con la misma sintaxis funcional que otros predicados (por ejemplo, escribiendo less(*X*, 3)) o usando la notación simplificada *X*<3. Observe que, ya que los dominios de estos predicados son potencialmente infinitos, deben usarse con precaución a la hora de definir reglas. Por ejemplo, el predicado greater(*X*, 3), si se usa por separado, genera un conjunto infinito de valores para *X* que satisface el predicado (todos los números enteros mayores que 3).

Un **literal** puede ser una fórmula atómica como las definidas anteriormente (llamado **literal positivo**) o precedida por **not**. Esta última forma es una fórmula atómica negada llamada **literal negativo**. Los programas Datalog pueden ser considerados como un subconjunto de las fórmulas de **cálculo de predicado**, que pueden considerarse similares a las de cálculo relacional de dominio (consulte la Sección 6.7). Sin embargo, en Datalog, estas fórmulas son convertidas primeramente en algo conocido como **forma clausal** antes de ser expresadas; y en Datalog sólo pueden usarse aquellas fórmulas expresadas en una forma clausal concreta, las cláusulas Horn.[28]

## 24.4.4 Forma clausal y cláusulas Horn

Recuerde de la Sección 6.6 que una fórmula en el cálculo relacional es una condición que incluye predicados denominados *átomos* (basados en los nombres de relación). Además, una fórmula puede tener cuantificado-

[27] Normalmente, un sistema Prolog cuenta con varios predicados de igualdad diferentes que pueden tener diferentes interpretaciones.

[28] Así denominadas en honor al matemático Alfred Horn.

res (en concreto, el *cuantificador universal* [para todo] y el *cuantificador existencial* [allí existe]). En la forma clausal, una fórmula debe transformarse en otra fórmula con las siguientes características:

- Todas las variables de la fórmula se cuantifican universalmente. Por tanto, no es preciso incluir explícitamente los cuantificadores universales (para todo); los cuantificadores se eliminan y todas las variables de la fórmula son cuantificadas implícitamente por el cuantificador universal.
- En la forma clausal, la fórmula está constituida por unas cuantas cláusulas, donde cada una de ellas está compuesta por unos *literales* conectados únicamente mediante conexiones lógicas OR. Por tanto, cada cláusula es una *disjunción* de literales.
- Las *propias cláusulas* están conectadas únicamente mediante conectores lógicos AND, para constituir la fórmula. Por tanto, la *forma clausal de una fórmula* es una *conjunción* de cláusulas.

Es demostrable que *cualquier fórmula puede convertirse en una forma clausal*. Para nuestros objetivos, estamos especialmente interesados en la forma de las cláusulas individuales; cada una es una disjunción de literales. Recuerde que los literales pueden ser positivos o negativos. Consideremos una cláusula de esta forma:

$$\text{NOT}(P_1) \text{ OR } \text{NOT}(P_2) \text{ OR } \ldots \text{ OR } \text{NOT}(P_n) \text{ OR } Q_1 \text{ OR } Q_2 \text{ OR } \ldots \text{ OR } Q_m \quad (1)$$

Esta cláusula tiene $n$ literales negativos y $m$ literales positivos. Una cláusula semejante puede transformarse en la siguiente fórmula lógica equivalente:

$$P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_n \Rightarrow Q_1 \text{ OR } Q_2 \text{ OR } \ldots \text{ OR } Q_m \quad (2)$$

donde $\Rightarrow$ es el símbolo **implica**. Las fórmulas (1) y (2) son equivalentes, lo que significa que sus valores de veracidad son siempre iguales. Ésta es la causa por la que, si todos los literales $P_i$ ($i$ = 1, 2, ..., $n$) son verdaderos, la fórmula (2) sólo es verdadera si al menos uno de los valores $Q_i$ es verdadero, que es el significado del símbolo $\Rightarrow$ (implica). Para la fórmula (1), si todos los literales $P_i$ ($i$ = 1, 2, ..., $n$) son verdaderos, sus negaciones son todas falsas; por lo que en este caso, la fórmula (1) es verdadera sólo si al menos uno de los valores $Q_i$ es verdadero. En Datalog, las reglas se expresan como una forma restringida de cláusulas denominadas **cláusulas Horn**, según las cuales una cláusula puede contener *a lo sumo un literal positivo*. Por tanto, una cláusula Horn es de la forma:

$$\text{NOT } (P_1) \text{ OR } \text{NOT}(P_2) \text{ OR } \ldots \text{ OR } \text{NOT}(P_n) \text{ OR } Q \quad (3)$$

o de la forma:

$$\text{NOT } (P_1) \text{ OR } \text{NOT}(P_2) \text{ OR } \ldots \text{ OR } \text{NOT}(P_n) \quad (4)$$

La cláusula Horn (3) puede transformarse en esta cláusula:

$$P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_n \Rightarrow Q \quad (5)$$

que se escribe en Datalog como esta regla:

$$Q :- P_1, P_2, \ldots, P_n. \quad (6)$$

La cláusula Horn (4) puede transformarse en:

$$P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_n \Rightarrow \quad (7)$$

que en Datalog se escribe de este modo:

$$P_1, P_2, \ldots, P_n. \quad (8)$$

Una regla Datalog, como en (6), es, por tanto, una cláusula Horn, y su significado, basado en la fórmula (5), es que si los predicados $P_1$ AND $P_2$ AND . . . AND $P_n$ son todos verdaderos para una asociación particular con sus argumentos variable, entonces $Q$ también es verdadera y puede, por tanto, inferirse. La expresión Datalog

(8) puede considerarse como una restricción de integridad, donde todos los predicados deben ser verdaderos para satisfacer la consulta.

En general, una consulta en Datalog tiene dos componentes:

- Un programa Datalog, que es un conjunto finito de reglas.
- Un literal $P(X_1, X_2, ..., X_n)$, donde cada $X_i$ es una variable o una constante.

Un sistema Prolog o Datalog tiene un **motor de inferencia** interno que puede utilizarse para procesar y calcular los resultados de dichas consultas. Los motores de inferencia de Prolog normalmente devuelven un resultado a la consulta (es decir, un conjunto de valores para las variables de la misma) en un momento dato y deben consultarse para devolver resultados adicionales. Por el contrario, Datalog devuelve resultados *set-at-a-time*.

### 24.4.5 Interpretaciones de las reglas

Hay dos alternativas principales para interpretar el significado teórico de las reglas: *proof-theoretic* y *model-theoretic*. En los sistemas prácticos, el mecanismo de inferencia dentro de un sistema define la interpretación exacta, que no puede coincidir con las dos interpretaciones teóricas. El mecanismo de inferencia es un procedimiento computacional y, por tanto, proporciona una interpretación computacional del significado de las reglas. En esta sección primero explicamos las dos interpretaciones teóricas. Después, explicamos brevemente los mecanismos de inferencia como un forma de definir el significado de las reglas. En la interpretación ***proof-theoretic*** de las reglas, consideramos los hechos y las reglas como sentencias verdaderas, o **axiomas**. Los **axiomas *ground*** no contienen variables. Los hechos son axiomas ground que se dan para ser ciertos. Las reglas se denominan **axiomas deductivos**, puesto que se pueden utilizar para deducir hechos nuevos. Los axiomas deductivos se pueden utilizar para construir pruebas que derivan hechos nuevos a partir de hechos existentes. Por ejemplo, la Figura 24.12 muestra cómo probar el hecho SUPERIOR(eduardo, luis) a partir de las reglas dadas en la Figura 24.11. La interpretación *proof-theoretic* nos ofrece un método procedimental o computacional para calcular una respuesta a la consulta Datalog. El proceso de demostrar si un determinado hecho (teorema) se cumple se conoce como *comprobación del teorema.*

El segundo tipo de interpretación se denomina interpretación ***model-theoretic***. Con ella, dado un dominio finito o infinito de valores constantes,[29] asignamos a un predicado cada posible combinación de valores a modo de argumentos. Debemos determinar entonces si el predicado es verdadero o falso. En general, es suficiente especificar las combinaciones de argumentos que hacen que el predicado sea cierto, y declarar todas las demás combinaciones que lo hacen falso. Si esto se hace para cada predicado, se denomina **interpretación** del conjunto de predicados. Por ejemplo, considere la interpretación de la Figura 24.13 para los predicados SUPERVISAR y SUPERIOR. Esta interpretación asigna un valor de veracidad (verdadero o falso) a cada posible combinación de valor de argumento (a partir de un dominio finito) para los dos predicados.

**Figura 24.12.** Comprobación de un hecho nuevo.

| | |
|---|---|
| 1. SUPERIOR(*X*, *Y* ) :– SUPERVISAR(*X*, *Y* ). | (regla 1) |
| 2. SUPERIOR(*X*, *Y* ) :– SUPERVISAR(*X*, *Z* ), SUPERIOR(*Z*, *Y*). | (regla 2) |
| 3. SUPERVISAR(juana, luis). | (axioma *ground*, dado) |
| 4. SUPERVISAR(eduardo, juana). | (axioma *ground*, dado) |
| 5. SUPERIOR(juana, luis). | (aplicar regla 1 en 3) |
| 6. SUPERIOR(eduardo, luis). | (aplicar regla 2 en 4 y 5) |

[29] Lo más normal es elegir un dominio finito, que se conoce con el nombre de *Herbrand Universe*.

**Figura 24.13.** Una interpretación que es un modelo mínimo.

**Reglas**

SUPERIOR(*X*, *Y* ) :– SUPERVISAR(*X*, *Y* ).
SUPERIOR(*X*, *Y* ) :– SUPERVISAR(*X*, *Z* ), SUPERIOR(*Z*, *Y* ).

**Interpretación**

*Hechos conocidos:*

SUPERVISAR(alberto, josé) es **verdadero**.
SUPERVISAR(alberto, fernando) es **verdadero**.
SUPERVISAR(alberto, aurora) es **verdadero**.
SUPERVISAR(juana, alicia) es **verdadero**.
SUPERVISAR(juana, luis) es **verdadero**.
SUPERVISAR(eduardo, alberto) es **verdadero**.
SUPERVISAR(eduardo, juana) es **verdadero**.
SUPERVISAR(*X*, *Y* ) es **falso** para todas las demás combinaciones (*X*, *Y* ) posibles.

*Hechos derivados:*

SUPERIOR(alberto, josé) es **verdadero**.
SUPERIOR(alberto, fernando) es **verdadero**.
SUPERIOR(alberto, aurora) es **verdadero**.
SUPERIOR(juana, alicia) es **verdadero**.
SUPERIOR(juana, luis) es **verdadero**.
SUPERIOR(eduardo, alberto) es **verdadero**.
SUPERIOR(eduardo, juana) es **verdadero**.
SUPERIOR(eduardo, josé) es **verdadero**.
SUPERIOR(eduardo, fernando) es **verdadero**.
SUPERIOR(eduardo, aurora) es **verdadero**.
SUPERIOR(eduardo, alicia) es **verdadero**.
SUPERIOR(eduardo, luis) es **verdadero**.
SUPERIOR(*X*, *Y* ) es **falso** para todas las demás combinaciones (*X*, *Y* ) posibles.

Se dice que una interpretación es un **modelo** para un *conjunto específico de reglas* si estas reglas son *siempre verdaderas* bajo dicha interpretación; es decir, para cualesquiera valores asignados a las variables de las reglas, la cabecera de las reglas es verdadera cuando sustituimos los valores de veracidad asignados a los predicados en el cuerpo de la regla por esa interpretación. Por tanto, siempre que se aplica una sustitución particular a las variables de las reglas, si todos los predicados del cuerpo de una regla son verdaderos bajo la interpretación, el predicado de la cabecera de la regla también debe ser verdadero. La interpretación mostrada en la Figura 24.13 es un modelo para las dos reglas mostradas, puesto que nunca puede provocar que las reglas sean violadas. Una regla se viola si una vinculación de constantes a las variables hace que todos los predicados del cuerpo de la regla sean ciertos, pero provoca que el predicado de la cabecera de la regla sea falso. Por ejemplo, si SUPERVISAR(*a, b*) y SUPERIOR(*b, c*) son los dos verdaderos bajo una determinada interpretación, pero SUPERIOR(*a, c*) no es verdadero, la interpretación no puede ser un modelo para la regla recursiva:

SUPERIOR(*X*, *Y*) :– SUPERVISAR(*X*, *Z*), SUPERIOR(*Z*, *Y*)

En el método *model-theoretic*, el significado de las reglas se establece proporcionando un modelo para todas las reglas. Un modelo es un **modelo mínimo** para un conjunto de reglas si no podemos cambiar ningún hecho de verdadero a falso y todavía obtenemos un modelo para estas reglas. Por ejemplo, tengamos en cuenta la interpretación de la Figura 24.13 y asumamos que el predicado SUPERVISAR está definido por un conjunto de hechos conocidos, mientras que el predicado SUPERIOR está definido como una interpretación (modelo) para las reglas. Supongamos que queremos añadir el predicado SUPERIOR(eduardo, roberto) a los predicados verdaderos. Sigue siendo un modelo para las reglas mostradas, pero no es un modelo mínimo, puesto que al cambiar el valor de veracidad de SUPERIOR(eduardo,roberto) de verdadero a falso todavía nos proporciona un modelo para las reglas. El modelo mostrado en la Figura 24.13 es el modelo mínimo para el conjunto de hechos que están definidos por el predicado SUPERVISAR.

En general, el modelo mínimo que corresponde a un conjunto dado de hechos en la interpretación *model-theoretic* debe ser el mismo que los hechos generados por la interpretación *proof-theoretic* para el mismo conjunto original de axiomas *ground* y deductivos. Sin embargo, esto sólo es generalmente verdadero para las reglas con una estructura sencilla. Una vez que permitimos la negación en la especificación de reglas, la correspondencia entre interpretaciones *no se mantiene*. De hecho, con la negación, son posibles numerosos modelos mínimos para un conjunto dado de hechos.

Un tercer método para interpretar el significado de las reglas implica la definición de un mecanismo de inferencia que el sistema utiliza para deducir los hechos a partir de las reglas. Este mecanismo de inferencia definiría una **interpretación computacional** del significado de las reglas. El lenguaje de programación lógica Prolog utiliza este mecanismo de inferencia para definir el significado de las reglas y los hechos en un programa Prolog. No todos los programas Prolog se corresponden con las interpretaciones *proof-theoretic* o *model-theoretic*; depende del tipo de reglas del programa. Sin embargo, para muchos programas Prolog sencillos, el mecanismo de inferencia de Prolog infiere los hechos que corresponden a la interpretación *proof-theoretic* o a un modelo mínimo bajo la interpretación *model-theoretic*.

### 24.4.6 Programas Datalog y su seguridad

Hay dos métodos principales para definir los valores de veracidad de los predicados en los programas Datalog actuales. Los **predicados definidos por el hecho** (o **relaciones**) se definen listando todas las combinaciones de valores (las tuplas) que constituyen el predicado verdadero. Se corresponden con las relaciones base cuyo contenido está almacenado en un sistema de bases de datos. La Figura 24.14 muestra los predicados definidos por hechos EMPLEADO, VARÓN, MUJER, DEPARTAMENTO, SUPERVISAR, PROYECTO y TRABAJA_EN, que equivalen a parte de la base de datos relacional de la Figura 5.6. Los **predicados definidos por reglas** (o **vistas**) se definen siendo la cabecera (LHS) de una o más reglas Datalog; se corresponden con *relaciones virtuales* cuyos contenidos pueden ser inferidos por el motor de inferencia. La Figura 24.15 muestra varios predicados definidos por reglas.

Se dice que un programa o una regla es **segura** si genera un conjunto *finito* de hechos. El problema teórico general de determinar si un conjunto de reglas es seguro no se puede decidir. Sin embargo, podemos determinar la seguridad de las formas restringidas de las reglas. Por ejemplo, las reglas de la Figura 24.16 son seguras. Una situación en la que obtenemos reglas inseguras que puede generar una cantidad infinita de hechos es cuando una de las variables de la regla puede moverse por un dominio infinito de valores, y esta variable no está limitada a un rango por una relación finita. Por ejemplo, consideremos la siguiente regla:

GRAN_SUELDO(*Y*) :– *Y* > 60000

Aquí, podemos obtener un resultado infinito si *Y* se mueve por todos los enteros posibles. Pero supongamos que cambiamos la regla de este modo:

GRAN_SUELDO(*Y*) :– EMPLEADO(*X*), Sueldo(*X*, *Y*), *Y* > 60000

**Figura 24.14.** Predicados de hechos para parte de la base de datos de la Figura 5.6.

```
EMPLEADO(josé).
EMPLEADO(alberto).
EMPLEADO(alicia).
EMPLEADO(juana).
EMPLEADO(fernando).
EMPLEADO(aurora).
EMPLEADO(luis).
EMPLEADO(eduardo).

SUELDO(josé, 30000).
SUELDO(alberto, 40000).
SUELDO(alicia, 25000).
SUELDO(juana, 43000).
SUELDO(fernando, 38000).
SUELDO(aurora, 25000).
SUELDO(luis, 25000).
SUELDO(eduardo, 55000).

DEPARTAMENTO(josé, investigación).
DEPARTAMENTO(alberto, investigación).
DEPARTAMENTO(alicia, administración).
DEPARTAMENTO(juana, administración).
DEPARTAMENTO(fernando, investigación).
DEPARTAMENTO(aurora, investigación).
DEPARTAMENTO(luis, administración).
DEPARTAMENTO(eduardo, sedecentral).

SUPERVISAR(alberto, josé).
SUPERVISAR(alberto, fernando).
SUPERVISAR(alberto, aurora).
SUPERVISAR(juana, alicia).
SUPERVISAR(juana, luis).
SUPERVISAR(eduardo, alberto).
SUPERVISAR(eduardo, juana).

VARÓN(josé).
VARÓN(alberto).
VARÓN(fernando).
VARÓN(luis).
VARÓN(eduardo).

MUJER(alicia).
MUJER(juana).
MUJER(aurora).

PROYECTO(productox).
PROYECTO(productoy).
PROYECTO(productoz).
PROYECTO(computación).
PROYECTO(reorganización).
PROYECTO(comunicaciones).

TRABAJA_EN(josé, productox, 32).
TRABAJA_EN(josé, productoy, 8).
TRABAJA_EN(fernando, productoz, 40).
TRABAJA_EN(aurora, productox, 20).
TRABAJA_EN(aurora, productoy, 20).
TRABAJA_EN(alberto, productoy, 10).
TRABAJA_EN(alberto, productoz, 10).
TRABAJA_EN(alberto, computación, 10).
TRABAJA_EN(alberto, reorganización, 10).
TRABAJA_EN(alicia, comunicaciones, 30).
TRABAJA_EN(alicia, computación, 10).
TRABAJA_EN(luis, computación, 35).
TRABAJA_EN(luis, comunicaciones, 5).
TRABAJA_EN(juana, comunicaciones, 20).
TRABAJA_EN(juana, reorganización, 15).
TRABAJA_EN(eduardo, reorganización, 10).
```

En la segunda regla, el resultado no es finito, puesto que los valores a los que *Y* puede ajustarse están ahora restringidos a los valores que son el sueldo de algunos empleados de la base de datos; probablemente, un conjunto finito de valores. También podemos reescribir la regla de esta forma:

```
GRAN_SUELDO(Y) :– Y>60000, EMPLEADO(X), Sueldo(X, Y)
```

En este caso, la regla todavía es teóricamente segura. Sin embargo, en Prolog o en cualquier otro sistema que utilice un mecanismo de inferencia *top-down* (de arriba hacia abajo), *depth-first* (primero en profundidad), la regla crea un bucle infinito, pues primero buscamos un valor para *Y* y después comprobamos si es un sueldo

**Figura 24.15.** Predicados definidos por reglas.

SUPERIOR(*X*, *Y*) :– SUPERVISAR(*X*, *Y*).
SUPERIOR(*X*, *Y*) :– SUPERVISAR(*X*, *Z*), SUPERIOR(*Z*, *Y*).

SUBORDINADO(*X*, *Y*) :– SUPERIOR(*Y*, *X*).

SUPERVISOR(*X*) :– EMPLEADO(*X*), SUPERVISAR(*X*, *Y*).

MÁS_40K_EMP(*X*) :– EMPLEADO(*X*), SUELDO(*X*, *Y*), *Y* > = 40000.
MENOS_40K_SUPERVISOR( *X*) :– SUPERVISOR(*X*), NOT(MÁS_40_K_EMP(*X*)).
PRINCIPAL_PRODUCTOX_EMP( *X*) :– EMPLEADO(*X*), TRABAJA_EN(*X*, productox, *Y*), *Y* > = 20.
PRESIDENTE(*X*) :– EMPLEADO(*X*), NOT(SUPERVISAR(*Y*, *X*) ).

de un empleado. El resultado es la generación de un número infinito de valores *Y*, aunque éstos, después de un determinado punto, no pueden conducir a un conjunto de predicados RHS verdaderos. Una definición de Datalog considera que las dos reglas son seguras, porque no depende de un mecanismo de inferencia particular. Aun así, es generalmente aconsejable escribir semejante regla de la forma más segura, con los predicados que restringen posibles uniones de variables en primer lugar. Otro ejemplo de regla insegura es el siguiente:

TIENE_ALGO(*X*, *Y*) :– EMPLEADO(*X*)

Aquí, puede generarse una vez más una cantidad infinita de valores *Y*, puesto que la variable *Y* sólo aparece en la cabecera de la regla y, por tanto, no está limitada a un conjunto finito de valores. Para definir reglas seguras más formalmente, utilizamos el concepto de variable limitada. Una variable *X* está **limitada** en una regla si (1) aparece en un predicado regular (no incorporado) en el cuerpo de una regla; (2) aparece en un predicado de la forma $X = c$ o $c = X$ o ($c_1 <= X$ y $X <= c_2$) en el cuerpo de la regla, donde $c$, $c_1$ y $c_2$ son valores constantes; o (3) aparece en un predicado de la forma $X = Y$ o $Y = X$ en el cuerpo de la regla, donde *Y* es una variable limitada. Se dice que una regla es **segura** si todas sus variables están limitadas.

### 24.4.7 Uso de operaciones relacionales

Es fácil especificar muchas operaciones del álgebra relacional según las reglas Datalog que definen el resultado de aplicar esas operaciones en las relaciones de la base de datos (predicados de hechos). Esto significa que las consultas relacionales y las vistas pueden especificarse fácilmente en Datalog. La potencia adicional que Datalog proporciona está en la especificación de consultas recursivas, y en las vistas basadas en las consultas recursivas. En esta sección mostramos cómo pueden especificarse algunas de las operaciones relacionales estándar como reglas Datalog. Nuestros ejemplos utilizarán las relaciones base (predicados definidos por hechos) REL_UNA, REL_DOS y REL_TRES, cuyos esquemas se muestran en la Figura 24.16. En Datalog, no tenemos que especificar los nombres de atributos como en la Figura 24.16; en su lugar, el grado de cada predicado es el aspecto más importante. En un sistema práctico, el dominio (tipo de datos) de cada atributo también es importante para operaciones como UNION, INTERSECTION y JOIN, y asumimos que los tipos de atributos son compatibles para las distintas operaciones, como explicamos en el Capítulo 5.

La Figura 24.16 ilustra varias operaciones relacionales básicas. Si el modelo Datalog está basado en el modelo relacional y, por tanto, asume que los predicados (las relaciones de hechos y los resultados de las consultas) especifican conjuntos de tuplas, la tuplas duplicadas en el mismo predicado se eliminan automáticamente. Esto puede o no ser verdadero, en función del motor de inferencia Datalog. Sin embargo, definitivamente no es el caso de Prolog, por lo que cualquiera de las reglas de la Figura 24.16 que implican la eliminación de duplicados no son correctas en Prolog. Por ejemplo, si queremos especificar reglas Prolog para la operación UNION con eliminación de duplicados, debemos reescribirlas de este modo:

UNION_UNA_DOS(*X*, *Y*, *Z*) :– REL_UNA(*X*, *Y*, *Z*).

UNION_UNA_DOS(*X*, *Y*, *Z*) :– REL_DOS(*X*, *Y*, *Z*), NO(REL_UNA(*X*, *Y*, *Z*)).

No obstante, las reglas de la Figura 24.16 deben funcionar para Datalog, si se eliminan automáticamente los duplicados. De forma parecida, las reglas para la operación PROYECTO de la Figura 24.16 deben funcionar para Datalog en este caso, pero no son correctas para Prolog, puesto que en el último caso aparecen duplicados.

## 24.4.8 Evaluación de consultas Datalog no recursivas

A fin de utilizar Datalog como sistema de bases de datos deductivo, es adecuado definir un mecanismo de inferencia basado en los conceptos de procesamiento de consultas de bases de datos relacionales. La estrategia inherente implica una evaluación de abajo hacia arriba, empezando con las relaciones base; el orden de las operaciones es flexible y está sujeto a la optimización de la consulta. En esta sección explicamos un mecanismo de inferencia basado en las operaciones relacionales que se pueden aplicar a las consultas Datalog **no recursivas**. Utilizamos el hecho y la regla base de las Figuras 24.14 y 24.15 para ilustrar nuestra explicación.

Si una consulta sólo implica predicados definidos por hechos, la inferencia se convierte en una de búsqueda entre los hechos para el resultado de la consulta. Por ejemplo, una consulta como la siguiente:

DEPARTAMENTO(*X*, Investigación)?

es una selección de todos los empleados que se llaman *X* y que trabajan para el departamento de Investigación. En el álgebra relacional, es la consulta:

$\pi_{\$1}\ (\sigma_{\$2 = \text{“Investigación”}}\ (\text{DEPARTAMENTO}))$

**Figura 24.16.** Predicados para ilustrar operaciones relacionales.

REL_UNA(*A*, *B*, *C*).
REL_DOS(*D*, *E*, *F* ).
REL_TRES(*G*, *H*, *I*, *J* ).

SELECCIONAR_UNA_A_IGUAL_C(*X*, *Y*, *Z* ) :– REL_UNA(*C*, *Y*, *Z*).
SELECCIONAR_UNA_B_MENOR_5(*X*, *Y*, *Z* ) :– REL_UNA(*X*, *Y*, *Z* ), *Y*< 5.
SELECCIONAR_UNA_A_IGUAL_C_Y_B_MENOR_5(*X*, *Y*, *Z* ) :– REL_UNA(*C*, *Y*, *Z*), *Y*<5

SELECCIONAR_UNA_A_IGUAL_C_O_B_MENOR_5(*X*, *Y*, *Z* ) :– REL_UNA(*C*, *Y*, *Z*).
SELECCIONAR_UNA_A_IGUAL_C_O_B_MENOR_5(*X*, *Y*, *Z* ) :– REL_UNA(*X*, *Y*, *Z* ), *Y*<5.

PROYECTO_TRES_EN_G_H(*W*, *X* ) :– REL_TRES(*W*, *X*, *Y*, *Z* ).

UNION_UNA_DOS(*X*, *Y*, *Z* ) :– REL_UNA(*X*, *Y*, *Z* ).
UNION_UNA_DOS(*X*, *Y*, *Z* ) :– REL_DOS(*X*, *Y*, *Z* ).

INTERSECTAR_UNA_DOS(*X*, *Y*, *Z* ) :– REL_UNA(*X*, *Y*, *Z* ), REL_DOS(*X*, *Y*, *Z* ).

DIFERENCIA_DOS_UNA(*X*, *Y*, *Z* ) :– REL_DOS(*X*, *Y*, *Z* ) NO(REL_UNA(*X*, *Y*, *Z* ).

PROD CART_UNA_TRES(*T*, *U*, *V*, *W*, *X*, *Y*, *Z* ) :–
REL_UNA(T, U, V), REL_TRES(*W*, *X*, *Y*, *Z* ).

UNION_NATURAL_UNA_TRES_C_IGUAL_G(*U*, *V*, *W*, *X*, *Y*, *Z* ) :–
REL_UNA(*U*, *V*, *W*), REL_TRES(*W*, *X*, *Y*, *Z* ).

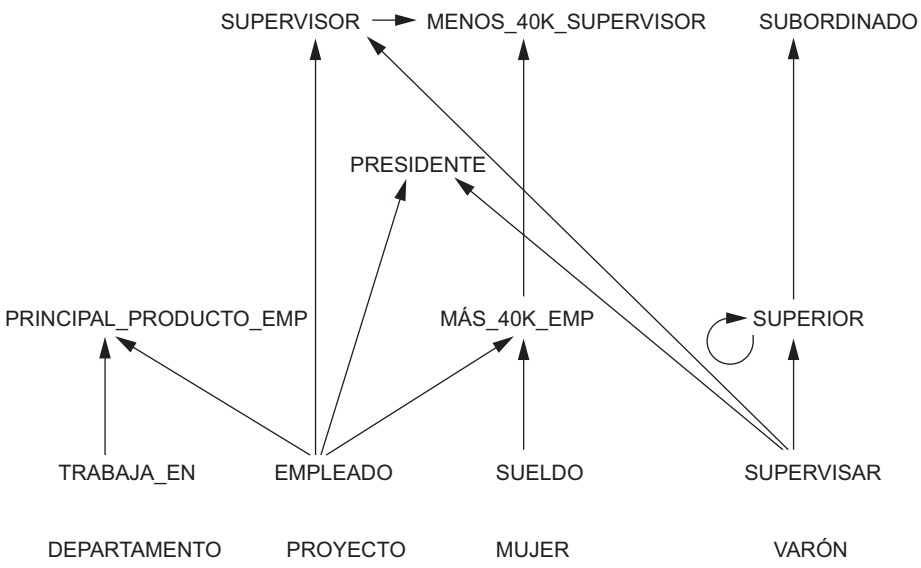
que puede contestarse buscando a través del predicado definido por hechos departamento($X$,$Y$). La consulta implica las operaciones SELECT y PROJECT relacionales en una relación base, y puede manipularse mediante las técnicas de optimización y procesamiento de consultas explicadas en el Capítulo 15.

Cuando una consulta implica predicados definidos por reglas, el mecanismo de inferencia debe calcular el resultado basado en las definiciones de reglas. Si una consulta no es recursiva e implica un predicado $p$ que aparece como la cabecera de una regla $p :- p_1, p_2, ..., p_n$, la estrategia es calcular primero las relaciones correspondientes a $p_1, p_2, ..., p_n$ y después calcular la relación correspondiente a $p$. Es útil hacer un seguimiento de la dependencia entre los predicados de una base de datos deductiva mediante un **gráfico de dependencia de predicados**. La Figura 24.17 muestra el gráfico para los predicados de hechos y reglas mostrados en las Figuras 24.14 y 24.15. El gráfico de dependencia contiene un **nodo** por cada predicado. Siempre que se especifica un predicado $A$ en el cuerpo (RHS) de una regla, y la cabecera de esa regla (LHS) es el predicado $B$, decimos que $B$ **depende de** $A$, y dibujamos un borde tendente de $A$ a $B$. Esto indica que, a fin de calcular los hechos para el predicado $B$ (la cabecera de la regla), primero debemos calcular los hechos para todos los predicados $A$ del cuerpo de la regla. Si el gráfico de dependencia no tiene ciclos, decimos que la regla **no es recursiva**. Si existe al menos un ciclo, decimos que la regla es **recursiva**. En la Figura 24.17, hay un predicado definido recursivamente (en concreto, SUPERIOR) que tiene un borde recursivo apuntando a sí mismo. Además, como el subordinado del predicado depende de SUPERIOR, también necesita la recursividad para calcular su resultado.

Una consulta que sólo incluye predicados no recursivos se denomina **consulta no recursiva**. En esta sección sólo explicamos los mecanismos de inferencia para las consultas no recursivas. En la Figura 24.17, cualquier consulta que no implique predicados subordinados o superiores no es recursiva. En el gráfico de dependencia de predicados, los nodos correspondientes a los predicados definidos por hechos no tienen ningún borde entrante, puesto que todos los predicados definidos por hechos tienen sus hechos almacenados en una relación de base de datos. El contenido de un predicado definido por hechos se puede calcular directamente recuperando las tuplas de la relación de base de datos correspondiente.

La función principal de un mecanismo de inferencia es calcular los hechos correspondientes a los predicados de la consulta. Esto se puede acometer generando una **expresión relacional** que implique operadores relacionales como SELECT, PROJECT, JOIN, UNION y SET DIFFERENCE (preocupándose adecuadamente de los problemas de seguridad) que, al ejecutarse, proporcionen el resultado de la consulta. La consulta puede

**Figura 24.17.** Gráfico de dependencia de predicados para las Figuras 24.14 y 24.15.



ejecutarse después utilizando las operaciones internas de procesamiento y optimización de un sistema de gestión de bases de datos relacional. Siempre que el mecanismo de inferencia tenga que calcular el conjunto de hechos correspondiente a un predicado definido por reglas no recursivo *p*, primero localiza todas las reglas que tienen *p* como cabecera. La idea es calcular el conjunto de hechos para cada regla semejante y después aplicar la operación UNION a los resultados, puesto que UNION equivale a una operación OR lógica. El gráfico de dependencia indica todos los predicados *q* de los que depende cada *p*, y como asumimos que el predicado no es recursivo, siempre podemos determinar un orden parcial entre dichos predicados *q*. Antes de calcular el conjunto de hechos para *p*, primero calculamos los conjuntos de hechos para todos los predicados *q* de los que depende *p*, basándonos en su orden parcial. Por ejemplo, si una consulta implica el predicado MENOS_40K_SUPERVISOR, primero debemos calcular tanto el supervisor como MÁS_40K_EMP. Como los dos últimos sólo dependen de los predicados definidos por hechos EMPLEADO, SUELDO y SUPERVISAR, podemos calcularlos directamente a partir de las relaciones de bases de datos almacenadas.

Esto concluye nuestra introducción a las bases de datos deductivas. En el sitio web complementario del libro encontrará material adicional, así como el Capítulo 25 completo de la tercera edición. Esto incluye una explicación de los algoritmos para el procesamiento de consultas recursivas.

# 24.5 Resumen

Este capítulo es una introducción de algunas de las características comunes que las aplicaciones avanzadas necesitan: bases de datos activas, bases de datos temporales, y bases de datos espaciales y multimedia. Es importante saber que cada uno de estos tipos de bases de datos es un tema muy amplio que merece un libro entero.

Primero vimos el tema de las bases de datos activas, que ofrecen funcionalidades adicionales para especificar reglas activas. Hablamos del modelo ECA (evento-condición-acción) para las bases de datos activas. Los eventos que se producen (por ejemplo, la actualización de la base de datos) pueden disparar automáticamente las reglas, además de iniciar determinadas acciones que se hubieran especificado en la declaración de la regla, siempre y cuando se cumplan ciertas condiciones. Muchos paquetes comerciales ya cuentan con parte de la funcionalidad que ofrecen las bases de datos activas, en forma de *triggers*. También hemos explicado las diferentes opciones que hay para especificar las reglas (a nivel de fila frente a nivel de sentencia, antes frente a después, inmediata frente a diferida). Asimismo, hemos puesto algunos ejemplos de *triggers* a nivel de fila en el sistema Oracle comercial, y de reglas a nivel de sentencia en el sistema experimental STARBURST. También hemos explicado la sintaxis de los *triggers* en el estándar SQL-99, así como algunos problemas de diseño y algunas posibles aplicaciones para las bases de datos activas.

A continuación hemos introducido algunos conceptos de las bases de datos temporales, que permiten al sistema de bases de datos almacenar un historial de los cambios, de modo que el usuario puede consultar estados pasados y el estado actual de la base de datos. Hemos explicado cómo podemos representar el tiempo y distinguir entre las dimensiones de tiempo válido y tiempo de transacción. Hemos visto cómo implementar el tiempo válido, el tiempo de transacción y las relaciones bitemporales utilizando el versionado de tuplas en el modelo relacional, con ejemplos para ilustrar la implementación de actualizaciones, inserciones y eliminaciones. También hemos mostrado cómo podemos utilizar los objetos complejos para implementar bases de datos de tiempo (temporales) utilizando el versionado de atributos. Asimismo, hemos visto algunas operaciones de consulta con las bases de datos relacionales temporales y hemos ofrecido una breve introducción al lenguaje TSQL2.

Después pasamos a las bases de datos espaciales y multimedia. Las primeras ofrecen conceptos para las bases de datos que hacen un seguimiento de los objetos que tienen características espaciales, y requiere modelos para representar esas características espaciales y operadores para poder compararlas y manipularlas. Las bases de datos multimedia proporcionan características que permiten a los usuarios almacenar y consultar diferentes tipos de información multimedia, lo que incluye imágenes (fotografías o dibujos), clips de vídeo (pelícu-

las, noticiarios o vídeos caseros), clips de audio (sonidos, mensajes telefónicos o discursos) y documentos (libros o artículos). También hemos ofrecido una breve visión general de los distintos orígenes de medios y de cómo los orígenes multimedia se pueden indexar. Si desea más información sobre las bases de datos multimedia y la gestión de datos espaciales en GIS, puede consultar el Capítulo 30.

Concluimos el capítulo con una introducción a las bases de datos deductivas y Datalog.

## Preguntas de repaso

**24.1.** ¿Cuáles son las diferencias entre las reglas activas a nivel de fila y a nivel de sentencia?

**24.2.** ¿Cuáles son las diferencias entre *consideración* inmediata, diferida y aislada de las condiciones de las reglas activas?

**24.3.** ¿Cuáles son las diferencias entre *ejecución* inmediata, diferida y aislada de las acciones de las reglas activas?

**24.4.** Explique brevemente los problemas de la coherencia y la terminación al designar un conjunto de reglas activas.

**24.5.** Explique algunas aplicaciones de las bases de datos activas.

**24.6.** Explique cómo se representa el tiempo en las bases de datos de tiempo y compare las distintas dimensiones del tiempo.

**24.7.** ¿Cuáles son las diferencias entre tiempo válido, tiempo de transacción y relaciones bitemporales?

**24.8.** Describa cómo deben implementarse los comandos de inserción, eliminación y actualización en una relación de tiempo válida.

**24.9.** Describa cómo deben implementarse los comandos de inserción, eliminación y actualización en una relación bitemporal.

**24.10.** Describa cómo deben implementarse los comandos de inserción, eliminación y actualización en una relación de tiempo de transacción.

**24.11.** ¿Cuáles son las principales diferencias entre el versionado de tuplas y el versionado de atributos?

**24.12.** ¿En qué difieren las bases de datos espaciales de las bases de datos normales?

**24.13.** ¿Cuáles son los diferentes tipos de orígenes multimedia?

**24.14.** ¿Cómo se indexan los orígenes multimedia para una recuperación basada en el contenido?

## Ejercicios

**24.15.** Consideremos la base de datos EMPRESA descrita en la Figura 5.6. Utilizando la sintaxis de los *triggers* de Oracle, escriba reglas activas para hacer lo siguiente:

- **a.** Siempre que cambien las asignaciones de proyecto de un empleado, compruebe si el número total de horas por semana invertidas en los proyectos del empleado es inferior a 30 o superior a 40; si es así, notificar al supervisor directo del empleado.
- **b.** Siempre que se elimina un empleado, eliminar las tuplas PROYECTO y las tuplas SUBORDINADO relacionadas con ese empleado, y si el empleado dirige un departamento o supervisa a otros empleados, establecer DniDirector para ese departamento a NULL y establecer SuperDni para esos empleados a NULL.

**24.16.** Repita 24.15 pero utilice la sintaxis de las reglas activas de STARBURST.

**24.17.** Considerando el esquema relacional de la Figura 24.18, escriba reglas activas para que el atributo SumaComisiones de VENDEDORES sea igual a la suma del atributo Comisión de VENTAS para cada vendedor. Sus reglas también deberían comprobar si SumaComisiones excede de 100.000; si es así, llame a un procedimiento NotificarDirector(S_id). Escriba las reglas a nivel de sentencia en la notación STARBURST y a nivel de fila en Oracle.

**Figura 24.18.** Esquema de la base de datos para las ventas y las comisiones de los vendedores del Ejercicio 24.17.

**VENTAS**

| S_id | V_id | Comisión |
|---|---|---|

**VENDEDORES**

| Id_Vendedor | Nombre | Rango | Teléfono | SumaComisiones |
|---|---|---|---|---|

**24.18.** Según el esquema EER de UNIVERSIDAD de la Figura 4.10, escriba algunas reglas (en inglés) que puedan implementarse a través de reglas activas a fin de implementar algunas restricciones de integridad comunes que piense que son importantes para esta aplicación.

**24.19.** Explique cuáles de las actualizaciones que crearon cada una de las tuplas de la Figura 24.9 se aplicaron retroactivamente y cuáles se aplicaron proactivamente.

**24.20.** Demuestre cómo las actualizaciones siguientes, si se aplican en secuencia, modificarían el contenido de la relación EMP_BT bitemporal de la Figura 24.9. Para cada actualización, indique si se trata de una actualización retroactiva o proactiva.

**a.** A fecha 10-03-2004,17:30:00, el sueldo de Ojeda se actualizó a 40.000; efectivo a 01-03-2004.

**b.** A fecha 30-07-2003, 08:31:00, el sueldo de Pérez se corrigió para revelar que debería haberse introducido 31000 (en lugar de los 30000 que figuran), con vigencia desde 01-06-2003.

**c.** A fecha 18-03-2004, 08:31:00, se modificó la base de datos para indicar que Ojeda dejaba la empresa (es decir, se eliminó lógicamente) el día 31-03-2004.

**d.** A fecha 20-04-2004, 14:07:33, se modificó la base de datos para indicar la contratación de un nuevo empleado que se llama Torres, con la tupla <‘Torres’, ‘334455667’, 1, NULL >, haciéndose efectivo a fecha 2-04-2004.

**e.** A fecha 28-04-2004, 12:54:02, se modificó la base de datos para indicar que Campos dejaba la empresa (es decir, se eliminó lógicamente) el día 01-06-2004.

**f.** A fecha 05-05-2004, 13:07:33, se modificó la base de datos para indicar la nueva contratación de Lobato, con el mismo departamento y supervisor pero con un sueldo de 35.000, que entraba en vigor el 01-05-2004.

**24.21.** Demuestre cómo las actualizaciones del Ejercicio 24.20, si se aplican en secuencia, cambiarían el contenido de la relación EMP_TV de tiempo válido de la Figura 24.8.

**24.22.** Añada los siguientes hechos a la base de datos de ejemplo de la Figura 24.3:

SUPERVISAR(luis, roberto), SUPERVISAR(alberto, palacios).

Primero modifique el árbol administrativo de la Figura 24.1(b) para reflejar este cambio. Después, modifique el diagrama de la Figura 24.4 mostrando la evaluación de arriba abajo de la consulta SUPERIOR(eduardo, *Y*).

**24.23.** Tenga en cuenta el siguiente conjunto de hechos para la relación PADRE(*X*, *Y*), donde *Y* es el padre de *X*:

PADRE(a, aa), PADRE(a, ab), PADRE(aa, aaa), PADRE(aa, aab),

PADRE(aaa, aaaa), PADRE(aaa, aaab).

Considere las reglas:

*r*1: PREDECESOR(*X*, *Y*) :– PADRE(*X*, *Y*)

*r*2: PREDECESOR(*X*, *Y*) :– PADRE(*X*, *Z*), PREDECESOR(*Z*, *Y*)

que define el antepasado *Y* de *X* como arriba.

**a.** Demuestre cómo resolver la consulta Datalog.

PREDECESOR(aa, *X*)?

utilizando la estrategia sencilla. Muestre su trabajo paso a paso.

**b.** Presente la misma consulta calculando sólo los cambios en la relación predecesora y utilizándola cada vez en la regla 2.

[Esta pregunta proviene de Bancilhon y Ramakrishnan (1986).]

**24.24.** Considere una base de datos deductiva con las siguientes reglas:

PREDECESOR(*X*, *Y*) :– PADRE(*X*, *Y*)

PREDECESOR(*X*, *Y*) :– PADRE(*X*, *Z*), PREDECESOR(*Z*, *Y*)

PADRE(*X*, *Y*) significa que *Y* es el padre de *X*; PREDECESOR(*X*, *Y*) significa que *Y* es el antepasado de *X*.

Considere el siguiente hecho:

PADRE(Javier, Isaac), PADRE(Isaac, Juan), PADRE(Juan, Carlos).

**a.** Construya un modelo de interpretación teórica de las reglas anteriores utilizando los hechos dados.

**b.** Piense que una base de datos contiene las relaciones anteriores PADRE(*X*, *Y*), otra relación HERMANO(*X*, *Y*) y una tercera relación NACIMIENTO(*X*, *B*), donde *B* es la fecha de nacimiento de la persona *X*. Presente una regla que calcule el primo carnal según lo siguiente: sus padres deben ser hermanos.

**c.** Presente un programa completo en Datalog con literales basados en hechos y basados en reglas que calcule la siguiente relación: lista de pares de primos, en los que la primera persona haya nacido después de 1960 y la segunda después de 1970. Debe utiliza *mayor que* como un predicado integrado. (*Nota*: También debe presentar muestras de hechos para hermano, nacimiento y persona.)

**24.25.** Considere las siguientes reglas:

ALCANZABLE(*X*, *Y*) :– VUELO(*X*, *Y*)

ALCANZABLE(*X*, *Y*) :– VUELO(*X*, *Z*), ALCANZABLE(*Z*, *Y*)

donde ALCANZABLE(*X*, *Y*) significa que es posible alcanzar la ciudad *Y* desde la ciudad *X*, y VUELO(*X*, *Y*) significa que hay un vuelo a la ciudad *Y* desde la ciudad *X*.

**a.** Construya predicados de hechos que describan lo siguiente:

i. Los Ángeles, Nueva York, Chicago, Atlanta, Frankfurt, París, Singapur y Sydney son ciudades.

ii. Existen los siguientes vuelos: LA a NY, NY a Atlanta, Atlanta a Frankfurt, Frankfurt a Atlanta, Frankfurt a Singapur y Singapur a Sydney. (*Nota*: No podemos asumir automáticamente ningún vuelo en la dirección contraria.)

**b.** ¿Son cíclicos los datos dados? Si es así, ¿en qué sentido?

**c.** Construya un modelo de interpretación teórica (es decir, una interpretación parecida a la mostrada en la Figura 25.3) de los hechos y reglas anteriores.

**d.** Considere esta consulta:

ALCANZABLE(Atlanta, Sydney)?

¿Cómo se ejecutará esta consulta utilizando la evaluación *naive* (sencilla) y *seminaive* (semisencilla)? Enumere las series de pasos a seguir.

**e.** Considere los siguientes predicados definidos por reglas:

IDA-VUELTA-ALCANZABLE(*X*, *Y*) :–

ALCANZABLE(*X*, *Y*), ALCANZABLE(*Y*, *X*)

DURACIÓN(*X*, *Y*, *Z*)

Dibuje un gráfico de dependencia de predicados para lo predicados anteriores. (*Nota*: DURACIÓN(*X*, *Y*, *Z*) significa que es posible tomar un vuelo de *X* a *Y* en *Z* horas.)

**f.** Considere la siguiente consulta: ¿Qué ciudades se pueden alcanzar en 12 horas desde Atlanta? Expréselo en Datalog. Asuma que existen predicados del tipo mayor-que(*X*, *Y*). ¿Puede convertir esto en una sentencia de álgebra relacional de una forma directa? ¿Por qué o por qué no?

**g.** Considere el predicado población(*X*, *Y*) donde *Y* es la población de la ciudad *X*. Considere la siguiente consulta: enumere todas las uniones posibles del par de predicado (*X*, *Y*), donde *Y* es una ciudad que puede alcanzarse en dos vuelos desde la ciudad *X*, que tiene más de 1 millón de personas. Presente esta consulta en Datalog. Trace el árbol de consulta correspondiente en términos algebraicos relacionales.

## Bibliografía seleccionada

El libro Zaniolo y otros (1997) costa de varias partes, cada una de las cuales describe un concepto avanzado de bases de datos, como las bases de datos activas, temporales y espaciales/multimedia. Widom y Ceri (1996) y Ceri y Fraternali (1997) se centran en los conceptos y sistemas de bases de datos activas. Snodgrass y otros (1995) describe el lenguaje TSQL2 y el modelo de datos. Khoshafian y Baker (1996), Faloutsos (1996), y Subrahmanian (1998) describen los conceptos relativos a las bases de datos multimedia. Tansel y otros (1992) es una colección de capítulos sobre bases de datos temporales.

Las reglas de STARBURST se describen en Widom y Finkelstein (1990). Los primeros trabajos sobre bases de datos activas incluían el proyecto HiPAC, que se explica en Chakravarthy y otros (1989) y en Chakravarthy (1990). En Jensen y otros (1994) encontrará un glosario sobre las bases de datos temporales. Snodgrass (1987) se centra en TQuel, un antiguo lenguaje de consulta temporal. La normalización temporal se define en Navathe y Ahmed (1989). Paton (1999) y Paton y Díaz (1999) analizan las bases de datos activas. Chakravarthy y otros (1994) describen SENTINEL y los sistemas activos basados en objetos. Lee y otros (1998) explica la gestión de las series de tiempo.

Los primeros desarrollos sobre lógica y bases de datos se examinaron en Gallaire y otros (1984). Reiter (1984) ofrece una reconstrucción de la teoría de bases de datos relacionales, mientras que Levesque (1984) proporciona una explicación del conocimiento incompleto a la luz de la lógica. Gallaire y Minker (1978) es un antiguo libro sobre este tema. Ullman (1989, Volumen 2) ofrece un tratamiento detallado de la lógica y las bases de datos, y en el Volumen 1 (1988) hay un capítulo relacionado. Ceri, Gottlob y Tanca (1990) presenta un estudio global y conciso de la lógica y las bases de datos. Das (1992) es un libro sobre bases de datos deductivas y programación lógica. La historia de Datalog se cuenta en Maier y Warren (1988). Clocksin y Mellish (1994) es una referencia excelente sobre el lenguaje Prolog.

Aho y Ullman (1979) proporciona un antiguo algoritmo para tratar con consultas recursivas, utilizando el operador de punto fijo mínimo. Bancilhon y Ramakrishnan (1986) ofrece una descripción excelente y detallada de los métodos de procesamiento de consultas recursivas, con ejemplos detallados de los métodos *naive* y *seminaive*. Warren (1992) y Ramakrishnan y Ullman (1993) son unos estudios excelentes sobre las bases de datos deductivas y el procesamiento de consultas recursivas. Bancilhon (1985) ofrece una completa descripción del método *seminaive* basado en el álgebra relacional. Otros métodos para procesar las consultas recursivas son la estrategia consulta/subconsulta recursiva de Vieille (1986), que es una estrategia interpretada de arriba hacia abajo, y la estrategia iterativa compilada de arriba abajo de Henschen-Naqvi (1984). Balbin y Rao (1987) explica una extensión del método diferencial *seminaive* para predicados múltiples.

El estudio original sobre los conjuntos mágicos es Bancilhon y otros (1986). Beeri y Ramakrishnan (1987) lo extiende. Mumick y otros (1990) muestra la aplicabilidad de los conjuntos mágicos a las consultas SQL anidadas no recursivas. En Vieille (1986, 1987) aparecen otros métodos para optimizar las reglas sin tener que reescribirlas. Kifer y Lozinskii (1986) propone una técnica diferente. Bry (1990) explica cómo pueden reconciliarse los métodos arriba-abajo y abajo-arriba. Whang y Navathe (1992) describe una técnica de forma normal disjunta extendida para tratar con la recursividad en las expresiones de álgebra relacional, a fin de proporcionar una interfaz de sistema experto sobre un DBMS relacional.

Chang (1981) describe un sistema antiguo que combina las reglas deductivas con las bases de datos relacionales. El prototipo de sistema LDL se describe en Chimenti y otros (1990). Krishnamurthy y Naqvi (1989) ofrece una introducción a la noción *choice* en LDL. Zaniolo (1988) explica los problemas del lenguaje para el sistema LDL. Una visión general de CORAL aparece en Ramakrishnan y otros (1992), y la implementación se describe en Ramakrishnan y otros (1993). En Srivastava y otros (1993) se describe una extensión para dar soporte a las características de orientación a objetos, denominada CORAL++. Ullman (1985) proporciona los fundamentos para el sistema NAIL!, que se describe en Morris y otros (1987). Phipps y otros (1991) describe el sistema de bases de datos deductivo GLUE-NAIL!

Zaniolo (1990) examina el fondo teórico y la importancia práctica de las bases de datos deductivas. Nicolas (1997) ofrece una historia excelente de los desarrollos que han llevado a los sistemas de bases de datos orientados a objetos y deductivos (DOODs). Falcone y otros (1997) examina el panorama DOOD. Algunas referencias sobre el sistema VALIDITY son Friesen y otros (1995), Vieille (1997), y Dietrich y otros (1999).

CAPÍTULO 25

# Bases de datos distribuidas y arquitecturas cliente-servidor

En este capítulo vamos a centrar nuestra atención en los DDBS (Sistema de bases de datos distribuidas, *Distributed DataBase System*), los DDBMS (Sistemas de administración de bases de datos distribuidas, *Distributed DataBase Management Systems*) y en cómo se usa la arquitectura cliente-servidor como plataforma para el desarrollo de aplicaciones de bases de datos. La tecnología DDB emergió gracias a la unión de otras dos tecnologías: la de base de datos y la de comunicación de datos y de redes. Lo más reciente son los enormes avances dados en el área de las tecnologías alámbricas e inalámbricas (desde las comunicaciones por satélite y celulares hasta las MAN [Redes de área metropolitana, *Metropolitan Area Networks*], la estandarización de protocolos como Ethernet, TCP/IP y ATM [Modo de transferencia asíncrono, *Asynchronous Transfer Mode*], así como la explosión de Internet). Mientras en los 70 y principios de los 80 las bases de datos se movían hacia la centralización produciendo unas monolíticas y gigantescas estructuras, la tendencia a finales de esa década era la contraria: más descentralización y autonomía de procesamiento. Con los avances en la computación y el procesamiento distribuido de los sistemas operativos, la comunidad de investigación de las bases de datos trabajó sin descanso para resolver los problemas derivados de la distribución de datos, las consultas distribuidas y el procesamiento de las transacciones, la administración de los metadatos de las bases de datos distribuidas y otros temas, y desarrolló muchos prototipos. Sin embargo, un DDBMS totalmente operativo que incluyera la funcionalidad y técnicas propuestas por la investigación DDB nunca emergió como un producto comercialmente viable. La mayoría de fabricantes no centró sus esfuerzos en el desarrollo de un producto DDBMS *puro*, sino en la generación de sistemas basados en cliente-servidor, o a través de tecnologías para el acceso a fuentes de datos distribuidos de manera heterogénea.

Sin embargo, las organizaciones han estado muy interesadas en la *descentralización* del procesamiento (a nivel de sistema) a la vez que conseguían una *integración* de las fuentes de información (a nivel lógico) dentro de sus usuarios, aplicaciones y sistemas de bases de datos distribuidos geográficamente. Asociada a los avances en las comunicaciones, ahora existe una aceptación general del esquema cliente-servidor en el desarrollo de una aplicación, el cual asume muchos de los temas DDB.

En este capítulo vamos a tratar las bases de datos distribuidas y las arquitecturas cliente-servidor[1] en el desarrollo de la tecnología de base de datos que está íntimamente ligada a los avances en comunicaciones y redes. Los detalles posteriores están fuera del alcance de esta obra; si lo desea, puede consultar varios textos relativos a comunicaciones y *networking* (consulte la bibliografía seleccionada al final de este capítulo).

[1] El lector debe repasar la introducción a la arquitectura cliente-servidor de la Sección 2.5.

La Sección 25.1 contiene una introducción a la administración de una base de datos distribuida y a los conceptos relacionados con ella. La Sección 25.2 ofrece información detallada sobre el diseño de una base de datos distribuida, que comporta la fragmentación de los datos y su distribución a lo largo de múltiples localizaciones con posible replicación. La Sección 25.3 presenta distintos tipos de sistemas de bases de datos distribuidas, incluyendo los federados y los de "múltiples bases de datos", poniendo de relieve los problemas de heterogeneidad y las necesidades de autonomía de los sistemas de bases de datos federadas, los cuales dominarán el panorama en los años venideros. Las Secciones 25.4 y 25.5 presentan, respectivamente, las técnicas para el procesamiento de transacciones y consultas en una base de datos distribuida. La Sección 25.6 aborda cómo están relacionados los conceptos de la arquitectura cliente-servidor con las bases de datos distribuidas. La Sección 25.7 amplía los futuros problemas de las arquitecturas cliente-servidor. La Sección 25.8 se centra en las características de la base de datos distribuida Oracle RDBMS. Si desea una breve introducción al tema, consulte las Secciones 25.1, 25.3 y 25.6.

# 25.1 Conceptos de bases de datos distribuidas

Las bases de datos distribuidas aportan las ventajas del procesamiento distribuido al dominio de la administración de una base de datos. Un **sistema de computación distribuido** consiste en un número de elementos de procesamiento, no necesariamente homogéneos, que están interconectados mediante una red de computadores, y que cooperan para la realización de ciertas tareas asignadas. Como objetivo general, estos sistemas dividen un gran e inmanejable problema en piezas más pequeñas para resolverlo de una manera coordinada. La viabilidad económica de este planteamiento procede de dos razones: una mayor potencia de computación emparejada a la resolución de una tarea compleja, y que cada elemento de procesamiento autónomo pueda ser administrado de manera independiente y desarrollar sus propias aplicaciones.

Podemos definir una **DDB (Base de datos distribuida, *Distributed DataBase*)** como una colección de múltiples bases de datos distribuidas interrelacionadas de forma lógica sobre una red de computadores, y un **DDBMS (Sistema de administración de bases de datos distribuidas, *Distributed DataBase Management System*)** como el software encargado de administrar la base de datos distribuida mientras hace la distribución transparente para el usuario.[2]

Una colección de ficheros almacenados en nodos diferentes de una red y el mantenimiento de las interrelaciones entre ellos a través de hiperenlaces se ha convertido en una organización común en Internet, todo ello mediante páginas web. Las funciones comunes de la administración de una base de datos, incluyendo el procesamiento uniforme de las consultas y las transacciones, *no se aplica* aún a este escenario. Sin embargo, la tecnología se está moviendo hacia las bases de datos *World Wide Web* (WWW) distribuidas, lo que será una realidad en un futuro cercano. Veremos los problemas derivados del acceso a bases de datos en la Web en el Capítulo 26.

## 25.1.1 Tecnología paralela frente a distribuida

Tornando nuestra atención hacia las arquitecturas de sistema paralelo, existen dos tipos fundamentales de arquitecturas de sistema multiprocesador:

- **Arquitectura de memoria compartida (estrechamente acoplada o *tightly coupled*).** Varios procesadores comparten el almacenamiento secundario (disco) y la memoria primaria.
- **Arquitectura de disco compartido (débilmente acoplada o *loosely coupled*).** Varios procesadores comparten el almacenamiento secundario (disco), pero cada uno de ellos tiene su propia memoria primaria.

[2] Esta definición y parte de los argumentos de esta sección están basados en Ozsu y Valduriez (1999).

Estas arquitecturas permiten a los procesadores comunicarse entre sí sin los gastos derivados del intercambio de mensajes sobre una red.[3] Los sistemas de bases de datos desarrollados sobre las arquitecturas anteriores se llaman **sistemas de administración paralelos de bases de datos** en lugar de DDBMS, ya que emplean tecnología de procesador paralelo. La **arquitectura "nada compartido"** es otro tipo de sistema multiprocesador. En ella, cada procesador tiene su propia memoria (disco) primaria y secundaria, no existe memoria común, y esos procesadores se comunican mediante una red de interconexión de alta velocidad (bus o *switch*). Aunque esta arquitectura se asemeja a un entorno de procesamiento de base de datos distribuida, existen unas grandes diferencias en el modo de operar. En sistemas multiprocesador de nada compartido, existe simetría y homogeneidad de nodos; esto no se cumple en un entorno de base de datos distribuida, donde es muy común la heterogeneidad del hardware y el sistema operativo de cada nodo. La arquitectura de nada compartido está considerada también como un entorno para bases de datos paralelas. La Figura 25.1 pone de manifiesto estas arquitecturas tan diferentes.

## 25.1.2 Ventajas de las bases de datos distribuidas

La administración de bases de datos distribuidas ha sido propuesta por varias razones que van desde la descentralización organizativa y el procesamiento económico a una mayor autonomía. Vamos a ver ahora algunas de estas ventajas.

1. **Administración de datos distribuidos con distintos niveles de transparencia.** De manera ideal, un DBMS debe ser una **distribución transparente** en el sentido de ocultar los detalles de dónde está físicamente ubicado cada fichero (tabla, relación) dentro del sistema. Consideremos la base de datos empresarial de la Figura 5.5 que hemos utilizado como ejemplo a lo largo de este libro. Las tablas EMPLEADO, PROYECTO y TRABAJA_EN podrían estar fragmentadas horizontalmente (esto es, en conjuntos de filas, tal y como veremos más adelante en la Sección 25.2) y almacenadas con posible replicación de la forma mostrada en la Figura 25.2. En este caso, son posibles los siguientes tipos de transparencias:

   - **Transparencia de red o de distribución.** Hace referencia a la autonomía del usuario de los detalles operacionales de la red. Puede dividirse en transparencia de localización y de denominación. La **transparencia de localización** hace mención al hecho de que el comando usado para llevar a cabo una tarea es independiente de la ubicación de los datos y del sistema desde el que se ejecutó dicho comando. La **transparencia de denominación** implica que, una vez especificado un nombre, puede accederse a los objetos nombrados sin ambigüedad y sin necesidad de ninguna especificación adicional.

   - **Transparencia de replicación.** Como podemos ver en la Figura 25.2, pueden almacenarse copias de los datos en distintos lugares para disponer de una mayor disponibilidad, rendimiento y fiabilidad. La transparencia de replicación permite que el usuario no se entere de la existencia de copias.

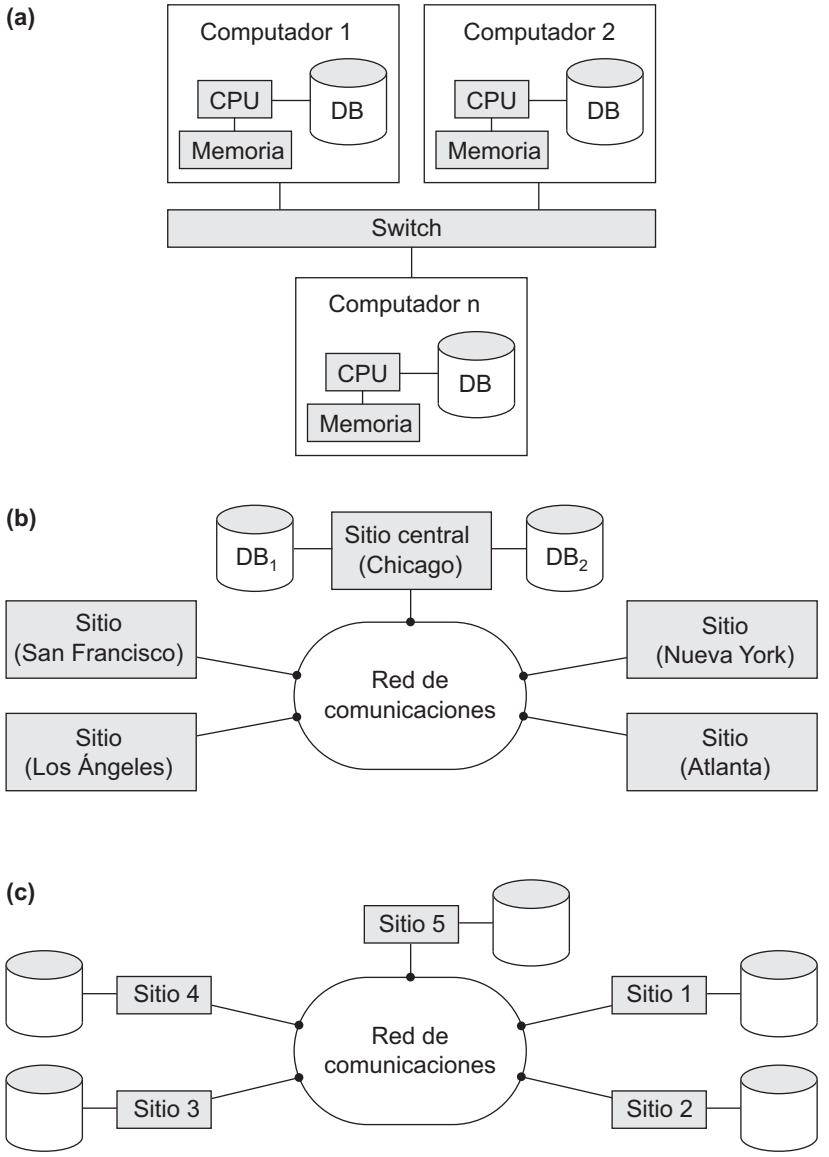
   - **Transparencia de fragmentación.** Existen dos posibles tipos de fragmentación: la **horizontal** distribuye una relación en conjuntos de tuplas (filas), mientras que la **vertical** lo hace en subrelaciones, de modo que cada subrelación está definida por un subconjunto de las columnas de la relación original. Una consulta global del usuario debe ser transformada en varias consultas fragmentadas. La transparencia de fragmentación permite que el usuario no se entere de la existencia de fragmentos.

   - La **transparencia de diseño** y de **ejecución** hace referencia a la libertad de saber cómo está diseñada la base de datos distribuida y dónde ejecuta una transacción.

---

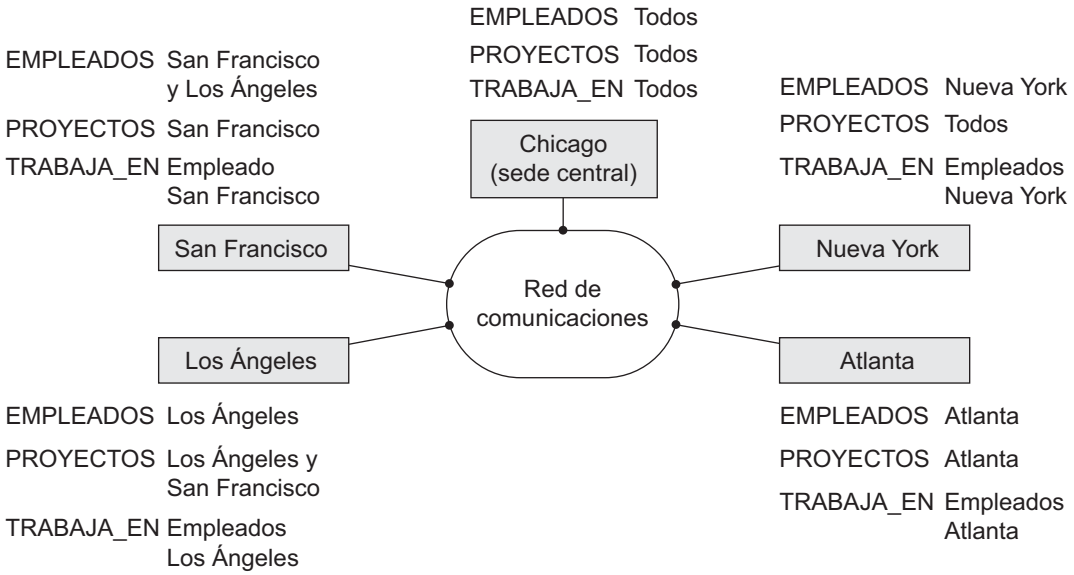
[3] Si se comparten tanto la memoria primaria como la secundaria, la arquitectura se conoce también como **arquitectura "todo compartido"**.

**Figura 25.1.** Algunos tipos de arquitecturas de base de datos. (a) Arquitectura "nada compartido". (b) Arquitectura en red con una base de datos centralizada en una de sus ubicaciones. (c) Arquitectura de base de datos distribuida auténtica.




2. **Incremento de la fiabilidad y la disponibilidad.** Éstas son dos de las más importantes ventajas de las bases de datos distribuidas. La **fiabilidad** está definida ampliamente como la probabilidad de que un sistema esté funcionando (no caído) en un momento de tiempo, mientras que la **disponibilidad** es la probabilidad de que el sistema esté continuamente disponible durante un intervalo de tiempo. Cuando los datos y el software DBMS están distribuidos a lo largo de distintas localizaciones, uno de ellos puede fallar, mientras el resto continúa operativo. Sólo los datos y el software almacenados en la localización que falla serán los que no estén disponibles. Esto mejora tanto la fiabilidad como la disponibilidad. Se logra una apreciable mejora al *replicar* tanto los datos como el software en más de una ubicación. En un sistema centralizado, el fallo de una ubicación provoca la caída del sistema para todos

**Figura 25.2.** Distribución de datos y replicación entre bases de datos distribuidas.




los usuarios. En una base de datos distribuida, parte de la información puede estar inaccesible, pero sí se podrá acceder a otras partes de la base de datos.

3. **Rendimiento mejorado.** Un DBMS distribuido fragmenta la base de datos manteniendo la información lo más cerca posible del punto donde es más necesaria. La **localización de datos** reduce el enfrentamiento por la CPU y los servicios de E/S, a la vez que atenúa los retardos en el acceso implícito a las redes de área extendida. Cuando se distribuye una base de datos a lo largo de varias localizaciones, lo que obtenemos son bases de datos más pequeñas. Como resultado, las consultas locales y las transacciones de acceso a los datos de uno de estos sitios tienen un mayor rendimiento debido al menor tamaño de esas bases de datos. Además, cada sitio tiene que ejecutar un menor número de transacciones que si todas ellas fueran llevadas a cabo por una base de datos centralizada. Por otra parte, el paralelismo interconsultas e intraconsultas puede conseguirse ejecutando múltiples consultas de diferentes sitios, o dividiendo esa consulta en otras más pequeñas que se ejecutan en paralelo. Todo esto contribuye a mejorar el rendimiento.
4. **Expansión más sencilla.** En un entorno distribuido, la expansión del sistema en términos de incorporación de más datos, incremento del tamaño de las bases de datos o la adición de más procesadores es mucho más sencilla.

Las transparencias que hemos comentado en el punto (1) conducen a un compromiso entre la facilidad de uso y el coste de proporcionar dicha transparencia. La transparencia total ofrece al usuario global una vista de todo el DDBS de igual manera que si se tratase de un sistema centralizado. La transparencia se ofrece como un complemento a la **autonomía**, que ofrece a los usuarios un control severo sobre las bases de datos locales. Las características de transparencia deben implementarse como parte del lenguaje de usuario, el cual debería traducir los servicios solicitados en las operaciones apropiadas. Adicionalmente, la transparencia impacta sobre las características que deben ser mejoradas por el sistema operativo y el DBMS.

### 25.1.3 Funciones adicionales de las bases de datos distribuidas

La distribución conlleva un incremento en la complejidad del diseño del sistema y de su implementación. Para obtener todas esas ventajas de las que hemos hablado anteriormente, el software DDBMS debe ser capaz de ofrecer las siguientes funciones además de todas las de un DBMS centralizado:

- **Seguimiento de los datos.** La capacidad de controlar la distribución de los datos, la fragmentación y la replicación expandiendo el catálogo DDBMS.
- **Procesamiento de consultas distribuidas.** La posibilidad de acceder a sitios remotos y de transmitir consultas y datos a lo largo de todos esos sitios mediante una red de comunicación.
- **Administración de transacciones distribuidas.** La facultad de diseñar estrategias de ejecución de consultas y transacciones que accedan a los datos desde más de una ubicación y de sincronizar el acceso a los datos distribuidos y de mantener la integridad de toda la base de datos.
- **Administración de datos replicados.** La capacidad de decidir a qué copia de un dato acceder y de mantener la consistencia de las copias de un elemento de datos replicado.
- **Recuperación de una base de datos distribuida.** La facultad de recuperarse de las caídas de una localización individual u otro tipo de fallos, como los fallos en los enlaces de comunicación.
- **Seguridad.** Las transacciones distribuidas deben ejecutarse con una adecuada administración de la seguridad de los datos y contando con los privilegios de autorización/acceso de los usuarios.
- **Administración del directorio (catálogo) distribuido.** Un directorio contiene información (metadatos) sobre los datos de la base de datos. Puede ser global a toda la DDB, o local para cada sitio. La colocación y distribución del directorio son temas relacionados con el diseño y las políticas.

Estas funciones en sí mismas aumentan la complejidad de un DDBMS en relación a un DBMS centralizado. Antes de disfrutar plenamente de todas las ventajas potenciales de la distribución debemos encontrar soluciones satisfactorias a estos problemas de diseño. Incluir todas estas funcionalidades adicionales es difícil de conseguir, y encontrar soluciones óptimas supone dar un paso más allá.

A nivel de **hardware**, los siguientes son los factores principales que distinguen un DDBMS de un sistema centralizado:

- Existen múltiples computadores llamados **sitios** o **nodos**.
- Estos sitios deben estar conectados por algún tipo de **red de comunicación** para transmitir los datos y los comandos entre ellos, como puede verse en la Figura 25.1(c).

Estos sitios pueden estar cercanos entre sí (digamos, dentro del mismo edificio o grupo de edificios adyacentes) y conectados mediante una **red de área local**, o estar geográficamente distribuidos a larga distancia y enlazados a través de una **red de área expandida** o ***long-haul***. Las redes de área local suelen emplear cables mientras que las *long-haul* utilizan líneas telefónicas o satélites. También es posible usar una combinación de redes.

Las redes pueden tener diferentes **topologías** que definen las rutas de comunicación directas entre los sitios. El tipo y la topología de la red usada pueden tener un efecto significativo sobre el rendimiento y, por ello, sobre las estrategias de distribución del procesamiento de las consultas y el diseño de la base de datos distribuida. Sin embargo, para arquitecturas de alto nivel, no importa el tipo de red empleada; sólo cuenta que cada sitio sea capaz de comunicarse, directa o indirectamente, con los demás. Para el resto de este capítulo asumiremos que existe algún tipo de red de comunicación entre los sitios, sin importarnos su topología particular. No nos centraremos en ningún problema específico de las redes, aunque es importante saber que para que un DDBS opere de manera eficiente, el diseño de la red y los problemas de rendimiento son críticos.

## 25.2 Técnicas de fragmentación, replicación y asignación de datos para el diseño de bases de datos distribuidas

En esta sección vamos a ver las diferentes técnicas empleadas para partir una base de datos en unidades lógicas, llamadas **fragmentos**, los cuales pueden almacenarse en varios sitios. Trataremos también el uso de la

**replicación de datos**, que permite que cierta información se almacene en más de un sitio, y el proceso de **asignación** de fragmentos (o réplicas de fragmentos) para su almacenamiento en distintos sitios. Estas técnicas se emplean durante el proceso de **diseño de la base de datos distribuida**. La información relativa a la fragmentación, asignación y replicación de los datos se almacena en un **directorio global** al cual acceden las aplicaciones DDBS según sea necesario.

### 25.2.1 Fragmentación de datos

En una DDB, las decisiones se toman en función del sitio que debe emplearse para almacenar dichas porciones de la base de datos. Por ahora asumiremos que *no existe replicación;* esto es, cada relación (o porción de una relación) se almacena sólo en un sitio. Más adelante en esta sección trataremos la replicación y sus efectos. Usamos también la terminología de las bases de datos relacionales (conceptos similares se aplican a otros modelos de datos). Partimos de un esquema de base de datos relacional y debemos decidir cómo distribuir las relaciones ente los diferentes sitios. Para ilustrar nuestro debate usaremos el esquema de la Figura 5.5.

Antes de decidir cómo distribuir los datos, debemos determinar las *unidades lógicas* de la base de datos que deben distribuirse. Las unidades lógicas más simples son las propias relaciones; es decir, cada relación *completa* almacenada en un sitio particular. En nuestro ejemplo, debemos decidir el sitio en el que almacenar las relaciones EMPLEADO, DEPARTAMENTO, PROYECTO, TRABAJA_EN y SUBORDINADO de la Figura 5.5. Sin embargo, en muchos casos, una relación puede dividirse en unidades lógicas más pequeñas con vistas a su distribución. Por ejemplo, consideremos la base de datos de empresa de la Figura 5.6 y asumamos que existen tres sitios de computación (uno por cada departamento de la empresa).[4]

Puede que queramos almacenar la información de la base de datos referente a cada departamento en el computador de ese departamento. Para ello, una técnica llamada *fragmentación horizontal* nos permite dividir cada relación por departamento.

**Fragmentación horizontal.** La fragmentación horizontal de una relación es un subconjunto de las tuplas de esa relación. Las tuplas pertenecientes al fragmento horizontal se especifican mediante una condición sobre uno o más atributos de la relación. Con frecuencia, sólo es necesario uno de estos atributos. Por ejemplo, podemos definir tres fragmentos horizontales en la relación EMPLEADO de la Figura 5.6 con las siguientes condiciones: (Dno = 5), (Dno = 4) y (Dno = 1) (cada fragmento contiene las tuplas EMPLEADO que trabajan para un departamento particular). De forma análoga, podemos definir tres fragmentos horizontales en la relación PROYECTO con las condiciones (NumDptoProyecto = 5), (NumDptoProyecto = 4) y (NumDptoProyecto = 1) (cada fragmento contiene las tuplas PROYECTO controladas por un departamento particular). La **fragmentación horizontal** divide una relación *horizontalmente* agrupando filas para crear subconjuntos de tuplas, cada uno de ellos con un cierto significado lógico. Estos fragmentos pueden asignarse entonces a diferentes sitios del sistema distribuido. La **fragmentación horizontal derivada** aplica el particionamiento de una relación primaria (DEPARTAMENTO en nuestro ejemplo) a otras relaciones secundarias (EMPLEADO y PROYECTO) que están relacionadas con aquélla a través de una clave externa (*foreign key*). De esta forma, los datos relacionados entre las relaciones primaria y secundaria permanecen fragmentados de la misma forma.

**Fragmentación vertical.** Puede que cada sitio no necesite todos los atributos de una relación, lo que indicaría la necesidad de un tipo distinto de fragmentación. La fragmentación vertical divide una relación "verticalmente" por columnas. Un **fragmento vertical** de una relación sólo mantiene ciertos atributos de la misma. Por ejemplo, queremos fragmentar la relación EMPLEADO en dos fragmentos verticales. El primero incluye la información personal (Nombre, FechaNac, Dirección y Sexo) y la segunda la relativa al trabajo (Dni, Sueldo, SuperDni y Dno). Esta fragmentación vertical no es demasiado adecuada ya que, si los dos fragmentos se almacenan de forma separada, no podemos volver a obtener las tuplas de los empleados originales porque *no existe un atributo común* entre esos fragmentos. Es necesario incluir la clave primaria, o alguna otra

[4] Desde luego, en una situación real, existirán muchas más tuplas en la relación de las que pueden verse en la Figura 5.6.

clave candidata, en *cada* fragmento vertical de modo que nos permita reconstruir la relación completa a partir de ellos. Por consiguiente, debemos añadir el atributo Dni al fragmento con la información personal.

Cada fragmento horizontal de una relación $R$ puede especificarse con una operación $\sigma_{Ci}(R)$ en el álgebra relacional. Un conjunto de fragmentos horizontales cuyas condiciones $C_1$, $C_2$, ..., $C_n$ incluyan todas las tuplas de $R$ [esto es, cada tupla de $R$ satisface ($C_1$ OR $C_2$ OR ... OR $C_n$)], recibe el nombre de **fragmentación horizontal completa** de $R$. En muchos casos, una de estas fragmentaciones es también **disjunta**; es decir, ninguna tupla de $R$ satisface ($C_i$ AND $C_j$) para cualquier $i \neq j$. Nuestros dos ejemplos anteriores de fragmentación horizontal de las relaciones EMPLEADO y PROYECTO eran completas y disjuntas. Para reconstruir la relación $R$ a partir de una fragmentación horizontal *completa* necesitamos aplicar la operación UNION a los fragmentos.

Un fragmento vertical de una relación $R$ puede especificarse mediante una operación $\pi_{Li}(R)$ en el álgebra relacional. Un conjunto de fragmentos verticales cuyas listas de proyección $L_1$, $L_2$, ..., $L_n$ incluyen todos los atributos de $R$ pero que sólo comparten el atributo de clave primaria recibe el nombre de **fragmentación vertical completa** de $R$. En este caso, las listas de proyección satisfacen las dos siguientes condiciones:

- $L_1 \cup L_2 \cup ... \cup L_n = \text{ATTRS}(R)$.
- $L_i \cap L_j = \text{PK}(R)$ para cualquier $i \neq j$, donde ATTRS($R$) es el conjunto de los atributos de $R$ y pk($R$) es la clave primaria de $R$.

Para reconstruir la relación $R$ a partir de una fragmentación vertical *completa* aplicamos la operación OUTER UNION sobre los fragmentos verticales (asumimos que no se ha usado fragmentación horizontal). Podemos aplicar una operación FULL OUTER JOIN y obtener el mismo resultado para una fragmentación vertical completa, aun cuando haya podido aplicarse también alguna fragmentación horizontal. Los dos fragmentos verticales de la relación EMPLEADO con las listas de proyección $L_1 = \{$Dni, Nombre, FechaNac, Dirección, Sexo$\}$ y $L_2 = \{$Dni, Sueldo, SuperDni, Dno$\}$ constituyen una fragmentación vertical completa de EMPLEADO.

Dos fragmentos horizontales que no son ni completos ni disjuntos son los que están definidos en la relación EMPLEADO de la Figura 5.5 por las condiciones (Sueldo > 50000) y (Dno = 4); pueden no incluir todas las tuplas EMPLEADO y las comunes. Dos fragmentos verticales que no son completos son los definidos por las listas de atributo $L_1 = \{$Nombre, Dirección$\}$ y $L_2 = \{$Dni, Nombre, Sueldo$\}$; estas listas violan ambas condiciones de una fragmentación vertical completa.

**Fragmentación mixta (híbrida).** Podemos entremezclar los dos tipos de fragmentación para obtener una **fragmentación mixta**. Por ejemplo, podemos combinar las fragmentaciones verticales y horizontales de la relación EMPLEADO en otra que incluye seis fragmentos. En este caso, la relación original puede reconstruirse aplicando operaciones UNION *y* OUTER UNION (u OUTER JOIN) en el orden apropiado. En general, un **fragmento** de una relación $R$ puede especificarse mediante una combinación de operaciones SELECT-PROYECTO $\pi_L(\sigma_C(R))$. Si $C$ = TRUE (es decir, se seleccionan todas las tuplas) y $L \neq$ ATTRS($R$), obtenemos un fragmento vertical, y si $C \neq$ TRUE y $L =$ ATTRS($R$), se consigue un fragmento horizontal. Por último, si $C \neq$ TRUE y $L \neq$ ATTRS($R$), se genera un fragmento mixto. Una relación puede considerarse en sí misma como un fragmento si $C =$ TRUE y $L =$ ATTRS($R$). En el siguiente debate, el término *fragmento* se utiliza para referirnos tanto a una relación como a cualquiera de los tipos de fragmentos anteriores.

Un esquema de **fragmentación** de una base de datos es un conjunto de fragmentos que incluyen *todos* los atributos y tuplas de esa base de datos y que satisface la condición de que es posible reconstruirla en su totalidad aplicando alguna secuencia de operaciones OUTER UNION (u OUTER JOIN) y UNION. A veces resulta útil (aunque no es necesario) que todos los fragmentos sean disjuntos excepto para la repetición de las claves primarias a lo largo de los fragmentos verticales (o mixtos). En el último caso, toda la replicación y distribución de fragmentos está claramente especificado en una etapa posterior y separada de la fragmentación.

Un **esquema de ubicación** describe la colocación de los fragmentos en los sitios del DDBS; por tanto, es un mapa que indica, por cada fragmento, el (los) sitio(s) en el que está almacenado. Si un fragmento se encuentra en más de un sitio se dice que está **replicado**.

### 25.2.2 Replicación y ubicación de los datos

La replicación es útil para mejorar la disponibilidad de los datos. El caso más extremo es la replicación de *toda la base de datos* en cada sitio del sistema distribuido, lo que genera una **base de datos distribuida totalmente replicada**. Esto puede mejorar considerablemente la disponibilidad, ya que el sistema puede seguir funcionando con tal de que uno de los sitios esté activo. También mejora el rendimiento en la recuperación de consultas globales porque los resultados de este tipo de consultas pueden obtenerse localmente desde cada uno de estos sitios; por tanto, una consulta de recuperación puede llevarse a cabo en el sitio local desde el que se lanzó, siempre y cuando incluya un módulo servidor. La principal desventaja de este entorno es que puede ralentizar drásticamente las operaciones de actualización, ya que es preciso ejecutar dicha operación en cada copia de la base de datos para mantener la consistencia de las mismas. Esto es especialmente claro en el caso de que existan muchas copias de la base de datos. La replicación total hace que las técnicas de control de concurrencia y recuperación sean más caras que las utilizadas en el caso de que esa replicación no existiera, como veremos en la Sección 25.5.

Al otro extremo de la replicación total se encuentra la **no replicación**, es decir, cada fragmento está almacenado en un sitio. En este caso, todos los fragmentos *deben ser disjuntos*, excepto en la repetición de claves primarias a lo largo de los fragmentos verticales (o mixtos). Esto suele conocerse también como **ubicación no redundante**.

Entre estos dos extremos contamos con un amplio espectro de **replicaciones parciales** de datos, esto es, algunos fragmentos de la base de datos pueden replicarse mientras que otros no. El número de copias de cada fragmento puede oscilar desde una hasta el número total de sitios del sistema distribuido. Existe un caso especial de replicación parcial que ocurre entre trabajadores móviles (como comerciales, planificadores financieros y cobradores de deudas), los cuales llevan las bases de datos parcialmente replicadas en sus portátiles y PDAs y que las sincronizan periódicamente contra la base de datos del sistema.[5] El **esquema de replicación** es una descripción de los fragmentos de la replicación.

Cada fragmento, o cada copia del mismo, debe estar asignado a un sitio concreto del sistema distribuido. Este proceso recibe el nombre de **distribución de datos** (o **ubicación de datos**). La elección del sitio y el grado de replicación dependen de los objetivos de rendimiento y la disponibilidad del sistema, así como de los tipos y frecuencias de las transacciones efectuadas en cada sitio. Por ejemplo, si se necesita una disponibilidad alta, que las transacciones puedan ser efectuadas en cada sitio y si la mayoría de éstas son de recuperación, una base de datos totalmente replicada podría ser una buena opción. Sin embargo, si un sitio ejecuta ciertas transacciones que acceden a partes particulares de la base de datos, lo más sensato es ubicar esos fragmentos únicamente en ese sitio. Los datos a los que se accede desde múltiples sitios pueden ser replicados en ellos. Si se efectúan muchas actualizaciones, podría ser interesante limitar la replicación. Encontrar una solución óptima, o incluso buena, a la ubicación de datos distribuidos supone un problema que genera muchos quebraderos de cabeza.

### 25.2.3 Ejemplo de fragmentación, ubicación y replicación

Vamos a ver ahora un ejemplo de fragmentación y distribución de la base de datos empresarial de las Figuras 5.5 y 5.6. Supongamos que la empresa cuenta con tres sitios operativos (uno por cada departamento). Los sitios 2 y 3 pertenecen a los departamentos 5 y 4, respectivamente. En cada uno de estos sitios se espera un acceso frecuente a la información de EMPLEADO y PROYECTO para aquellos asalariados *que trabajan en ese departamento* y los proyectos *controlados por ese departamento*. Más adelante asumiremos que esos sitios accederán fundamentalmente a los atributos Nombre, Dni, Sueldo y SuperDni de EMPLEADO. El sitio 1 lo utiliza la oficina central y accede regularmente a toda la información de empleados y proyectos, además de controlar los datos de SUBORDINADO con miras a los seguros.

[5] Para proponer una acercamiento escalable a la sincronización de bases de datos parcialmente replicadas, consulte Mahajan y otros.

**Figura 25.3.** Ubicación de fragmentos en los sitios. (a) Relación de fragmentos del sitio 2 correspondiente al departamento 5. (b) Relación de fragmentos del sitio 3 correspondiente al departamento 4.

**(a)** EMPD_5

| Nombre | Apellido1 | Apellido2 | Dni | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|
| José | Pérez | Pérez | 123456789 | 30000 | 333445555 | 5 |
| Alberto | Campos | Sastre | 333445555 | 40000 | 888665555 | 5 |
| Fernando | Ojeda | Ordóñez | 666884444 | 38000 | 333445555 | 5 |
| Aurora | Oliva | Avezuela | 453453453 | 25000 | 333445555 | 5 |

DEP_5

| NombreDpto | NúmeroDpto | DniDirector | FechaIngresoDirector |
|---|---|---|---|
| Investigación | 5 | 333445555 | 1988-05-22 |

DEP_5_LOCS

| NúmeroDpto | Ubicación |
|---|---|
| 5 | Valencia |
| 5 | Sevilla |
| 5 | Madrid |

TRABAJA_EN_5

| DniEmpleado | NumProy | Horas |
|---|---|---|
| 123456789 | 1 | 32,5 |
| 123456789 | 2 | 7,5 |
| 666884444 | 3 | 40,0 |
| 453453453 | 1 | 20,0 |
| 453453453 | 2 | 20,0 |
| 333445555 | 2 | 10,0 |
| 333445555 | 3 | 10,0 |
| 333445555 | 10 | 10,0 |
| 333445555 | 20 | 10,0 |

PROYS_5

| NombreProyecto | NumProyecto | UbicaciónProyecto | NumDptoProyecto |
|---|---|---|---|
| Producto X | 1 | Valencia | 5 |
| Producto Y | 2 | Sevilla | 5 |
| Producto Z | 3 | Madrid | 5 |

**Datos del sitio 2**

**(b)** EMPD_4

| Nombre | Apellido1 | Apellido2 | Dni | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|
| Alicia | Jiménez | Celaya | 999887777 | 25000 | 987654321 | 4 |
| Juana | Sainz | Oreja | 987654321 | 43000 | 888665555 | 4 |
| Luis | Pajares | Morera | 987987987 | 25000 | 987654321 | 4 |

DEP_4

| NombreDpto | NúmeroDpto | DniDirector | FechaIngresoDirector |
|---|---|---|---|
| Administración | 4 | 987654321 | 1995-01-01 |

DEP_4_LOCS

| NúmeroDpto | Ubicación |
|---|---|
| 4 | Gijón |

TRABAJA_EN_4

| DniEmpleado | NumProy | Horas |
|---|---|---|
| 333445555 | 10 | 10,0 |
| 999887777 | 30 | 30,0 |
| 999887777 | 10 | 10,0 |
| 987987987 | 10 | 35,0 |
| 987987987 | 30 | 5,0 |
| 987654321 | 30 | 20,0 |
| 987654321 | 20 | 15,0 |

PROYS_4

| NombreProyecto | NumProyecto | UbicaciónProyecto | NumDptoProyecto |
|---|---|---|---|
| Computación | 10 | Gijón | 4 |
| Comunicaciones | 30 | Gijón | 4 |

**Datos del sitio 3**

Según todos estos requerimientos, toda la base de datos de la Figura 5.6 puede estar contenida en el sitio 1. Para determinar los fragmentos a replicar en los sitios 2 y 3, primero podemos fragmentar horizontalmente DEPARTAMENTO por su clave NúmeroDpto. A continuación, aplicamos la fragmentación derivada a las relaciones EMPLEADO, PROYECTO y LOCALIZACIONES_DPTO basándonos en sus claves externas para el número de departamento (Dno, NumDptoProyecto y NúmeroDpto, respectivamente, en la Figura 5.5). Podemos partir verticalmente los fragmentos EMPLEADO resultantes para incluir sólo los atributos {Nombre, Dni, Sueldo, SuperDni, Dno}. La Figura 25.3 muestra los fragmentos mixtos EMPD_5 y EMPD_4, que incluyen las tuplas EMPLEADO que satisfacen las condiciones Dno = 5 y Dno = 4, respectivamente. Los fragmentos horizontales de PROYECTO, DEPARTAMENTO y LOCALIZACIONES_DPTO están fragmentadas de forma similar por el número de departamento. Todos estos fragmentos (almacenados en los sitios 2 y 3) están replicados porque también se encuentran en las oficinas centrales (sitio 1).

Ahora nos toca partir la relación TRABAJA_EN y decidir cuáles de sus fragmentos se almacenan en los sitios 2 y 3. Nos enfrentamos al problema de que ningún atributo de TRABAJA_EN indica directamente el departamento al que pertenece cada tupla. De hecho, cada una de ellas relaciona a un empleado *e* con un proyecto *P*. Podríamos fragmentar TRABAJA_EN en base al departamento *D* en el que trabaja *e* o en función al departamento *D*' que controla *P*. La fragmentación se simplifica si contamos con una restricción que declara que $D = D'$ para todas las tuplas de TRABAJA_EN, es decir, si los empleados pueden trabajar sólo en los proyectos controlados por el departamento entonces trabajan en él. Sin embargo, no existe una restricción de este tipo en nuestra base de datos de la Figura 5.6. Por ejemplo, la tupla TRABAJA_EN <333445555, 10, 10.0> relaciona un empleado que trabaja para el departamento 5 con un proyecto controlado por el 4. En este caso, podríamos fragmentar TRABAJA_EN basándonos en el departamento en el que trabaja el empleado (lo cual está expresado por la condición *C*) para después volver a dividir en función al departamento que controla los proyectos en los que está trabajando el empleado, tal y como puede verse en la Figura 25.4.

En esta figura, la unión de los fragmentos $G_1$, $G_2$ y $G_3$ brinda todas las tuplas TRABAJA_EN para los empleados que trabajan para el departamento 5. De forma análoga, la unión de los fragmentos $G_4$, $G_5$ y $G_6$ contiene las tuplas de los que trabajan para el departamento 4. En el otro extremo, la unión de los fragmentos $G_1$, $G_4$ y $G_7$ ofrece las tuplas TRABAJA_EN de los proyectos controlados por el departamento 5. Las condiciones de cada uno de los fragmentos desde $G_1$ hasta $G_9$ pueden verse en la Figura 25.4. Las relaciones del tipo M:N, como ocurre con TRABAJA_EN, suelen contar con varias fragmentaciones lógicas posibles. En nuestra distribución de la Figura 25.3 hemos elegido incluir todos los fragmentos que pueden unirse mediante una tupla EMPLEADO o una tupla PROYECTO de los sitios 2 y 3. Por tanto, colocamos la unión de los fragmentos $G_1$, $G_2$, $G_3$, $G_4$ y $G_7$ en el sitio 2 y la de los fragmentos $G_4$, $G_5$, $G_6$, $G_2$ y $G_8$ en el 3. Observe que los fragmentos $G_2$ y $G_4$ están repetidos en ambos sitios. Esta estrategia de ubicación permite que la concatenación entre los fragmentos EMPLEADO o PROYECTO locales del sitio 2 o 3 y el de TRABAJA_EN pueda llevare a cabo de una manera completamente local. Esto demuestra muy a las claras lo complejo que es el problema de la fragmentación y ubicación de una base de datos grande. La bibliografía seleccionada del final de este capítulo ofrece algo del trabajo llevado a cabo en esta área.

## 25.3 Tipos de sistemas de bases de datos distribuidas

El término sistema de administración de base de datos distribuida puede describir varios sistemas que difieren entre sí en bastantes aspectos. Lo que todos estos sistemas tienen en común es el hecho de que tanto los datos como el software están distribuidos a lo largo de distintas localizaciones que están conectadas por algún tipo de red de comunicaciones. En esta sección vamos a tratar algunos de estos DDBMSs y los criterios y factores que hacen diferentes a algunos de ellos.

El primer factor a considerar es el **grado de homogeneidad** del software DDBMS. Si todos los servidores (o DBMSs locales individuales) y todos los usuarios (clientes) usan idéntico software, se dice que el DDBMS

**Figura 25.4.** Fragmentos completos y disjuntos de la relación TRABAJA_EN. (a) Fragmentos de TRABAJA_EN de los empleados que trabajan para el departamento 5 (C=[DniEmpleado en (SELECT Dni FROM EMPLEADO WHERE Dno=5)]). (b) Fragmentos de TRABAJA_EN de los empleados que trabajan para el departamento 4 (C=[DniEmpleado en (SELECT Dni FROM EMPLEADO WHERE Dno=4)]). (c) Fragmentos de TRABAJA_EN de los empleados que trabajan para el departamento 1 (C=[DniEmpleado en (SELECT Dni FROM EMPLEADO WHERE Dno=1)]).

**(a) Empleados del Departamento 5**

**G1**

| DniEmpleado | NumProy | Horas |
|---|---|---|
| 123456789 | 1 | 32,5 |
| 123456789 | 2 | 7,5 |
| 666884444 | 3 | 40,0 |
| 453453453 | 1 | 20,0 |
| 453453453 | 2 | 20,0 |
| 333445555 | 2 | 10,0 |
| 333445555 | 3 | 10,0 |

C1 = C and (NumProy in (SELECT NumProyecto FROM PROJECT WHERE NumDptoProyecto = 5))

**G2**

| DniEmpleado | NumProy | Horas |
|---|---|---|
| 333445555 | 10 | 10,0 |

C2 = C and (NumProy in (SELECT NumProyecto FROM PROJECT WHERE NumDptoProyecto = 4))

**G3**

| DniEmpleado | NumProy | Horas |
|---|---|---|
| 333445555 | 20 | 10,0 |

C3 = C and (NumProy in (SELECT NumProyecto FROM PROJECT WHERE NumDptoProyecto = 1))

**(b) Empleados del Departamento 4**

**G4**

| DniEmpleado | NumProy | Horas |
|---|---|---|

C4 = C and (NumProy in (SELECT NumProyecto FROM PROJECT WHERE NumDptoProyecto = 5))

**G5**

| DniEmpleado | NumProy | Horas |
|---|---|---|
| 999887777 | 30 | 30,0 |
| 999887777 | 10 | 10,0 |
| 987987987 | 10 | 35,0 |
| 987987987 | 30 | 5,0 |
| 987654321 | 30 | 20,0 |

C5 = C and (NumProy in (SELECT NumProyecto FROM PROJECT WHERE NumDptoProyecto = 4))

**G6**

| DniEmpleado | NumProy | Horas |
|---|---|---|
| 987654321 | 20 | 15,0 |

C6 = C and (NumProy in (SELECT NumProyecto FROM PROJECT WHERE NumDptoProyecto = 1))

**(c) Empleados del Departamento 1**

**G7**

| DniEmpleado | NumProy | Horas |
|---|---|---|

C7 = C and (NumProy in (SELECT NumProyecto FROM PROJECT WHERE NumDptoProyecto = 5))

**G8**

| DniEmpleado | NumProy | Horas |
|---|---|---|

C8 = C and (NumProy in (SELECT NumProyecto FROM PROJECT WHERE NumDptoProyecto = 4))

**G9**

| DniEmpleado | NumProy | Horas |
|---|---|---|
| 888665555 | 20 | Null |

C9 = C and (NumProy in (SELECT NumProyecto FROM PROJECT WHERE NumDptoProyecto = 1))

es **homogéneo**; en cualquier otro caso, es **heterogéneo**. Otro factor relacionado con el grado de homogeneidad es el **grado de autonomía local**. Si no hay previsión de que el sitio local funcione como un DBMS aislado, se dice que **no tiene autonomía local**. Por otro lado, si se permite que las transacciones tengan *acceso directo* a un servidor, el sistema tiene cierto grado de autonomía local.

En un extremo del espectro de la autonomía tenemos un DDBMS que *tiene el aspecto* de un DBMS centralizado para el usuario. Existe un único esquema conceptual, y todos los accesos al sistema se obtienen a través de un sitio que forma parte del DDBMS (lo que significa que existe autonomía local). En el otro extremo encontramos un tipo de DDBMS, llamado *DDBMS federado* o *sistema de "múltiples bases de datos"*, en el que cada servidor es un DBMS centralizado independiente y autónomo que tiene sus propios usuarios, transacciones y DBA locales y que, por consiguiente, cuenta con un amplio grado de *autonomía local.* El término **FDBS (Sistema de base de datos federado, *Federated DataBase System*)** se utiliza cuando existe alguna vista global o esquema de la federación de bases de datos que está compartida por las aplicaciones. En el otro extremo, un **sistema de "múltiples bases de datos"** no cuenta con un esquema global y construye uno de forma interactiva a medida que la aplicación los necesita. Ambos sistemas son híbridos entre los esquemas distribuido y centralizado, y la distinción que hacemos entre ellos no se sigue de forma estricta. Por tanto, haremos referencia al término FDBS en un sentido genérico.

En un FDBS heterogéneo, un servidor puede ser un DBMS relacional, otro un DBMS de red y un tercero un DBMS de objetos o jerárquico; en esta situación es necesario contar con un lenguaje de sistema ortodoxo e incluir traductores de lenguaje para convertir las subconsultas realizadas en este lenguaje al que cada servidor entiende. A continuación veremos brevemente algunos de los problemas que afectan al diseño de los FDBSs.

**Problemas de los sistemas de administración de una base de datos federada.** El tipo de heterogeneidad presente en los FDBSs puede provenir de distintas fuentes. Veremos estas fuentes en primer lugar para, a continuación, centrarnos en cómo contribuyen los distintos tipos de autonomías a la heterogeneidad de la semántica que debe resolverse en un FDBS de este tipo.

- **Diferencias en los modelos de datos.** Las bases de datos de una organización proceden de una gran variedad de modelos entre los que se incluyen los llamados heredados (red y jerárquico; consulte los Apéndices D y E), el relacional, el de objetos e, incluso, el de ficheros. La capacidad de modelado de cada uno de ellos varía. Por tanto, tratar con ellos de manera uniforme a través de un único esquema global, o procesarlos mediante un solo lenguaje, supone todo un desafío. Incluso si dos bases de datos están en un entorno RDBMS, la misma información podría estar representada como un nombre de atributo, de relación o como un valor en cada base de datos. Esto exige un mecanismo de procesamiento de consultas inteligente que pueda relacionar la información basada en metadatos.
- **Diferencias en las restricciones.** La forma de especificar e implementar las restricciones varían de un sistema a otro. Existen características comparables que deben ser reconciliadas a la hora de construir un esquema global. Por ejemplo, las relaciones de los modelos ER están representadas como restricciones de integridad referencial en el modelo relacional. Los disparadores (*triggers*) pueden usarse para implementar ciertas restricciones en el modelo relacional. El esquema global debe ser capaz de lidiar con los potenciales conflictos que aparezcan entre las restricciones.
- **Diferencias en los lenguajes de consulta.** Incluso con el mismo modelo de datos, los lenguajes y sus versiones varían. Por ejemplo, SQL tiene múltiples versiones como el SQL-89, SQL-92 y SQL-99, y cada sistema tiene su propio conjunto de tipos de datos, operadores de comparación, elementos de manipulación de cadenas, etc.

**Heterogeneidad semántica.** Este fenómeno ocurre cuando existen diferencias en el significado, la interpretación y el uso pretendido del mismo dato, o uno relacionado. La heterogeneidad semántica entre DBSs supone el mayor obstáculo a la hora de diseñar esquemas globales de bases de datos heterogéneas. La **auto-**

**nomía de diseño** de DBSs componente hace referencia a su libertad de elegir los siguientes parámetros de diseño, los cuales afectan a la eventual complejidad de los FDBSs:

- **El universo de discurso desde el que se dibujan los datos.** Por ejemplo, dos cuentas de clientes; las bases de datos en la federación pueden estar en Estados Unidos y Japón y tienen distintos conjuntos de atributos acerca de las cuentas de cliente necesarias para las prácticas contables. Las fluctuaciones en la cotización de las diferentes monedas podría presentar también un problema. Por tanto, las relaciones de estas dos bases de datos que tienen nombres idénticos (CLIENTE o CUENTA) podrían tener alguna información común y otra totalmente diferente.
- **Representación y nombrado.** La representación y el nombrado de los elementos de datos y la estructura del modelo de datos podrían estar especificados previamente por cada base de datos local.
- **La comprensión, significado e interpretación subjetiva de los datos.** Ésta es una contribución fundamental a la heterogeneidad de la semántica.
- **Transacción y política de restricciones.** Se ocupa del criterio de serializabilidad, compensación de transacciones y otras políticas de transacción.
- **Derivación de resúmenes.** La agregación, resumen y otras características del proceso de datos y operaciones soportadas por el sistema.

La **autonomía de comunicación** de un componente DBS se refiere a su capacidad para decidir si se comunica con otro componente DBS. La **autonomía de ejecución** es la capacidad de un componente DBS de ejecutar operaciones locales sin interferencia de operaciones externas procedentes de otros componentes DBSs y su facultad para decidir el orden de ejecución. La **autonomía de asociación** de un componente DBS implica que tiene la potestad de decidir si quiere compartir, y en qué medida desea hacerlo, su funcionalidad (operaciones que soporta) y recursos (los datos que gestiona) con otros componentes DBSs. El mayor desafío en el diseño de los FDBS es permitir que los componentes DBSs operen entre sí a la vez que les ofrece los tipos de autonomía comentados anteriormente.

Una típica arquitectura de cinco niveles que soporta aplicaciones globales en el entorno FDBS puede verse en la Figura 25.5. En este entorno, el **esquema local** es el esquema conceptual (definición completa de la base de datos) de una base de datos componente, mientras que el **esquema de componente** se deduce traduciendo el esquema local a un modelo de datos ortodoxo, o CDM (Modelo de datos común, *Common Data Model*), para el FDBS. La conversión de un esquema local a otro de componente se acompaña con la generación de asignaciones que transformen los comandos de un esquema de componente a los correspondientes en el esquema local. El **esquema de exportación** representa el subconjunto de un esquema de componente que está disponible para el FDBS. El **esquema federado** es el esquema, o vista, global resultante de la integración de todos los esquemas de exportación compartibles. Los **esquemas externos** definen el esquema para un grupo de usuarios o una aplicación, como en la arquitectura de esquema de tres niveles.[6]

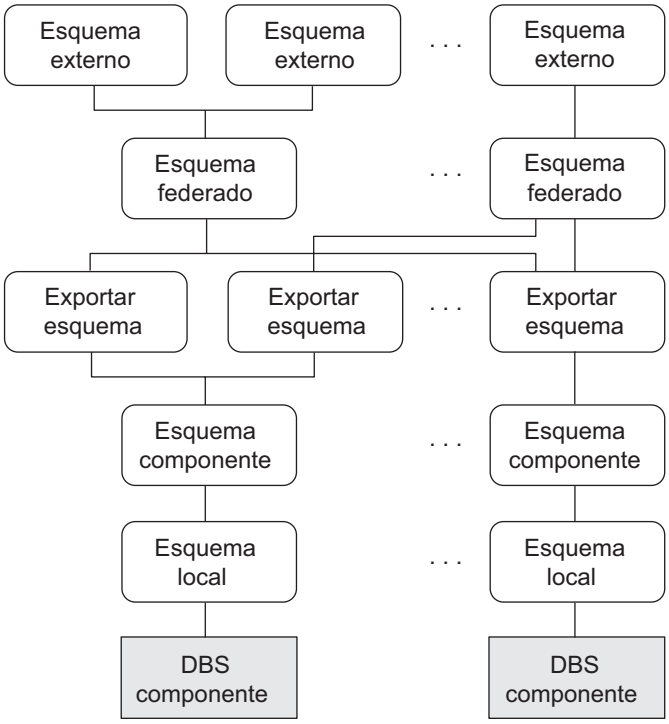
Todos los problemas relativos al procesamiento de consultas, de transacciones y la administración y recuperación de directorios y metadatos pueden aplicarse a los FDBSs con ciertas consideraciones adicionales. Pero no se encuentra dentro de nuestro objetivo hablar de ellos.

## 25.4 Procesamiento de consultas en bases de datos distribuidas

Ahora vamos a ver el modo en que un DDBMS procesa y optimiza una consulta. Primero trataremos los costes de comunicación del procesamiento de una consulta distribuida, para después pasar a un tipo de

[6] Para obtener información más detallada acerca de las autonomías y de la arquitectura de cinco niveles de los FDBMSs, consulte Sheth y Larson (1990).

**Figura 25.5.** La arquitectura de cinco niveles en un FDBS.




*Fuente*: Adaptado de Sheth y Larson, "Federated Database Systems for Managing Distributed Heterogeneous Autonomous Databases". *ACM Computing Surveys* (Vol. 22: No. 3, Septiembre de 1990).

operación especial, llamada *semijoin* (semiconcatenación), que se utiliza para optimizar ciertas consultas en un DDBMS.

### 25.4.1 Costes de transferencia de datos del procesamiento de una consulta distribuida

En el Capítulo 15 vimos los problemas relacionados con el procesamiento y la optimización de una consulta en un DBMS centralizado. En un sistema distribuido, otros factores adicionales complican el tratamiento de la consulta. El primero es el coste de la transferencia de los datos a través de una red. Entre estos datos se incluyen los ficheros intermedios que son transferidos a otros sitios para un mejor procesamiento, así como el resultado final que debe devolverse al sitio que solicitó la información. Aunque estos costes pueden no ser muy grandes si los sitios están conectados mediante una red de área local de alto rendimiento, la situación cambia ostensiblemente cuando se habla de otro tipo de redes. Por consiguiente, los algoritmos de optimización de consultas DDBMS tienen como objetivo fundamental reducir la *cantidad de la transferencia de datos* a la hora de elegir una estrategia de ejecución de una consulta distribuida.

Vamos a ilustrar este tema con dos sencillas consultas de ejemplo. Supongamos que las relaciones EMPLEADO y DEPARTAMENTO de la Figura 5.5 están distribuidas tal y como aparece en la Figura 25.6. Asumiremos que ninguna de las dos relaciones está fragmentada. Según la Figura 25.6, el tamaño de la relación EMPLEADO es de $100 * 10.000 = 10^6$ bytes, mientras que la de DEPARTAMENTO es de $35 * 100 = 3.500$ bytes. Consideremos la consulta C: *por cada empleado, recuperar su nombre y el del departamento para el cual trabaja.* Esto puede formularse del siguiente modo en el álgebra relacional:

**Figura 25.6.** Ejemplo para ilustrar el volumen de datos transferidos.

**Sitio 1:**

**EMPLEADO**

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|

10.000 registros
Longitud de registro: 100 bytes
Longitud del campo Dni: 9 bytes
Longitud del campo Nombre: 15 bytes
Longitud del campo Dno: 4 bytes
Longitud del campo Apellido1: 15 bytes

**Sitio 2:**

**DEPARTAMENTO**

| NombreDpto | NúmeroDpto | DniDirector | FechaIngresoDirector |
|---|---|---|---|

100 registros
Longitud de registro: 35 bytes
Longitud del campo NombreDpto: 4 bytes
Longitud del campo NombreDpto: 10 bytes
Longitud del campo DniDirector: 9 bytes

---

C: $\pi_{\text{Nombre,Apellido1,NombreDpto}}(\text{EMPLEADO} \bowtie_{\text{Dno}=\text{NúmeroDpto}} \text{DEPARTAMENTO})$

Si asumimos que cada empleado está relacionado con un departamento, el resultado de esta consulta incluirá 10.000 registros. Supongamos que cada registro tiene *40 bytes de longitud*. La consulta es enviada a un sitio 3 distinto, llamado **sitio resultado** porque es el lugar en el que se precisan los resultados. Ni la relación EMPLEADO ni DEPARTAMENTO residen en este sitio. Para ejecutar esta consulta distribuida existen tres tipos de estrategias simples:

1. Transferir las relaciones EMPLEADO y DEPARTAMENTO al sitio resultado, y ejecutar la concatenación en el sitio 3. En este caso deben trasferirse 1.000.000 + 3500 = 1.003.500 bytes.
2. Transferir la relación EMPLEADO al sitio 2, ejecutar la concatenación en él y transferir el resultado al 3. El tamaño del resultado de la consulta es de $40 * 10.000 = 400{,}000$ bytes, por lo que se tendrán que transferir 400.000 + 1.000.000 = 1.400.000 bytes.
3. Transferir la relación DEPARTAMENTO al sitio 1, ejecutar la concatenación en él y enviar el resultado al 3. En este caso, lo que se transfieren son 400.000 + 3.500 = 403.500 bytes.

Si nuestro criterio de optimización es minimizar la cantidad de la transferencia de datos, debemos elegir la tercera estrategia. Ahora vamos a considerar una nueva consulta C': *por cada departamento, recuperar su nombre y el de su director*. Esto quedaría del siguiente modo en el álgebra relacional:

C': $\pi_{\text{Nombre,Apellido1,NombreDpto}}(\text{DEPARTAMENTO} \bowtie_{\text{DniDirector}=\text{Dni}} \text{EMPLEADO})$

De nuevo, suponemos que la consulta se envía al sitio 3. En este caso, aplicaremos las mismas estrategias que las empleadas para la consulta C, excepto por el hecho de que el resultado de C' sólo incluye 100 registros, asumiendo que cada departamento tiene un único director:

1. Transferir EMPLEADO y DEPARTAMENTO al sitio resultado, y ejecutar la consulta en el sitio 3. En este caso deben transferirse un total de 1.000.000 + 3.500 = 1.003.500 bytes.
2. Transferir la relación EMPLEADO al sitio 2, ejecutar la concatenación en él y enviar el resultado al 3. El tamaño del resultado de la consulta es de $40 * 100 = 4.000$ bytes, por lo que se transferirán 4.000 + 1.000.000 = 1.004.000 bytes.

3. Transferir la relación DEPARTAMENTO al sitio 1, ejecutar la concatenación en él y enviar el resultado al 3. En este caso, se transfieren 4.000 + 3.500 = 7.500 bytes.

De nuevo elegimos la tercera estrategia (en este caso, por un abrumador margen con relación a las otras dos). Las tres metodologías anteriores son las más obvias cuando se trata de enviar el resultado a un sitio diferente (sitio 3) del que se encuentran los ficheros involucrados en la consulta (sitios 1 y 2). Sin embargo, supongamos que el sitio resultado es el 2; en este caso tenemos dos estrategias:

1. Transferir la relación EMPLEADO al sitio 2, ejecutar la consulta y presentar el resultado al usuario en el sitio 2. En este caso, se transfieren el mismo número de bytes (1.000.000) tanto para C como para C'.
2. Transferir la relación DEPARTAMENTO al sitio 1, ejecutar la consulta en él y devolver el resultado al sitio 2. En este caso, habrá que transferir 400.000 + 3.500 = 403.500 bytes para C y 4.000 + 3.500 = 7.500 para C'.

Existe otro tipo de planteamiento más complejo que, a veces, funciona mejor que estas estrategias simples. Dicho planteamiento emplea una operación llamada ***semijoin***, que veremos a continuación.

### 25.4.2 Procesamiento de una consulta distribuida usando una semijoin

La idea que se esconde tras el procesamiento de una consulta distribuida mediante una *operación de semijoin* es reducir el número de tuplas de una relación antes de transferirla a otro sitio. Intuitivamente, lo que pretendemos hacer es enviar la *columna de concatenación* de una relación *R* al sitio en el que se encuentra la otra relación *S* para, a continuación, concatenar esta columna con *S*. Tras esto, tanto los atributos de concatenación como los necesarios en el resultado son proyectados y embarcados de vuelta al sitio original y concatenados con *R*. De este modo, sólo se transfiere en una dirección la columna de concatenación de *R*, mientras que en la otra sólo viaja un subconjunto de *S* sin tuplas o atributos extraños. Si en la concatenación sólo participa un pequeño número de tuplas de *S*, esto puede ser una solución eficiente para reducir la transferencia de datos.

Para ilustrar todo este comentario, consideremos la siguiente estrategia para ejecutar c o C':

1. Proyectar los atributos de concatenación de DEPARTAMENTO al sitio 2 y trasferirlos al sitio 1. Para C enviamos $F = \pi_{\text{NúmeroDpto}}(\text{DEPARTAMENTO})$, cuyo tamaño es de $4 * 100 = 400$ bytes, mientras que para C' trasmitimos $F' = \pi_{\text{DniDirector}}(\text{DEPARTAMENTO})$, que tiene un tamaño de $9 * 100 = 900$ bytes.
2. Concatenamos el fichero transferido con la relación EMPLEADO en el sitio 1, y devolvemos al sitio 2 los atributos requeridos. Para C, enviamos $R = \pi_{\text{Dno, Nombre, Apellido1}}(F \bowtie_{\text{NúmeroDpto5Dno}} \text{EMPLEADO})$, cuyo tamaño es de $34 * 10.000 = 340.000$ bytes, mientras que para C' transferimos $R' = \pi_{\text{DniDirector, Nombre, Apellido1}}(F' \bowtie_{\text{DniDirector5Dni}} \text{EMPLEADO})$, que tiene un tamaño de $39 * 100 = 3.900$ bytes.
3. Ejecutar la consulta concatenando el fichero transferido *R* o *R*' con DEPARTAMENTO, y presentamos el resultado al usuario del sitio 2.

Usando este planteamiento, enviamos 340.400 bytes para C y 4.800 para C'. En el paso 2, limitamos los atributos y tuplas EMPLEADO trasmitidas al sitio 2 a sólo aquéllas que *serán concatenadas* con la tupla DEPARTAMENTO en el paso 3. Para la consulta C, esta reunión incluye todas las tuplas EMPLEADO, con lo que se obtiene una pequeña mejora. Sin embargo, para C' sólo serán necesarias 100 de las 10.000 tuplas EMPLEADO.

La semiconcatenación fue ideada para formalizar esta estrategia. Una **operación *semijoin*** $R \ltimes_{A=B} S$, en la que *A* y *B* son atributos de dominio compatible de *R* y *S* respectivamente, produce el mismo resultado que la expresión de álgebra relacional $\pi_R(R \bowtie_{A=B} S)$. En un entorno distribuido donde *R* y *S* residen en sitios diferentes, esta operación suele estar implementada de forma que primero se transfiere $F = \pi_B(S)$ al sitio en el que reside *R* para después concatenar *F* con *R*, lo que nos conduce a la estrategia mostrada aquí.

Observe que la operación semiconcatenación no es conmutativa; esto es,

$$R \ltimes S \neq S \ltimes R$$

### 25.4.3 Descomposición de una actualización y una consulta

En un DDBMS sin *transparencia de distribución,* el usuario articula una consulta directamente en términos de fragmentos específicos. Por ejemplo, consideremos otra consulta C: *recuperar los nombres y horas semanales de los empleados que trabajan en algún proyecto controlado por el departamento 5*, cuyas relaciones de los sitios 2 y 3 aparecen en la Figura 25.3, mientras que las del sitio 1 pueden verse en la Figura 5.6, como en nuestro anterior ejemplo. Un usuario que lanza una consulta de este tipo debe especificar si hace referencia a las relaciones PROYS5 y TRABAJA_EN_5 del sitio 2 (Figura 25.3) o a PROYECTO y TRABAJA_EN del sitio 1 (Figura 5.6). Además, es preciso que mantenga la consistencia de los datos replicados cuando se lleva a cabo la actualización de un DDBMS sin *transparencia de replicación.*

**Figura 25.7.** Condiciones guardián y lista de atributos para los fragmentos. (a) Fragmentos del sitio 2. (b) Fragmentos del sitio 3.

**a.** EMPD5
Lista de atributos: Nombre, Apellido1, Apellido2, Dni, Sueldo, SuperDni, Dno
Condición guardián: Dno=5
DEP5
Lista de atributos: * (todos los atributos NombreDpto, NúmeroDpto, DniDirector, FechaIngresoDirector)
Condición guardián: NúmeroDpto=5
DEP5_LOCS
Lista de atributos: * (todos los atributos NúmeroDpto, Localización)
Condición guardián: NúmeroDpto=5
PROYS5
Lista de atributos: * (todos los atributos NombreProyecto, NumProyecto, UbicaciónProyecto, NumDptoProyecto)
Condición guardián: NumDptoProyecto=5
TRABAJA_EN5
Lista de atributos: * (todos los atributos DniEmpleado, NumProy, Horas)
Condición guardián: DniEmpleado IN ($\pi_{Dni}$ (EMPD5)) OR NumProy IN ($\pi_{NumProyecto}$ (PROYS5))

**b.** EMPD4
Lista de atributos: Nombre, Apellido1, Apellido2, Dni, Sueldo, SuperDni, Dno
Condición guardián: Dno=4
DEP4
Lista de atributos: * (todos los atributos NombreDpto, NúmeroDpto, DniDirector, FechaIngresoDirector)
Condición guardián: NúmeroDpto=4
DEP4_LOCS
Lista de atributos: * (todos los atributos NúmeroDpto, Localización)
Condición guardián: NúmeroDpto=4
PROYS4
Lista de atributos: * (todos los atributos NombreProyecto, NumProyecto, UbicaciónProyecto, NumDptoProyecto)
Condición guardián: NumDptoProyecto=4
TRABAJA_EN4
Lista de atributos: * (todos los atributos DniEmpleado, NumProy, Horas)
Condición guardián: DniEmpleado IN ($\pi_{Dni}$ (EMPD4)) OR NumProy IN ($\pi_{NumProyecto}$ (PROYS4))

En el otro extremo, un DDBMS que soporta *distribución total, fragmentación y transparencia de replicación* permite que el usuario especifique una consulta o una petición de actualización en el esquema de la Figura 5.5 exactamente igual que si el DBMS estuviera centralizado. En el caso de actualizaciones, el DDBMS es responsable de mantener la *consistencia entre los elementos replicados* usando uno de los algoritmos de control de concurrencia distribuida comentados en la Sección 25.5. Para las consultas, un módulo de **descomposición de consulta** debe partir, o **descomponer**, esas consultas en **subconsultas** que puedan ejecutarse individualmente en los distintos sitios. Por otro lado, es preciso generar un procedimiento para combinar los resultados de estas subconsultas y formar así el resultado final de la consulta global. Siempre que el DDBMS determina que un elemento referenciado en la consulta está replicado debe elegir, o **materializar**, una réplica particular durante la ejecución de la misma. Para determinar qué réplicas incluyen los datos referenciados en la consulta, el DDBMS utiliza la información de fragmentación, replicación y distribución almacenada en el catálogo DDBMS. En el caso de fragmentación vertical, la lista de atributos de cada fragmento se mantiene en el catálogo. Para la fragmentación horizontal se mantiene, por cada fragmento, una condición especial llamada a veces **guardián**. Esta condición es, básicamente, una selección que especifica las tuplas existentes en el fragmento; el nombre de guardián hace referencia a que *sólo las tuplas que satisfacen esta condición* se almacenan en el fragmento. En el caso de fragmentos mixtos, se mantienen tanto la lista de atributos como el guardián en el catálogo.

En nuestro ejemplo anterior, las condiciones guardián de los fragmentos del sitio 1 (Figura 5.6) son TRUE (todas las tuplas), y la lista de atributos es $*$ (todos los atributos). Para los fragmentos mostrados en la Figura 25.3 contamos con las condiciones guardián y la lista de atributos mostrados en la Figura 25.7. Cuando el DDBMS descompone una petición de actualización, puede determinar qué fragmentos debe poner al día examinando sus condiciones guardián. Por ejemplo, cuando un usuario solicita insertar una nueva tupla EMPLEADO <‘Alejandro’, ‘Pérez’, ‘Cerezo’, ‘34567123’, ‘22-ABR-64’, ‘Cerro Verde 12’, H, 33000, ‘98765432’, 4>, ésta debe ser descompuesta por el DDBMS en dos peticiones: la primera inserta la tupla completa en el fragmento EMPLEADO del sitio 1 mientras que la segunda añade la tupla proyectada <‘Alejandro’, ‘Pérez’, ‘Cerezo’, ‘34567123’, 33000, ‘98765432’, 4> en el fragmento EMPD4 del sitio 3.

Para descomponer la consulta, el DDBMS puede determinar qué fragmentos pueden contener las tuplas requeridas comparando la condición de esa consulta con el guardián. Por ejemplo, consideremos la consulta C: *recuperar los nombres y horas semanales de cada empleado que trabaja para algún proyecto controlado por el departamento 5*; todo esto puede especificarse en SQL mediante el esquema mostrado en la Figura 5.5 del siguiente modo:

```
Q:  SELECT    Nombre, Apellido1, Horas
FROM          EMPLEADO, PROYECTO, TRABAJA_EN
WHERE         NumDptoProyecto=5 AND NumProyecto=NumProy AND DniEmpleado=Dni;
```

Supongamos que la consulta se envía al sitio 2, que es el lugar donde se precisa el resultado. El DDBMS puede determinar, a partir de la condición guardián de PROYS5 y TRABAJA_EN5 que todas las tuplas que satisfacen las condiciones (NumDptoProyecto = 5 AND NumProyecto = NumProy) residen en el sitio 2. Por tanto, puede descomponer la consulta en las siguientes subconsultas de álgebra relacional:

$$T_1 \leftarrow \pi_{\text{DniEmpleado}}(\text{PROYS5}\ _{\text{NumProyecto}=\text{NumProy}}\text{TRABAJA_EN5})$$
$$T_2 \leftarrow \pi_{\text{DniEmpleado, Nombre, Apellido2}}(T_1 \bowtie_{\text{DniEmpleado}=\text{Dni}} \text{EMPLEADO})$$
$$\text{RESULTADO} \leftarrow \pi_{\text{Nombre, Apellido2, Horas}}(T_2 * \text{TRABAJA_EN5})$$

Esta descomposición puede usarse para ejecutar la consulta usando una estrategia de *semijoin*. El DDBMS, gracias a las condiciones guardián de PROYS5, conoce exactamente las tuplas que satisfacen (NumDptoProyecto = 5) y sabe que TRABAJA_EN5 contiene todas las tuplas que deben concatenarse a PROYS5; por tanto, la subconsulta $T_1$ puede llevarse a cabo en el sitio 2 y enviar la columna proyectada DniEmpleado al sitio 1. $T_2$ puede ejecutarse entonces en el sitio 1, y devolver el resultado al sitio 2, que es el

lugar en el que se calculará y se mostrará al usuario el resultado completo de la consulta. Existe una estrategia alternativa que sería enviar la propia consulta C al sitio 1, que contiene todas las tuplas de la base de datos, donde sería ejecutada localmente y desde donde se devolvería el resultado al sitio 2. El optimizador de consultas debería estimar los costes de ambas estrategias y elegir la que tuviese un coste menor.

## 25.5 El control de la concurrencia y la recuperación en bases de datos distribuidas

Acerca del control de la concurrencia y la recuperación, son muchos los problemas que aparecen en un DBMS distribuido que no se dan en un entorno centralizado. Entre ellos podemos citar los siguientes:

- **Tratar con *múltiples copias* de los datos.** El método de control de la concurrencia es responsable de mantener la consistencia de todas esas copias. El método de recuperación debe encargarse de que una copia sea coherente con el resto en caso de producirse un fallo en el sitio que la contiene y se efectúe una restauración posterior.
- **Fallo de los sitios individuales.** En caso de ser posible, el DDBMS debe seguir operando con los sitios que están en funcionamiento cuando uno o más de estos sitios fallan. Cuando se lleva a cabo la restauración del mismo, su copia de los datos debe actualizarse con el resto de los sitios antes de reinsertarse en el sistema.
- **Fallo de los enlaces de comunicación.** El sistema debe ser capaz de tratar con los fallos que se produzcan en los enlaces de comunicación que conectan los sitios. Un caso extremo de este problema es que se produzca un **particionamiento de la red**. Esta situación divide los sitios en una o más particiones, de modo que los sitios sólo pueden comunicarse con otros de la misma partición, pero no con los que se encuentran en otras particiones.
- **Confirmación distribuida.** Pueden aparecer problemas a la hora de confirmar (*commit*) una transacción que está accediendo a las bases de datos almacenadas en varios sitios si alguno de ellos falla durante el proceso. Para afrontar esta situación suele usarse el **protocolo de confirmación en dos fases** (consulte el Capítulo 19).
- **Estancamiento distribuido.** El interbloqueo (*deadlock*) puede producirse entre varios sitios, por lo que deben extenderse las técnicas para gestionarlo.

El control de la concurrencia distribuida y las técnicas de recuperación deben encargarse de estos y otros problemas. En las siguientes subsecciones revisaremos algunas de las estrategias que se han sugerido para ello.

### 25.5.1 Control de la concurrencia distribuida basada en una copia diferenciada de un elemento de datos

Se han propuesto varios métodos de control de la concurrencia para gestionar los datos replicados en una base de datos distribuida que amplían los propuestos para las bases de datos centralizadas. Vamos a estudiar estas técnicas en el contexto de un bloqueo (*locking)* centralizado extendido. Extensiones similares se aplican a las otras técnicas de control de la concurrencia. La idea que se esconde tras esto es designar una *copia particular* de cada elemento de datos como una **copia diferenciada**. El bloqueo de este elemento de datos está asociado *con la copia diferenciada*, y todas las peticiones de bloqueos y desbloqueos se envían al sitio que contiene esta copia.

Existen varios métodos basados en esta idea que difieren en su forma de elegir sus copias diferenciadas. En la **técnica de sitio primario**, todas las copias diferenciadas se mantienen en el mismo sitio. Una modificación de este planteamiento es el sitio primario con sitio de ***backup***. Otro acercamiento es el método de **copia pri-**

**maria**, en el que las copias diferenciadas de cada elemento pueden almacenarse en sitios diferentes, cada uno de los cuales actúa básicamente como el **sitio coordinador** para el control de concurrencia de ese elemento. Veremos a continuación estas técnicas.

**Técnica de sitio primario.** En este método se designa un único **sitio primario** para que sirva como **sitio coordinador** *de todos los elementos de la base de datos.* Por tanto, todos los bloqueos se envían a este sitio, así como todas las peticiones de bloqueos y desbloqueos. Así, este método es una extensión del bloqueo centralizado. Por ejemplo, si todas las transacciones siguen el protocolo de bloqueo de dos fases, la seriabilidad está garantizada. La ventaja de este acercamiento es ser una extensión simple del esquema centralizado y que, por consiguiente, no es demasiado complejo. Sin embargo, cuenta con ciertas desventajas inherentes. Una de ellas es que todas las peticiones de bloqueo se envían a un único sitio, probablemente sobrecargándolo y provocando un cuello de botella en el sistema. Una segunda desventaja es que el fallo del sitio primario paraliza el sistema, ya que toda la información sobre bloqueos se mantiene allí. Esto puede limitar la fiabilidad y disponibilidad de toda la estructura.

Aunque el acceso a todos los bloqueos se realiza en el sitio primario, los propios elementos pueden ser accesibles en cualquiera de los sitios en los que residan. Por ejemplo, una vez que una transacción obtiene desde el sitio primario un Read_lock (bloqueo de lectura) sobre un dato puede acceder a cualquier copia del mismo. Sin embargo, cuando esa transacción consigue un Write_lock (bloqueo de escritura) y actualiza el dato, el DDBMS será el único responsable de actualizar *todas las copias* de ese dato antes de liberar el bloqueo.

**Sitio primario con sitio de respaldo (*backup*).** Este método solventa la segunda de las desventajas del método de sitio primario al designar un **sitio de respaldo**. Toda la información sobre bloqueos se mantiene tanto en el sitio primario como en el de respaldo para que, si falla el primero, el segundo tome el control y se elija un nuevo sitio de respaldo. Esto simplifica el proceso de recuperación cuando falla el sitio primario, ya que, en este caso, el de respaldo toma el control y la operativa puede reemprenderse después de elegido un nuevo sitio de respaldo y copiada la información sobre bloqueos al mismo. Sin embargo, se ralentiza el proceso de adquisición de bloqueos, ya que todas estas peticiones y la asignación de los bloqueos deben grabarse *tanto en el sitio primario como en el de respaldo* antes de enviar una respuesta a la transacción que lo solicita. El problema de este método es que los sitios se sobrecargan con las peticiones con la consiguiente ralentización del sistema.

**Técnica de copia primaria.** Este método intenta distribuir la carga de la coordinación de bloqueos al disponer de copias diferenciadas de los datos *almacenadas en diferentes sitios*. El fallo de un sitio sólo afecta a aquellas transacciones que están accediendo a los bloqueos sobre los elementos cuyas copias primarias residen en ese sitio, manteniendo intactas el resto de transacciones. Este método puede usar también sitios de respaldo para mejorar la fiabilidad y la disponibilidad.

**Elección de un nuevo sitio coordinador en el caso de producirse un fallo.** Siempre que falla un sitio coordinador en cualquiera de las técnicas anteriores, el que aún permanece activo debe elegir un nuevo coordinador. En el caso de tratarse de un sitio primario *sin* sitio de respaldo, todas las transacciones en ejecución deben detenerse y reactivarse por medio de un tedioso proceso de recuperación. Parte de este proceso de recuperación implica la elección de un nuevo sitio primario, la creación de un proceso gestor de bloqueos y la grabación en ese sitio de toda la información de bloqueo. Para el caso de los métodos que utilizan sitios de respaldo, el procesamiento de una transacción queda en suspenso mientras el sitio de respaldo es designado como sitio primario y se elige uno nuevo de respaldo y se le envía copia de toda la información de bloqueo.

Si un sitio de respaldo *X* está próximo a convertirse en el nuevo sitio primario, *X* puede elegirlo de entre todos los que están operativos en el sistema. Sin embargo, si no existe sitio de respaldo, o si ambos están caídos, puede usarse un proceso llamado **elección** para elegir el nuevo sitio coordinador. En este proceso, cualquier sitio *Y* que intente comunicar repetidamente con el sitio coordinador sin obtener respuesta puede asumir que éste está caído e iniciar el proceso de elección. Para ello, expide un mensaje a todos los sitios en ejecución

proponiéndose él mismo como nuevo sitio coordinador. Tan pronto como *Y* reciba una mayoría de votos afirmativos, puede declararse como el nuevo sitio coordinador. El propio algoritmo de elección es algo complejo, pero lo relatado es la idea principal que se esconde tras él. El algoritmo también resuelve cualquier intento de que dos o más sitios quieran convertirse en coordinadores al mismo tiempo. Las referencias de la bibliografía seleccionada de este capítulo comentan el proceso con más detalle.

### 25.5.2 Control de concurrencia distribuida basada en la votación

Todos los métodos de control de la concurrencia para elementos replicados que hemos visto hasta ahora usan el principio de una copia diferenciada que mantiene los bloqueos para ese elemento. En el **método por votación** no existe esa copia; en lugar de ello, se envía una petición de bloqueo a todos los sitios que incluye una muestra del elemento de datos. Cada copia mantiene su propio bloqueo y puede otorgar o denegar la petición sobre él. Si una transacción que solicita un bloqueo recibe el permiso para ello de una *mayoría* de las copias, se queda con ese bloqueo e informa a *todas* ellas de la obtención del mismo. En caso de que la transacción no reciba esa mayoría de votos dentro de un cierto *periodo de tiempo muerto,* cancela su petición e informa a todos los sitios de la cancelación.

El método por votación está considerado como un auténtico sistema de control de concurrencia distribuida, ya que la responsabilidad de una decisión reside en todos los sitios involucrados. Diversas simulaciones han demostrado que la votación genera más tráfico de mensajes entre los sitios que los métodos de copia diferenciada. Si el algoritmo tiene en cuenta los posibles fallos producidos durante el proceso de votación, su complejidad se eleva enormemente.

### 25.5.3 Recuperación distribuida

El proceso de recuperación en una base de datos distribuida es algo complejo, y aquí sólo ofrecemos una breve introducción. En ciertos casos, es incluso más complicado determinar si un sitio está caído sin intercambiar numerosos mensajes con otros sitios. Por ejemplo, supongamos que el sitio *X* envía un mensaje al *Y* esperando recibir una respuesta que no llega. Existen varias explicaciones para esto:

- El mensaje no fue entregado a *Y* debido a un fallo en la comunicación.
- El sitio *Y* está caído y no pudo responder.
- El sitio *Y* está funcionando y envía una respuesta, pero ésta no fue entregada.

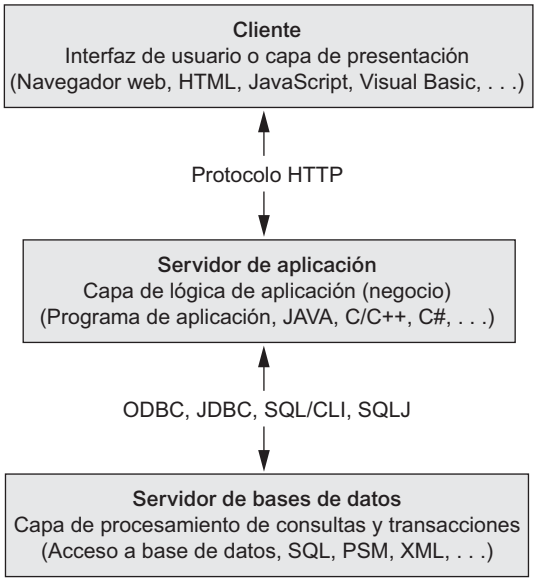
Sin otra información, o el envío de mensajes adicionales, resulta muy complicado determinar qué ha ocurrido realmente.

Otro problema que aparece en la recuperación distribuida es la confirmación distribuida. Cuando una transacción está actualizando datos en varios sitios, no puede realizar una confirmación hasta que no esté seguro de que el efecto de la misma no pueda perderse en *cada sitio*. Esto significa que cada uno de ellos debe haber registrado permanentemente los efectos locales de las transacciones en el *log* de sitio local en disco. El protocolo de confirmación en dos fases, comentado en la Sección 19.6, suele emplearse a veces para asegurar la exactitud de la confirmación distribuida.

## 25.6 Una aproximación a la arquitectura cliente-servidor de tres niveles

Tal y como apuntamos en la introducción del capítulo, un DDBMS a gran escala no está diseñado para soportar todos los tipos de funcionalidades que hemos visto hasta el momento. En su lugar, las aplicaciones de bases de datos distribuidas se han desarrollado en el contexto de las arquitecturas cliente-servidor. En la Sección 2.5

**Figura 25.8.** La arquitectura cliente-servidor de tres niveles.




vimos la arquitectura cliente-servidor de dos capas o niveles, aunque en la actualidad es más normal emplear la de tres, especialmente en aplicaciones web. Esta arquitectura se ilustra en la Figura 25.8.

En la arquitectura cliente-servidor de tres niveles se dan las siguientes capas:

1. **Capa de presentación (cliente).** Proporciona al usuario la interfaz e interactúa con él. Los programas de esta capa presentan al cliente interfaces web o formularios que sirven como conexión con la aplicación. Con frecuencia se emplean navegadores web, y entre los lenguajes utilizados se pueden citar HTML, JAVA, JavaScript, PERL o Visual Basic. Esta capa manipula las entradas, las salidas y la navegación aceptando comandos de usuario y mostrando la información necesaria, que suele tener la forma de páginas web estáticas o dinámicas. Las últimas se usan cuando la interacción implica el acceso a bases de datos. Cuando se utiliza una interfaz web, esta capa suele comunicarse con la de aplicación mediante el protocolo HTTP.

2. **Capa de aplicación (lógica de negocio).** Esta capa programa la lógica de aplicación. Por ejemplo, las consultas pueden ser formuladas en base a datos introducidos por el cliente, o el resultado de las mismas puede formatearse y enviarse al cliente para su presentación. Las comprobaciones de seguridad o la verificación de la identidad son funcionalidades adicionales que también pueden llevarse a cabo en esta capa. La capa de aplicación puede interactuar con una o más bases de datos o fuentes de información mediante ODBC, JDBC, SQL/CLI o cualquier otra técnica de acceso.

3. **Servidor de bases de datos.** Esta capa controla las consultas y peticiones de actualizaciones procedentes de la capa de aplicación, procesa las solicitudes y envía los resultados. Por lo general, se utiliza SQL para acceder a la base de datos si ésta es relacional o de objetos relacionales; también pueden invocarse los procedimientos almacenados. Los resultados de las consultas (y las propias consultas) pueden estar formateados en XML (consulte el Capítulo 27) cuando son trasmitidas entre el servidor de aplicaciones y el de base de datos.

La forma exacta de dividir la funcionalidad DBMS entre el cliente, el servidor de aplicación y el de base de datos puede variar. Lo más común es incluir la operatividad de un DBMS centralizado a nivel del servidor de base de datos. Son varios los productos DBMS comerciales que siguen este planteamiento, donde se propor-

ciona un servidor **SQL**. El servidor de aplicación debe formular entonces las consultas SQL apropiadas y conectar con el servidor de base de datos cuando lo precise. El cliente suministra el procesamiento para las interacciones con la interfaz de usuario. Ya que SQL es un estándar relacional, son muchos los servidores SQL que pueden aceptar comandos SQL a través de estándares como ODBC, JDBC y SQL/CLI (consulte el Capítulo 9).

En esta arquitectura, el servidor de aplicación debe hacer referencia también a un diccionario de datos que incluye la información acerca de la distribución de los mismos entre los distintos servidores SQL, así como módulos para la descomposición de una consulta global en otras locales más pequeñas que puedan ejecutarse en los distintos sitios. La interacción entre los servidores de aplicación y de base de datos puede producirse del siguiente modo durante el procesamiento de una consulta SQL:

1. El servidor de aplicación formula una consulta basada en la información introducida por el usuario a través de la capa cliente y la descompone en varias consultas de sitio independientes. Cada una de estas consultas es enviada al sitio del servidor de base de datos apropiado.
2. Cada servidor de base de datos procesa la consulta local y envía el resultado al sitio del servidor de aplicación. Cada vez más, se está recomendando XML como el estándar para el intercambio de datos (consulte el Capítulo 27), por lo que el servidor de base de datos puede formatear el resultado de la consulta en XML antes de enviarlo al servidor de aplicación.
3. El servidor de aplicación combina los resultados de las subconsultas para generar la salida final de la consulta propuesta, formateándola en HTML o en alguna otra forma aceptada por el sitio cliente, y se la envía para su visualización.

El servidor de aplicación es responsable de generar un plan de ejecución distribuida para una consulta o transacción multi-sitio y de supervisar la ejecución distribuida enviando comandos a los servidores. Estos comandos incluyen las consultas locales y las transacciones a ejecutar, así como los necesarios para transmitir datos a otros clientes o servidores. Otra función controlada por el servidor de aplicación (o coordinador) es la de asegurar la consistencia de las copias replicadas de los datos empleando técnicas de control de concurrencia distribuida (o global). El servidor de aplicación debe asegurar también la atomicidad de las transacciones globales llevando a cabo recuperaciones globales cuando fallen ciertos sitios. Veremos el control de la concurrencia y la recuperación distribuida en la Sección 25.5.

Si el DDBMS tiene la capacidad de *ocultar* los detalles de la distribución de datos al servidor de aplicación, entonces le permite ejecutar la consultas y las transacciones globales como si se tratase de una base de datos centralizada, sin necesidad de especificar los sitios en los que residen los datos referenciados en la consulta o la transacción. Esta propiedad se conoce como **transparencia de distribución**. Algunos DDBMSs no disponen de ella, lo que obliga a que las aplicaciones tengan que estar pendientes de los detalles de la distribución de datos.

## 25.7 Bases de datos distribuidas en Oracle[7]

En la arquitectura cliente-servidor, la base de datos Oracle está dividida en dos partes: un *front end* (frontal) como parte cliente y un *back end* como parte servidor. La parte cliente es la aplicación de base de datos *front-end* que interactúa con el usuario. El cliente no tiene ninguna responsabilidad en el acceso a los datos y su función se reduce a la manipulación, la solicitud, el procesamiento y la presentación de la información gestionada por el servidor. La parte servidor ejecuta Oracle y manipula las funciones relacionadas con el acceso compartido concurrente. Acepta sentencias SQL y PL/SQL procedentes de las aplicaciones cliente, las procesa y devuelve los resultados al usuario que los solicitó. Las aplicaciones cliente-servidor Oracle ofrecen transparencia de localización para que el usuario no sepa nada de la ubicación de los datos; a ello contribuyen

[7] El argumento está basado en las posibilidades de la versión 8. Oracle 10g se ha diseñado para ofrecer una operativa distribuida óptima.

varias características como las vistas, los sinónimos y los procedimientos. La designación global para referirse a una única tabla se consigue usando <NOMBRE_TABLA@NOMBRE_BASE_DE_DATOS>.

Oracle usa un protocolo de confirmación en dos fases para gestionar las transacciones distribuidas concurrentes. La sentencia COMMIT dispara el mecanismo. RECO (restablecimiento) anula el proceso y resuelve automáticamente el resultado de estas transacciones distribuidas en las que la confirmación se interrumpió. El RECO de cada servidor Oracle local consolida o anula cualquier transacción distribuida *dudosa* en todos los nodos implicados. Para fallos a gran escala, Oracle permite que cada DBA local consolide o anule manualmente cualquier operación dudosa y libere los recursos. La consistencia global puede mantenerse restaurando la base de datos de cada sitio a un estado predeterminado fijo del pasado.

La arquitectura de base de datos distribuida de Oracle aparece en la Figura 25.9. Un nodo de este sistema puede actuar como cliente, servidor o ambos, dependiendo de la situación. La figura muestra dos sitios en los que se mantienen las bases de datos HQ (central) y Ventas. Por ejemplo, para la aplicación que está funcionando en las oficinas centrales y una sentencia SQL que use datos locales (como DELETE FROM DEPT ...), el computador HQ actúa como un servidor, mientras que si usamos datos remotos (por ejemplo, INSERT INTO EMP@VENTAS), HQ se comporta como un cliente.

Todas las bases de datos Oracle de un DDBS utilizan el software de red de Oracle Net8 para la comunicación entre ellas. Net8 permite a las bases de datos comunicarse a través de una red para soportar transacciones distribuidas y remotas. Empaqueta las sentencias SQL en uno de los muchos protocolos de comunicación para facilitar el enlace cliente-servidor y después devuelve los resultados al cliente del mismo modo. Cada base de datos cuenta con un nombre global único asignado por una configuración jerárquica de nombres de dominio de red.

Oracle soporta enlaces que definen una ruta de comunicación en un único sentido desde una base de datos Oracle a otra. Por ejemplo,

```
CREATE DATABASE LINK ventas.us.americas;
```

establece una conexión a la base de datos de ventas de la Figura 25.9 bajo el dominio de red us que pertenece a americas.

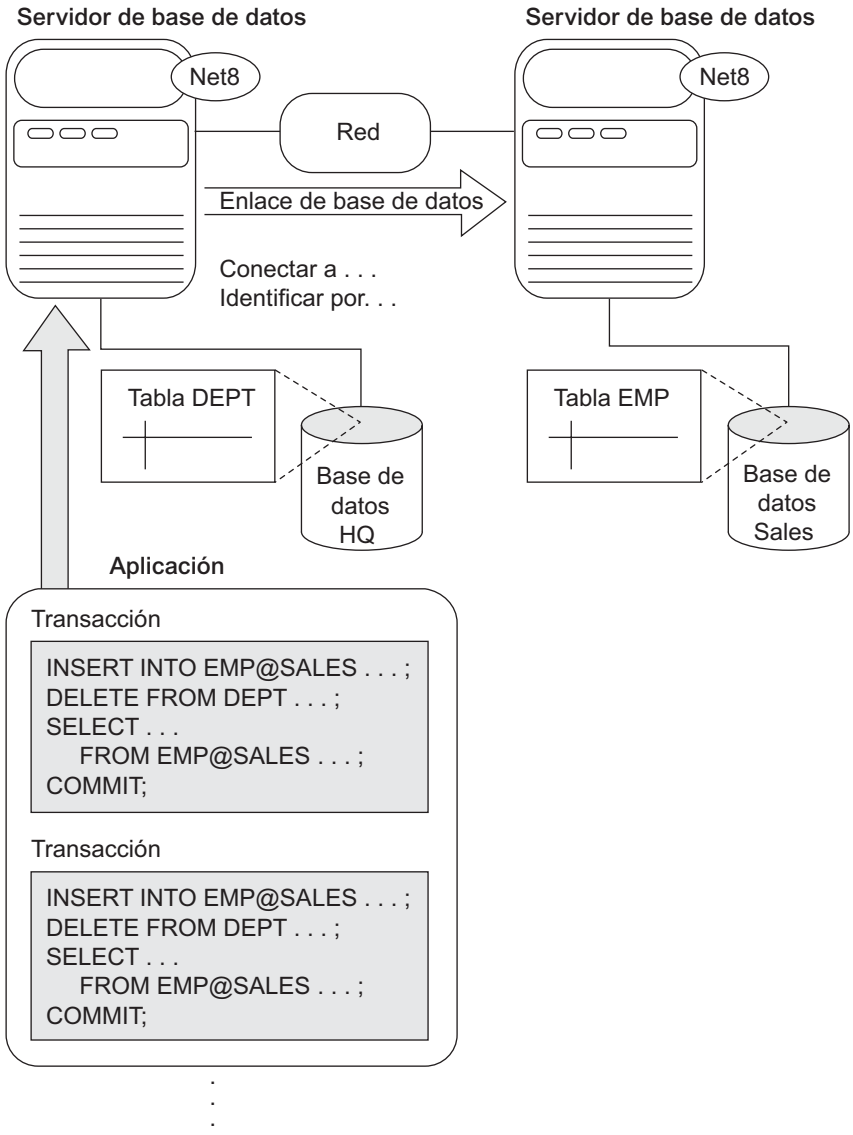
Los datos en un DDBS Oracle pueden replicarse usando instantáneas (*snapshots*) o tablas maestras replicadas. La replicación se consigue a los siguientes niveles:

- **Replicación básica.** Las copias de las tablas están administradas para un acceso de sólo-lectura. En las actualizaciones, a los datos se accede a través de un único sitio primario.
- **Replicación avanzada (simétrica).** Este método amplía las posibilidades de la replicación básica permitiendo que las aplicaciones actualicen copias de las tablas a través de un DDBS replicado. Los datos pueden leerse y actualizarse en cada sitio. Todo esto precisa de un software adicional llamado opción de replicación avanzada de Oracle. Una **instantánea** genera una copia de una parte de la tabla por medio de una *consulta de definición de instantánea*. La definición de una instantánea simple tiene este aspecto:

```
CREATE SNAPSHOT ORDENES_VENTAS AS
SELECT * FROM ORDENES_VENTAS@hq.us.americas;
```

Oracle agrupa las instantánea en grupos de actualización. Especificando un determinado valor, la instantánea se refresca cada ese intervalo automática y periódicamente hasta en diez **SNPs (Proceso de actualización de instantánea, *Snapshot Refresh Processes***). Si la consulta de definición de la instantánea contiene una cláusula como GROUP BY o CONNECT BY u operaciones de conjunto o de concatenación, lo que obtenemos es una **instantánea compleja** que precisa de procesamiento adicional. Oracle (hasta su versión 7.3) también soportaba instantáneas ROWID que estaban basadas en los identificadores físicos de las filas de la tabla maestra.

**Figura 25.9.** Sistemas de bases de datos distribuidas Oracle.




*Fuente*: Oracle (1997a). Copyright © Oracle Corporation 1997. Todos los derechos reservados.

**Bases de datos heterogéneas en Oracle.** En un DDBS heterogéneo, al menos una de las bases de datos no está en sistema Oracle. **Oracle Open Gateways** ofrece acceso a estas bases de datos desde un servidor Oracle, el cual utiliza enlaces de bases de datos para acceder a la información o ejecutar procedimientos remotos de un sistema que no es Oracle. Entre las características de Open Gateways podemos citar las siguientes:

- **Transacciones distribuidas.** Bajo el mecanismo de confirmación en dos fases, las transacciones pueden abarcar a sistemas Oracle y a otros que no lo sean.
- **Acceso SQL transparente.** Las sentencias SQL lanzadas por una aplicación son convertidas de manera transparente a instrucciones SQL para que sean entendidas por un sistema no Oracle.

- **Procedimientos almacenados y SQL *pass-through*.** Una aplicación puede acceder directamente a un sistema no Oracle usando esa versión de SQL del sistema. Los procedimientos almacenados en un sistema basado en SQL pero que no sea Oracle son tratados como si fueran procedimientos PL/SQL remotos.
- **Optimización de consulta global.** De la información de cardinalidad, índices, etc. del sistema no Oracle da cuenta el optimizador de consulta del servidor Oracle para llevar a cabo la optimización de consulta global.
- **Acceso procesal.** El servidor Oracle accede a los sistemas procedurales, como mensajería o entornos de encolamiento, a través de llamadas PL/SQL de procedimiento remoto.

Además de todo lo anterior, las referencias del diccionario de datos del sistema no Oracle son traducidas para que aparezcan como parte integrante del diccionario del servidor Oracle. Para conectar bases de datos multilingües se realizan conversiones entre los conjuntos de caracteres de cada lenguaje.

# 25.8 Resumen

En este capítulo hemos ofrecido una introducción a las bases de datos distribuidas. Éste es un tema muy amplio, y sólo hemos mostrado algunas de las técnicas básicas usadas en ellas. En primer lugar vimos las razones para realizar una distribución y las ventajas potenciales de estas bases de datos sobre los sistemas centralizados. Definimos también el concepto de transparencia de distribución y los conceptos relacionados de transparencia de fragmentación y de replicación. Abordamos los problemas de diseño relacionados con la fragmentación, replicación y distribución de datos, y distinguimos entre fragmentos de relaciones horizontales y verticales. Argumentamos acerca del uso de la replicación de datos para mejorar la fiabilidad y la disponibilidad del sistema. Catalogamos los DDBMSs usando distintos criterios como el grado de homogeneidad de los módulos software y el grado de autonomía local. Vimos con cierto detalle las complicaciones inherentes en la administración de las bases de datos federadas, centrándonos en las necesidades de soporte de los distintos tipos de autonomías y prestando atención a la heterogeneidad de la semántica.

Describimos algunas de las técnicas usadas para el procesamiento de una consulta distribuida y examinamos los costes de comunicación entre sitios, los cuales están considerados como uno de los principales factores en la optimización de una consulta distribuida. Comparamos distintas técnicas para ejecutar consultas y presentamos la técnica de *semijoin* para enlazar relaciones que residen en sitios diferentes. Comentamos brevemente el control de concurrencia y las técnicas de recuperación usadas en los DDBMSs. Revisamos algunos de los problemas adicionales que deben ser tratados en un entorno distribuido y que no aparecen en otro centralizado.

Para terminar, revisamos los conceptos relativos a la arquitectura cliente-servidor y los relacionamos con las bases de datos distribuidas, y describimos algunas de las facilidades ofrecidas por Oracle para el soporte a las bases de datos distribuidas.

## Preguntas de repaso

**25.1.** ¿Cuáles son las razones principales y las ventajas potenciales de las bases de datos distribuidas?

**25.2.** ¿Qué funciones adicionales debe tener un DDBMS sobre un DBMS centralizado?

**25.3.** ¿Cuáles son los módulos de software principales de un DDBMS? Exponga las funciones principales de cada uno de estos módulos en el contexto de la arquitectura cliente-servidor.

**25.4.** ¿Qué es un fragmento de una relación? ¿Cuáles son los principales tipos de fragmentos? ¿Por qué es útil la fragmentación en el diseño de una base de datos distribuida?

**25.5.** ¿Por qué es útil la replicación en un DDBMS? ¿Qué tipo de unidades de datos son replicadas normalmente?

**25.6.** ¿Qué significa la *ubicación de datos* en el diseño de una base de datos distribuida? ¿Cuáles son las unidades de datos que están distribuidas habitualmente entre los diferentes sitios?

**25.7.** ¿Cómo se especifica el particionamiento horizontal de una relación? ¿Cómo puede reponerse una relación a partir de un particionamiento horizontal completo?

**25.8.** ¿Cómo se especifica el particionamiento vertical de una relación? ¿Cómo puede reponerse una relación a partir de un particionamiento vertical completo?

**25.9.** Comente lo que quieren decir los siguientes términos: *grado de homogeneidad de un DDBMS*, *grado de autonomía local de un DDBMS*, *DBMS federado*, *transparencia de distribución*, *transparencia de fragmentación*, *transparencia de replicación y sistema de "múltiples bases de datos"*.

**25.10.** Exponga los problemas relativos al nombrado en las bases de datos distribuidas.

**25.11.** Comente las diferentes técnicas existentes para la ejecución de una *equijoin* de dos ficheros localizados en sitios distintos. ¿Cuáles son los factores primordiales que afectan al coste de la transferencia de datos?

**25.12.** Aborde el método de *semijoin* para la ejecución de una *equijoin* de dos ficheros localizados en sitios diferentes. ¿Bajo qué condiciones es eficiente esta estrategia?

**25.13.** Exponga los factores que afectan a la descomposición de una consulta. ¿Cómo se usan la condición guardián y las listas de atributos durante el proceso de descomposición de una consulta?

**25.14.** ¿En qué se diferencian la descomposición de una petición de actualización de la de una consulta? ¿Cómo se usan la condición guardián y las listas de atributos durante el proceso de descomposición de una petición de actualización?

**25.15.** Aborde los factores que no aparecen en los sistemas centralizados y que afectan al control de la concurrencia y a la recuperación en los sistemas distribuidos.

**25.16.** Compare el método de sitio primario con el de copia primaria para el control de la concurrencia distribuida. ¿Cómo afecta a cada uno de ellos el uso de un sitio de respaldo?

**25.17.** ¿Cuándo se usan la votación y las elecciones en las bases de datos distribuidas?

**25.18.** ¿Cuáles son los componentes software de un DDBMS cliente-servidor? Compare las arquitecturas cliente-servidor de dos y tres niveles.

## Ejercicios

**25.19.** Consideremos la distribución de datos de la base de datos EMPRESA, donde los fragmentos de los sitios 2 y 3 aparecen en la Figura 25.3 y los del 1 están en la Figura 5.6. Por cada una de las siguientes consultas muestre, al menos, dos estrategias para descomponerlas y ejecutarlas. ¿Bajo qué condiciones funcionaría mejor cada una de sus estrategias?

**a.** Por cada empleado del departamento 5, recuperar su nombre y el de sus subordinados.

**b.** Mostrar los nombres de todos los empleados que trabajan en el departamento 5 pero que lo hacen en algún proyecto *no* controlado por dicho departamento.

**25.20.** Consideremos las siguientes relaciones:

LIBROS(NúmeroLibro, PrimerAutor, Tema, CantidadTotal, Precio)
LIBRERÍA(NúmeroLibrería, Ciudad, Estado, Zip, ValorInventario)
STOCK(NumLibrería, NúmeroLibro, Cantidad)

CantidadTotal es el número total de libros en depósito, mientras que ValorInventario es el valor total del inventario de la tienda en euros.

**a.** Ponga un ejemplo de dos predicados simples que pudieran ser significativos para el particionado horizontal de la relación LIBRERÍA.

**b.** ¿Cómo podría definirse un particionamiento horizontal derivado de STOCK en base a la división de LIBRERÍA?

**c.** Muestre predicados que permitan particionar LIBROS horizontalmente por tema.

**d.** Indique de qué forma podría dividirse STOCK partiendo de las particiones del apartado (b) y añadiendo los predicados de (c).

**25.21.** Consideremos una base de datos distribuida para una cadena de librerías llamada National Books que cuenta con tres sitios llamados ESTE, CENTRO y OESTE. Los esquemas de relación aparecen en la pregunta 24.20. Supongamos que LIBROS está fragmentado por las cantidades indicadas en Precio de la siguiente forma:

$B_1$: LIBRO1: Precio hasta 20 €

$B_2$: LIBRO2: Precio desde 20,01 € hasta 50 €

$B_3$: LIBRO3: Precio desde 50,01 € hasta 100 €

$B_4$: LIBRO4: Precio a partir de 100,01 €

De forma parecida, LIBRERÍAS está dividida por el código postal (CP) en:

$S_1$: ESTE: CP hasta el 35000

$S_2$: CENTRO: CP desde el 35001 al 70000

$S_3$: OESTE: CP desde el 70001 al 99999

Asumimos que STOCK es un fragmento derivado basado sólo en LIBRERÍA.

**a.** Consideremos la consulta:

```
SELECT NúmeroLibro, CantidadTotal
FROM Libros
WHERE Precio > 15 AND Precio < 55;
```

Asumimos que los fragmentos de LIBRERÍA no están replicados y están asignados en función a la región. Asumimos por otro lado que LIBROS está ubicado como:

ESTE: $B_1$, $B_4$

CENTRO: $B_1$, $B_2$

OESTE: $B_1$, $B_2$, $B_3$, $B_4$

Dando por hecho que la consulta fue emitida en ESTE, ¿qué subconsultas remotas genera? (Escribir en SQL).

**b.** Si cambiamos el precio de la obra cuyo NúmeroLibro = 1234 de 45 a 55 en el sitio CENTRO, ¿qué actualizaciones se generan? Escribir en español y después en SQL.

**c.** Muestre un ejemplo de consulta lanzada en OESTE que genere una subconsulta para CENTRO.

**d.** Escriba una consulta que implique una selección y una proyección en las relaciones anteriores y muestre dos posibles árboles de consulta que denoten distintas formas de ejecución.

**25.22.** Consideremos que ha recibido el encargo de proponer la arquitectura de base de datos de una gran organización (como General Motors) para consolidar todos los datos incluyendo las bases de datos heredadas (desde modelos de red a jerárquicos, los cuales se explicaron en los Apéndices D y E; no son necesarios conocimientos especiales de estos modelos), así como las relacionales, las cuales están geográficamente distribuidas de forma que puedan soportar aplicaciones globales. Asumimos que la primera alternativa es dejar todas las bases de datos tal y como están, mientras que la segunda es convertirlas primero a un esquema relacional para después soportar las aplicaciones sobre una base de datos integrada distribuida.

**a.** Dibuje dos diagramas esquemáticos para ambas alternativas que muestren los enlaces entre los esquemas apropiados. Para el primer caso, elegimos la exportación de los esquemas de cada base de datos y la construcción de esquemas unificados para cada aplicación.

**b.** Enumere los pasos a dar bajo cada alternativa desde la situación actual hasta que las aplicaciones globales sean viables.

**c.** Compare ambos planteamientos desde el punto de vista de:

i. Consideraciones de tiempo de diseño.

ii. Consideraciones de tiempo de ejecución.

## Bibliografía seleccionada

Los libros Ceri y Pelagatti (1984a) y Ozsu y Valduriez (1999) están dedicados a las bases de datos distribuidas. Halsaal (1996), Tannenbaum (1996) y Stallings (1997) tratan las comunicaciones de datos y las redes de computadores. Comer (1997) aborda las redes e Internet. Dewire (1993) es un libro de consulta acerca del procesamiento cliente-servidor. Ozsu y otros (1994) dispone de una colección de artículos sobre la administración de objetos distribuidos.

El diseño de bases de datos distribuidas se ha dirigido en términos de fragmentación vertical, ubicación y replicación. Ceri y otros (1982) definió el concepto de fragmentos horizontales minterm. Ceri y otros (1983) desarrolló una programación entera basada en el modelo de optimización para la fragmentación horizontal y la ubicación. Navathe y otros (1984) desarrolló algoritmos para la fragmentación vertical basados en la afinidad de atributo y mostró una variedad de contextos para la ubicación de un fragmento vertical.

Wilson y Navathe (1986) presenta un modelo analítico para la ubicación óptima de fragmentos. Elmasri y otros (1987) comenta la fragmentación para el modelo ECR; Karlapalem y otros (1994) discute los problemas del diseño distribuido de bases de datos de objetos. Navathe y otros (1996) aborda la fragmentación mixta por la combinación de la horizontal y la vertical; Karlapalem y otros (1996) presenta un modelo para el rediseño de bases de datos distribuidas.

El procesamiento, la optimización y la descomposición de una consulta distribuida son temas tratados en Hevner y Yao (1979), Kerschberg y otros (1982), Apers y otros (1983), Ceri y Pelagatti (1984), y Bodorick y otros (1992). Bernstein y Goodman (1981) comenta la teoría que hay tras el procesamiento de una *semijoin*. Wong (1983) aborda el uso de relaciones en la fragmentación de una relación. El control de concurrencia y los esquemas de recuperación se tratan en Bernstein y Goodman (1981a). Kumar y Hsu (1998) escribieron algunos artículos relacionados con la recuperación en bases de datos distribuidas. Las elecciones en sistemas distribuidos se comentan en García-Molina (1982). Lamport (1978) trata los problemas existentes con la generación de marcas de tiempo únicas en un sistema distribuido.

Thomas (1979) presentó una técnica de control de concurrencia basada en la votación. Gifford (1979) propuso el uso de la mayoría ponderada, y Paris (1986) describe un método llamado votación con testigos. Jajodia y Mutchler (1990) comentan la votación dinámica. Bernstein y Goodman (1984) proponen una técnica llamada *copia disponible*, y otra que usa la idea de un grupo fue presentada en ElAbbadi y Toueg (1988). Otros trabajos que tratan la replicación de datos son Gladney (1989), Agrawal y ElAbbadi (1990), ElAbbadi y Toueg (1990), Kumar y Segev (1993), Mukkamala (1989), y Wolfson y Milo (1991). Bassiouni (1988) aborda los protocolos optimistas para el control de concurrencia DDB. García-Molina (1983) y Kumar y Stonebraker (1987) muestran técnicas que usan la semántica de las transacciones. Las técnicas de control de la concurrencia basadas en el bloqueo y las copias diferenciadas están presentes en Menasce y otros (1980), y en Minoura y Wiederhold (1982). Obermark (1982) presenta algoritmos para la detección de un interbloqueo distribuido.

Kohler (1981) muestra una encuesta acerca de las técnicas de recuperación en sistemas distribuidos. Reed (1983) conferencia sobre las acciones atómicas en datos distribuidos. Un libro editado por Bhargava (1987) presenta varios métodos y técnicas para la concurrencia y la fiabilidad en los sistemas distribuidos.

Los sistemas de bases de datos federadas fueron definidos en primer lugar en McLeod y Heimbigner (1985). Las técnicas para la integración del esquema en bases de datos federadas están presentes en Elmasri y otros (1986), Batini y otros (1986), Hayne y Ram (1990) y Motro (1987). Elmagarmid y Helal (1988) y Gamal-Eldin y otros (1988) comentan el problema de la actualización en los DDBSs heterogéneos. Los problemas de una base de datos distribuida heterogénea se comentan en Hsiao y Kamel (1989). Sheth y Larson (1990) presentan una exhaustiva encuesta de la administración de una base de datos federada.

Recientemente, los sistemas de "múltiples bases de datos" y la interoperabilidad se han convertido en temas importantes. Las técnicas para el tratamiento de semánticas incompatibles entre varias bases de datos se examinan en DeMichiel (1989), Siegel y Madnick (1991), Krishnamurthy y otros (1991), y en Wang y Madnick (1989). Castano y otros (1998) presenta un excelente examen de las técnicas para el análisis de esquemas. Pitoura y otros (1995) comenta la orientación a objetos en sistemas de "múltiples base de datos".

El procesamiento de transacciones en multibases de datos aparece en Mehrotra y otros (1992), Georgakopoulos y otros (1991), Elmagarmid y otros (1990) y en Brietbart y otros (1990), entre otros. Elmagarmid y otros (1992) aborda el tema del procesamiento de transacciones para aplicaciones avanzadas, incluyendo las aplicaciones de ingeniería comentadas en Heiler y otros (1992).

Los sistemas de flujo de trabajo (*workflow*), que se han hecho muy populares para administrar la información en organizaciones complejas, usan transacciones multinivel y anidadas en conjunción con bases de datos distribuidas. Weikum (1991) diserta sobre la administración de una transacción multinivel. Alonso y otros (1997) comenta las limitaciones de los actuales sistemas de flujo de trabajo.

Se han implementado varios DBMSs distribuidos experimentales. Entre ellos se incluye el INGRES distribuido de Epstein y otros, (1978), el DDTS de Devor y Weeldreyer, (1980), el SDD-1 de Rothnie y otros, (1980), el System R$*$ de Lindsay y otros, (1984), el SIRIUS-DELTA de Ferrier y Stangret, (1982) y el MULTIBASE de Smith y otros (1981). El sistema OMNIBASE de Rusinkiewicz y otros (1988) y el *Federated Information Base* desarrollado usando el modelo de datos Candide de Navathe y otros (1994) son ejemplos de DDBMSs federados. Pitoura y otros (1995) presenta un estudio comparativo de los prototipos de sistemas de bases de datos federados. La mayoría de desarrolladores de DBMSs comerciales cuentan con productos que usan el planteamiento cliente-servidor y ofrecen versiones distribuidas de sus sistemas. Algunos de los problemas concernientes a las arquitecturas DBMS cliente-servidor son tratados en Carey y otros (1991), DeWitt y otros (1990) y en Wang y Rowe (1991). Khoshafian y otros (1992) comenta los problemas de diseño de los DBMSs relacionales en el entorno cliente-servidor. Los problemas de administración de estos sistemas aparecen tratados en numerosos libros, como en Zantinge y Adriaans (1996).

PARTE 8

# Tecnologías emergentes

CAPÍTULO 26

# Programación de una base de datos web usando PHP

Vamos a prestar ahora atención al modo de utilizar y acceder a las bases de datos desde Internet. Muchas aplicaciones de Internet y de comercio electrónico (*e-commerce*) proporcionan interfaces web para acceder a la información almacenada en una o varias bases de datos, las cuales suelen recibir el nombre de **fuentes de datos**. Es común utilizar arquitecturas cliente/servidor de dos y tres niveles para las aplicaciones Internet (consulte la Sección 2.5) aunque, en otros casos, se usan otras variantes del modelo cliente/servidor. El *e-commerce* y otras aplicaciones de base de datos para Internet están diseñadas para interactuar con el usuario a través de interfaces web que muestran páginas web. El método más corriente de especificar los contenidos y formatear este tipo de páginas es mediante los **documentos de hipertexto**. Existen varios lenguajes para escribir este tipo de documentos, aunque el más habitual es HTML (Lenguaje de marcado de hipertexto, *HyperText Markup Language*). Aunque HTML es ampliamente utilizado para formatear y estructurar *documentos web*, no es adecuado para la especificación de los *datos estructurados* que se extraen de las bases de datos. Por ello, XML (Lenguaje de marcado extensible, *eXtensible Markup Language*) se ha erigido como el estándar para la estructuración y el intercambio de datos a través de la Web. XML puede usarse para ofrecer información acerca de la estructura y el significado de los datos en las páginas web en lugar de sólo especificar cómo están formateadas estas páginas para su visualización en la pantalla. Los aspectos de formateo se especifican de forma separada (por ejemplo, usando un lenguaje de formateo como XSL (Lenguaje de hojas de estilo extensible, *eXtensible Stylesheet Language*).

El lenguaje HTML básico es útil para la generación de páginas web *estáticas* que cuentan con texto y otros objetos fijos. Pero la mayoría de aplicaciones de *e-commerce* precisan de elementos que ofrezcan interactividad con el usuario. Por ejemplo, consideremos el caso de un cliente de una línea aérea que quiere consultar la hora de llegada y la puerta de un vuelo particular. El usuario puede introducir algún tipo de información como la fecha o el número de vuelo en ciertos campos de la página web. El programa que hay detrás debe enviar una consulta a la base de datos de la compañía para recuperar esa información, y después mostrarla. Este tipo de páginas, en las que parte de la información se extrae de bases de datos o de otras fuentes, reciben el nombre de *páginas dinámicas*, porque los datos extraídos y mostrados cada vez serán para diferentes vuelos y fechas.

Existen varias técnicas para programar características dinámicas en las páginas web. Nosotros nos centraremos en una de ellas, que está basada en el uso del lenguaje de *scripting* de código abierto PHP. Este lenguaje ha experimentado un auge enorme en los últimos tiempos. Los intérpretes de PHP son gratuitos y están escritos en lenguaje C, lo que le permite estar presente en la mayoría de plataformas. Un intérprete PHP cuen-

ta con el preprocesador de hipertexto, el cual se encarga de ejecutar los comandos PHP contenidos en un fichero de texto y crear el archivo HTML deseado. Para el acceso a las bases de datos es preciso incluir en este intérprete una librería de funciones PHP, tal y como veremos en la Sección 26.4. Los programas PHP se ejecutan en el servidor web, lo que contrasta con otros lenguajes de *scripting*, como JavaScript, que lo hacen en el cliente.

Este capítulo esta organizado del siguiente modo. La Sección 26.1 muestra en qué se diferencia la información almacenada en las páginas web de la que está almacenada en bases de datos estructuradas, y plantea las diferencias existentes entre información estructurada, semiestructurada y no estructurada. La Sección 26.2 muestra un ejemplo simple que ilustra cómo se usa PHP. La Sección 26.3 ofrece una panorámica general del lenguaje PHP y cómo se usa para programar algunas funciones básicas para páginas web interactivas. La Sección 26.4 se centra en el uso de PHP para interactuar con bases de datos SQL a través de una librería de funciones conocida como PEAR DB. Por último, la Sección 26.5 es un resumen del capítulo. En el Capítulo 27 trataremos el lenguaje XML para la representación y el intercambio de datos en la Web, y comentaremos algunas de las formas en las que puede usarse.

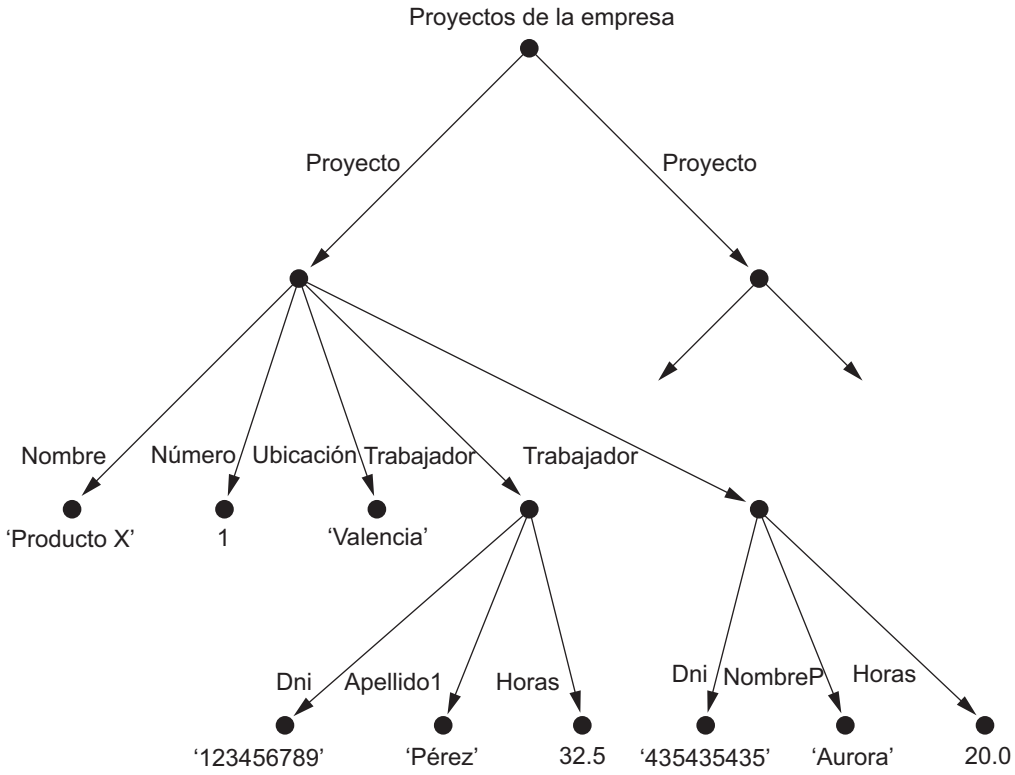
## 26.1 Datos estructurados, semiestructurados y no estructurados

La información contenida en una base de datos se dice que es **estructurada** porque está representada de una forma estricta. Por ejemplo, cada registro de una tabla de una base de datos relacional (como la tabla EMPLEADO de la Figura 5.6) sigue el mismo formato que los otros registros de esa tabla. Para crear el esquema que contenga los datos estructurados es común diseñar la base de datos usando técnicas como las descritas en los Capítulos 3, 4, 7, 10 y 11. Después, el DBMS se encarga de comprobar que todos los datos siguen las estructuras y restricciones especificadas en ese esquema.

Sin embargo, no todos los datos se recopilan e insertan en las cuidadosamente diseñadas bases de datos estructuradas. En algunas aplicaciones, la información se recopila de forma manual antes de saberse cómo será almacenada y administrada. Estos datos pueden tener una cierta estructura, pero no todos ellos seguirán el mismo patrón. Algunos atributos pueden estar compartidos entre las distintas entidades, pero otros pueden existir sólo en algunas de ellas. Por otra parte, es posible que se inserten atributos adicionales en alguno de los elementos nuevos, y recuerde que no contamos con ningún esquema predefinido. Esto es lo que se conoce como **dato semiestructurado**. Se han propuesto varios modelos de datos para representar este tipo de información, basados con frecuencia en el uso de árboles o grafos en lugar de las estructuras planas del modelo relacional.

Una diferencia clave entre un dato estructurado y otro semiestructurado hace referencia al modo de manipular la construcción del esquema (los nombres de los atributos, las relaciones y los tipos de entidades). En un dato semiestructurado, la información del esquema está *mezclada* con los valores de los datos, ya que cada objeto puede tener diferentes atributos que no son conocidos de antemano. Por consiguiente, este tipo de datos suele conocerse a veces como **datos autodescriptivos**. Consideremos el siguiente ejemplo. Queremos recopilar una lista de referencias bibliográficas relacionadas con cierto proyecto de investigación. Algunas de estas referencias pueden ser libros o informes técnicos; otras, artículos divulgativos aparecidos en revistas o conferencias; e, incluso, podemos tener publicaciones o conferencias completas. Claramente, cada una de estas fuentes puede tener atributos y tipos de información diferentes. E, incluso para el mismo tipo de referencia (digamos, extractos de una conferencia), la información podría ser distinta. Por ejemplo, la citación de un artículo podría ser completa si cuenta con el autor, el título, el número de páginas, etc., o incompleta si no facilitase ninguno de estos datos. En el futuro pueden aparecer nuevas fuentes bibliográficas (por ejemplo, referencias a páginas web o a conferencias tuteladas), cada una de ellas con nuevos atributos que las describen.

**Figura 26.1.** Representación de un dato semiestructurado como un grafo.




Un dato semiestructurado puede ser mostrado como un grafo tendente (véase la Figura 26.1). La información mostrada en esta figura se corresponde con alguno de los datos estructurados de la Figura 5.6. Como podemos ver, este modelo se asemeja en parte al modelo de objeto (véase la Figura 20.1) en su capacidad para representar objetos complejos y estructuras anidadas. En la Figura 26.1, las **etiquetas**, o ***tags***, de los bordes tendentes representan los nombres de esquema: los *nombres de los atributos,* los *tipos de objetos (o tipos o clases de entidad)* y las *relaciones*. Los nodos internos representan objetos individuales o atributos compuestos, mientras que los nodos hoja son valores de datos actuales de atributos simples (atómicos).

Existen dos diferencias principales entre el modelo semiestructurado y el de objeto que comentamos en el Capítulo 20:

1. La información de esquema [nombres de atributos, relaciones y clases (tipos objetos)] en el modelo semiestructurado está entremezclada, con los objetos y sus valores conviviendo en la misma estructura de datos.
2. En el modelo semiestructurado, no hay necesidad de que los objetos de datos cumplan un esquema predefinido.

Además de los datos estructurados y semiestructurados, existe una tercera categoría para los **datos no estructurados** porque existe una indicación muy leve acerca del tipo de datos. Un ejemplo típico de esto es un documento de texto que contiene información incrustada en él. Las páginas web escritas en HTML que contienen cierta información están consideradas como un dato no estructurado. Vamos a considerar parte del archivo HTML mostrado en la Figura 26.2. El texto que aparece entre los símbolos menor y mayor, <...>, es una **etiqueta HTML**. Una etiqueta con una barra inclinada, </...>, es una **etiqueta de finalización o cierre** que marca el final del efecto de la **etiqueta de inicio o apertura** coincidente. Las etiquetas **marcan** el docu-

mento[1] para informar al procesador HTML de cómo mostrar el texto que se encuentra entre las mismas etiquetas de inicio y cierre. Por tanto, especifican el formato de la página en lugar del significado de los diferentes elementos de datos que componen el documento. Las etiquetas HTML especifican información, como el tamaño y el estilo de la fuente (negrita, cursiva, etc.), el color, los niveles de encabezamiento en los documentos, etc. Otras permiten estructurar el texto especificando listas numeradas o sin numerar o tablas. Incluso estas etiquetas de estructuración especifican que el dato textual incrustado debe mostrarse de una cierta forma, en lugar de indicar el tipo de datos representado en la tabla.

HTML utiliza un gran número de etiquetas predefinidas, las cuales se utilizan para especificar una gran variedad de comandos para formatear documentos web. Las etiquetas de inicio y cierre especifican el rango del texto que cada comando debe formatear. Los siguientes son algunos ejemplos de las etiquetas mostradas en la Figura 26.2:

- <HTML> ... </HTML> especifican los limites del documento.
- La información de **cabecera del documento** (contenida entre las etiquetas <HEAD> ... </HEAD>) especifica comandos varios que se utilizan a lo largo del mismo. Por ejemplo, puede contener **funciones *script*** escritas en lenguajes como JavaScript, o **estilos de formateo** (fuentes, estilos de párrafo, estilos de cabecera, etc.) que pueden usarse en el documento. También permite indicar un título identificativo del fichero y otra información similar que no será mostrada como parte del documento.
- El **cuerpo** del documento (etiquetas <BODY> ... </BODY>) incluye el texto y las etiquetas que especifican la manera en que éste se formateará y aparecerá en la pantalla. También incluye referencias a otros objetos, como imágenes, mensajes de voz y otros documentos.
- Las etiquetas <H1> ... </H1> especifican que el texto debe aparecer como un nivel 1 de cabecera. Existen varios de estos niveles (<H2>, <H3>, etc.), y cada uno muestra la información un poco más pequeña que el anterior.
- Las etiquetas <TABLE> ... </TABLE> indican que el texto siguiente debe aparecer como una tabla. Cada fila está encerrada entre las etiquetas <TR> ... </TR>, mientras que el texto de cada columna aparece entre <TD> ... </TD>.[2]
- Algunas etiquetas de inicio pueden contener **atributos** para describir propiedades adicionales de la misma.[3]

En la Figura 26.2, la etiqueta de inicio <TABLE> tiene cuatro atributos que describen distintas características de la tabla. Las etiquetas <TD> y <FONT> que le siguen cuentan con uno y dos de estos atributos, respectivamente.

HTML tiene muchas etiquetas predefinidas, y cientos de libros dedicados exclusivamente a describir cómo usarlas. Si se diseña adecuadamente, un documento HTML puede formatearse de modo que las personas puedan entender fácilmente su contenido y ser capaces de navegar por ellos. Sin embargo, los *programas informáticos* tienen gran dificultad para interpretar automáticamente el código fuente HTML porque no incluye información de esquema acerca del tipo de datos contenido en los documentos. A medida que las aplicaciones de Internet y de *e-commerce* se han ido automatizando, es crucial la capacidad de intercambiar documentos web entre distintas ubicaciones e interpretar sus contenidos automáticamente. Esta necesidad fue una de las razones que condujeron al desarrollo de XML, que se comenta en el Capítulo 27.

---

[1] Ésta es la razón por la que se le conoce como Lenguaje de *marcado* (o marcas) de hipertexto.

[2] <TR> marca el inicio de una fila y <TD>, de una columna.

[3] Éste es el modo en que se utiliza el término *atributo* en los lenguajes de marcación de documentos, el cual difiere considerablemente de su acepción en los modelos de bases de datos.

**Figura 26.2.** Parte de un documento HTML representativo de un dato no estructurado.

```
<HTML>
  <HEAD>
  . . .
  </HEAD>
  <BODY>
    <H1>Listado de los proyectos de la compañía y de los empleados que trabajan en ellos</H1>
    <H2>Proyecto ProductoX:</H2>
    <TABLE width="100%" border="0" cellpadding="0" cellspacing="0">
      <TR>
        <TD width="50%"><FONT size="2" face="Arial">José Pérez:</FONT></TD>
        <TD>32,5 horas semanales</TD>
      </TR>
      <TR>
        <TD width="50%"><FONT size="2" face="Arial">Carmen Ojeda:</FONT></TD>
        <TD>20,0 horas semanales</TD>
      </TR>
    </TABLE>
    <H2>Proyecto ProductoY:</H2>
    <TABLE width="100%" border="0" cellpadding="0" cellspacing="0">
      <TR>
        <TD width="50%"><FONT size="2" face="Arial">José Pérez:</FONT></TD>
        <TD>7,5 horas semanales</TD>
      </TR>
      <TR>
        <TD width="50%"><FONT size="2" face="Arial">Carmen Ojeda:</FONT></TD>
        <TD>20,0 horas semanales</TD>
      </TR>
      <TR>
        <TD width= "50%"><FONT size="2" face="Arial">Eduardo Campos:</FONT></TD>
        <TD>10,0 horas semanales</TD>
      </TR>
    </TABLE>
  . . .
  </BODY>
</HTML>
```

El ejemplo de la Figura 26.2 ilustra una página HTML **estática**, ya que toda la información a mostrar está especificada explícitamente dentro del archivo HTML. En muchos casos, parte de esa información debe extraerse de una base de datos. Por ejemplo, los nombres de los proyectos y de los empleados que trabajan en ellos deben obtenerse de la base de datos de la Figura 5.6 a través de la consulta SQL apropiada. Puede que

queramos usar la mismas etiquetas HTML para mostrar los proyectos y los empleados que trabajan en ellos, pero cambiando la información particular de cada uno de ellos. Por ejemplo, puede que queramos mostrar una página web con la información del *ProyectoX* y después la del *ProyectoY*. Aunque ambas páginas usan las mismas etiquetas HTML de formateo, los datos a mostrar son diferentes. Este tipo de páginas reciben el nombre de **dinámicas**, ya que la parte de datos de las mismas puede ser diferente cada vez que se muestra, aun cuando la apariencia sea la misma.

Existen varias técnicas para definir páginas web dinámicas. El resto de este capítulo ofrece una perspectiva de una de estas técnicas que usa el lenguaje de *script* PHP.

# 26.2 Un sencillo ejemplo PHP

PHP es un lenguaje de *scripting* de código abierto y de propósito general. El motor intérprete de PHP está escrito en C, lo que permite utilizarlo en casi todos los tipos de computadores y sistemas operativos. PHP suele venir instalado en UNIX. En equipos con otros sistemas operativos, como Windows, Linux o Mac OS, el intérprete de PHP puede descargarse del URL *http://www.php.net.* Al igual que ocurre con otros lenguajes de *scripting*, PHP es particularmente adecuado para la manipulación de páginas de texto, y en particular para la manipulación de páginas HTML dinámicas en el servidor web. En esto se diferencia de JavaScript, el cual se descarga con la página web para ejecutarse en la máquina del cliente.

PHP cuenta con librerías de funciones para el acceso a bases de datos almacenadas en distintos tipos de esquemas relacionales como Oracle, MySQL, SQLServer o cualquier sistema que soporte el estándar ODBC (consulte el Capítulo 9). Bajo la arquitectura de tres niveles (consulte el Capítulo 2), el DBMS debe residir en la capa inferior del servidor de base de datos. PHP debe funcionar en el nivel medio del servidor web, donde los comandos del programa PHP manipularán los ficheros HTML para crear las páginas web dinámicas personalizadas. A continuación, dichas páginas se envían a la capa cliente para que el usuario las vea e interactúe con ellas.

Consideremos el ejemplo mostrado en la Figura 26.3(a), el cual solicita al usuario que introduzca su nombre y apellido para después mostrarle un mensaje de bienvenida. Los números de línea no forman parte del código del programa; sólo se usan mas adelante para aclarar lo que hacen:

1. Supongamos que el fichero que contiene el *script* PHP en el segmento de programa P1 está almacenado en la siguiente localización Internet: *http://www.miservidor.com/ejemplo/saludo.php*. Por tanto, si un usuario escribe este URL en el navegador, el intérprete PHP inicia la traducción del código y genera el formulario mostrado en la Figura 26.3(b). Explicaremos qué ocurre a medida que avancemos por las líneas de este fragmento de código.
2. La línea 0 muestra la etiqueta de inicio PHP `<?php`, la cual indica al motor del intérprete PHP que debe procesar todas las líneas siguientes hasta encontrar la etiqueta de cierre `?>` (línea 16). El texto que se encuentra tras estas marcas aparece tal cual. Esto permite que los segmentos de código PHP se coloquen dentro de grandes ficheros HTML, ya que sólo las secciones que se encuentren entre `<?php` y `?>` serán tratadas por el preprocesador PHP.
3. La línea 1 muestra una de las formas de introducir comentarios en un programa PHP, para lo cual lo precedemos por //. Los comentarios de una sola línea pueden empezar también por el carácter #, y terminan al final de la línea en la que se encuentre. Para colocar varias líneas de comentarios, utilice `/*` al comienzo del bloque y `*/` al final.
4. La variables autoglobal PHP predefinida `$_POST` (línea 2) es un *array* que contiene todos los valores introducidos a través de los parámetros de un formulario. Los *arrays* en PHP son dinámicos y no tienen un número fijo de elementos. Pueden indexarse de manera numérica, donde sus posiciones están marcadas mediante números (0, 1, 2, ...), o estar definidos como *arrays* asociativos, en los que los índices pueden ser cualquier valor de tipo cadena. Por ejemplo, un array asociativo basado en el color

puede contener los índices {"rojo", "azul", "verde"}. En este ejemplo, `$_POST` está indexado asociativamente por el valor `user_name` que aparece en el atributo `name` de la etiqueta `input` de la línea 10. Por consiguiente, `$_POST['user_name']` contendrá el valor tecleado por el usuario. Trataremos con más detalle los *arrays* PHP en la Sección 26.3.2.

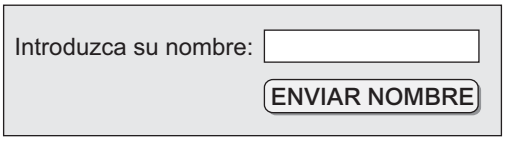
5. Cuando la página web ubicada en http://www.miservidor.com/ejemplo/saludo.php se abre por primera vez, la condición `if` de la línea 2 se avaluará como falsa porque aún no existe ningún valor en `$_POST['user_name']`. Por tanto, el intérprete PHP procesará las líneas entre la 6 y la 15, las cuales crean el texto para un archivo HTML que muestra el formulario de la Figura 26.3(b).
6. La línea 8 muestra una forma de crear cadenas de texto de gran tamaño en un archivo HTML. Veremos otras formas de manipular cadenas más adelante en esta sección. Todo el texto contenido entre `<<<_HTML_` de apertura y `_HTML_;` de cierre aparecen en el archivo HTML tal cual. La etiqueta de cierre debe aparecer sola en una línea separada. Por tanto, el texto añadido al archivo HTML enviado al cliente será el que se encuentra entre las líneas 10 y 13, las cuales incluyen las etiquetas HTML para crear el formulario de la Figura 26.3(b).

**Figura 26.3.** (a) Segmento de programa PHP para introducir un saludo. (b) Formulario inicial visualizado por el segmento de programa PHP. (c) El usuario introduce el nombre *José Pérez*. (d) El formulario imprime el mensaje de bienvenida para *José Pérez*.

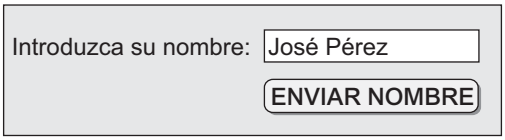
**(a)**

```
//Segmento de programa P1:
0)    <?php
1)    // Imprimir mensaje de bienvenida si el usuario envió su nombre
      // mediante el formulario HTML
2)     if ($_POST['user_name']) {
3)        print("Bienvenido, ") ;
4)        print($_POST['user_name']);
5)    }
6)    else {
7)        // Imprimir el formulario para introducir el nombre del usuario si todavía
          // no se ha introducido el nombre
8)        print <<<_HTML_
9)        <FORM method="post" action="$_SERVER['PHP_SELF']">
10)       Introduzca su nombre: <input type="text" name="user_name">
11)       <BR/>
12)       <INPUT type="submit" value="ENVIAR NOMBRE">
13)       </FORM>
14)       _HTML_;
15)    }
16)    ?>
```

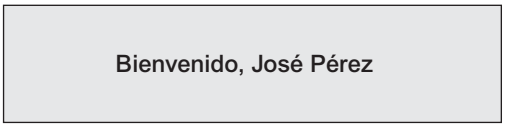
**(b)**




**(c)**




**(d)**



7. Los nombres de variables PHP empiezan con un carácter `$` y pueden incluir letras, números y el signo de subrayado (_). La variable autoglobal PHP predefinida `$_SERVER` (línea 9) es un *array* que incluye información acerca del servidor local. El elementos `$_SERVER['PHP_SELF']` de este *array* es la ruta del fichero PHP que se está ejecutando actualmente en el servidor. De este modo, el atributo `action` de la etiqueta `form` (línea 9) informa al intérprete PHP que debe reprocesar el mismo fichero una vez que el usuario ha rellenado los parámetros del formulario.
8. Una vez que el usuario ha escrito el nombre *José Pérez* en el cuadro de texto y pulsa el botón `ENVIAR NOMBRE` (Figura 26.3[c]), el segmento de programa *P1* es procesado de nuevo. Esta vez, `$_POST['user_name']` incluirá la cadena "José Pérez", lo que hará que se coloquen en el archivo HTML enviado al cliente las líneas 3 y 4, las cuales muestran el mensaje de la Figura 26.3(d).

Como hemos podido ver a lo largo de este ejemplo, el programa PHP puede generar dos tipos distintos de comandos HTML dependiendo de si es la primera vez que se ejecuta o si el usuario ha introducido su nombre a través del formulario. En general, un programa PHP puede crear numerosas variaciones de texto HTML en un fichero del servidor según las rutas condicionales particulares tomadas en el programa. Por tanto, el código HTML enviado al cliente será diferente en base a la interacción con el usuario. Ésta es una de las formas en las que se puede usar PHP para generar páginas web dinámicas.

# 26.3 Visión general de las características básicas de PHP

En esta sección ofrecemos una visión general de unas cuantas características de PHP que son útiles para crear páginas HTML interactivas. La siguiente sección se centra en cómo los programas PHP pueden acceder a las bases de datos para realizar consultas y actualizaciones. No vamos a ofrecer una explicación global de PHP, pues existen libros enteros dedicados a este tema.

En su lugar, nos centraremos en ilustrar determinadas características de PHP que son particularmente adecuadas para crear páginas web dinámicas que contienen comandos de acceso a bases de datos. Aunque esta sección no abarca las funciones PHP de acceso a una base de datos, sí explica algunos de los conceptos y características que serán necesarios cuando en la Sección 26.4 expliquemos el acceso a las bases de datos.

## 26.3.1 Variables, tipos de datos y estructuras de programación de PHP

Los nombres de variable de PHP empiezan con el símbolo $ y pueden incluir caracteres, letras y el carácter de subrayado (_). No están permitidos otros caracteres especiales. Los nombres de variable hacen distinción entre mayúsculas y minúsculas, y el primer carácter no puede ser un número. Las variables no tienen tipo; su tipo queda determinado por los valores que se les asignan. De hecho, la misma variable puede cambiar de tipo en cuanto se le asigna un nuevo valor. La asignación se realiza mediante el operador =.

Como PHP está orientado al procesamiento de texto, dispone de diferentes tipos de valores de cadena. También hay muchas funciones para manipular cadenas. Sólo explicaremos algunas de las propiedades básicas de los valores de cadena y de las variables. La Figura 26.4 ilustra algunos valores de cadena. Hay tres formas principales de expresar las cadenas y el texto:

1. **Cadenas con comillas simples.** La cadena se expresa entre comillas simples, como en las líneas 0, 1 y 2. Si necesitamos una comilla simple dentro de una cadena, tenemos que utilizar el carácter de escape (\) (véase la línea 2).
2. **Cadenas con comillas dobles.** La cadena se expresa entre comillas dobles, como en la línea 7. En este caso, los nombres de variable que aparecen dentro de la cadena son reemplazados por los valores

**Figura 26.4.** Ilustración de las cadenas PHP básicas y de los valores de texto.

```
0)   print 'Bienvenido a mi sitio web.';
1)   print 'Les respondí, "Bienvenidos a casa"';
2)   print 'Ahora visitaremos el \'siguiente\' sitio web';
3)   printf('El coste es $%.2f y los impuestos son $%.2f', $cost, $tax) ;
4)   print strtolower('AbCdE');
5)   print ucwords(strtolower('ENRIQUE campos'));
6)   print 'abc' . 'efg'
7)   print "remita su respuesta a: $email_address"
8)   print <<<FORM_HTML
9)   <FORM method="post" action="$_SERVER['PHP_SELF']">
10)  Introduzca su nombre: <input type="text" name="user_name">
11)  FORM_HTML
```

actualmente almacenados en esas variables. El intérprete identifica los nombres de variable dentro de las cadenas de comillas dobles por su carácter $ inicial y los reemplaza por el valor de la variable. Es lo que se conoce como interpolación de variables dentro de las cadenas. La interpolación no se da en las cadenas con comillas simples.

3. **Documentos "here" (o documentos aquí).** Encierran parte de un documento entre <<<`NOMBREDOC` y una línea que sólo contiene el nombre de documento `NOMBREDOC`. `NOMBREDOC` puede ser cualquier cadena siempre y cuando se utilice tanto para el inicio como para el final del documento "here". En la Figura 26.4 queda ilustrado en las líneas 8 a 11. Las variables también son interpretadas reemplazándolas por sus valores de cadena si aparecen dentro de los documentos "here".

4. **Comillas simples y dobles.** Las comillas simples y dobles que PHP utiliza para encerrar cadenas deben crearse mediante un editor de texto que no produzca comillas de apertura y cierre *tipográficas* (“”) alrededor de la cadena. Deben ser comillas *rectas* ("") en los dos extremos de la cadena.

También existe un operador de concatenación de cadenas que se especifica mediante el símbolo del punto (.), como se ilustra en la línea 6 de la Figura 26.4. Hay muchas funciones de cadena. Sólo ilustraremos un par de ellas. La función `strtolower` cambia los caracteres alfabéticos de la cadena a minúsculas, mientras que la función `ucwords` convierte en mayúsculas todas las palabras de una cadena. En la Figura 26.4 se ilustran en las líneas 4 y 5.

La regla general es utilizar cadenas con comillas sencillas para las cadenas literales que no contienen variables de programa PHP y los otros dos tipos (cadenas con comillas dobles, documentos "here") cuando los valores de las variables tienen que ser interpoladas en la cadena. En los bloques grandes de varias líneas, el programa debe utilizar el estilo de los documentos "here" para las cadenas.

PHP también tiene tipos de datos numéricos para los enteros y los números en coma flotante, y generalmente obedecen las reglas del lenguaje de programación C para procesar estos tipos. Los números se pueden formatear para su impresión como cadenas especificando el número de dígitos que siguen al punto decimal. Una variante de la función `print` denominada `printf` (impresión formateada) permite formatear los números como una cadena tal como se ilustra en la línea 3 de la Figura 26.4.

También existen estructuras estándar para crear bucles `for` y `while`, así como sentencias condicionales `if`. Por regla general, son parecidas a las del lenguaje C. No las explicaremos aquí. De forma parecida, cualquier valor se evalúa como verdadero si se utiliza como una expresión booleana, excepto para el cero numérico (0) y la cadena vacía, que se evalúan como falsos. También hay valores verdaderos y falsos literales que pueden asignarse. Los operadores de comparación también obedecen generalmente las reglas de C: son == (igual), != (distinto), > (mayor que), >= (mayor que o igual), < (menor que) y <= (menor que o igual).

### 26.3.2 Arrays PHP

Los arrays son muy importantes en PHP, ya que permiten listas de elementos. Se utilizan con frecuencia en los formularios que emplean menús desplegables. Un array unidimensional se utiliza para almacenar la lista de opciones de un menú desplegable. En el caso de las consultas de bases de datos, los arrays bidimensionales se utilizan de forma que la primera dimensión representa las filas de una tabla y la segunda dimensión, los atributos de la fila. Existen dos tipos principales de arrays: numéricos y asociativos. Explicaremos cada uno de ellos en el contexto de los arrays unidimensionales.

Un **array numérico** asocia un índice numérico (o posición o número de secuencia) con cada elemento del array. Los índices son números enteros que empiezan por cero y aumentan incrementalmete. A un elemento se accede proporcionando su índice. Un **array asociativo** proporciona pares de elementos (clave => valor). Al valor de un elemento se accede proporcionando su clave, y todos los valores clave de un array en particular deben ser únicos. Los valores de los elementos pueden ser cadenas o enteros, aunque también pueden ser arrays, lo que lleva a arrays de más dimensiones.

La Figura 26.5 ofrece dos ejemplos de variables array: `$teaching` y `$courses`. El primer array, `$teaching`, es asociativo (línea 0 de la Figura 26.5), y cada elemento asocia un nombre de curso (como clave) con el nombre del profesor del curso (como valor). En este array hay tres elementos. La línea 1 muestra la actualización del array. El primer comando de la línea 2 asigna un nuevo profesor al curso 'Graphics' actualizando su valor. Como el valor clave ya existe, no se crea un elemento nuevo, pero se actualiza el valor existente. El segundo comando crea un elemento nuevo porque el valor clave 'Data Mining' no existía antes en el array. Los elementos nuevos se añaden al final del array.

Si sólo proporcionamos valores (no claves) como elementos del array, las claves se convierten automáticamente a numéricas y se numeran como 0, 1, 2,.... (es lo que ocurre con el array `$courses` en la línea 5 de la Figura 26.5). Ninguno de los dos tipos de array tiene limitado su tamaño. Si algún valor de otro tipo de dato, por ejemplo un entero, se asigna a una variable PHP que almacenaba un array, la variable pasará a almacenar el valor entero y se perderá el contenido del array. Básicamente, la mayoría de las variables pueden asignarse a valores de cualquier tipo de datos en cualquier momento.

En PHP hay varias técnicas diferentes para recorrer los arrays. En la Figura 26.5 ilustramos dos de estas técnicas. Las líneas 3 y 4 muestran un método de bucle para recorrer todos los elementos de un array utilizando la estructura *foreach*, a fin de imprimir la clave y el valor de cada elemento en una línea separada. Las líneas 7 a 10 muestran cómo se puede utilizar un bucle `for` tradicional. La función integrada `count` (línea 7) devuelve la cantidad actual de elementos del array, que se asigna a la variable `$num` y se utiliza para controlar la finalización del bucle.

El ejemplo de las líneas 7 a 10 también ilustra la visualización de una tabla HTML con colores de fila alternativos, que se consigue especificando los dos colores en un array `$alt_row_color` (línea 8). Cada iteración del bucle, la función del resto, `$i` % 2, cambia de una fila (índice 0) a la siguiente (índice 1) (consulte la línea 8). El color se asigna al atributo HTML `bgcolor` de la etiqueta <TR> (fila de tabla).

**Figura 26.5.** Ilustración del procesamiento básico de arrays en PHP.

```
0)  $teaching = array('Database' => 'Smith', 'OS' => 'Carrick','Graphics' => 'Kam');
1)  $teaching['Graphics'] = 'Benson'; $teaching['Data Mining'] = 'Kam';
2)  sort($teaching);
3)  foreach ($teaching as $key => $value) {
4)     print " $key : $value\n";}
5)  $courses = array('Database', 'OS', 'Graphics', 'Data Mining');
6)  $alt_row_color = array('blue', 'yellow');
7)  for ($i = 0, $num = count($courses); i < $num; $i++) {
8)     print '<TR bgcolor="' . $alt_row_color[$i % 2] . '">';
9)     print "<TD>Course $i is</TD><TD>$course[$i]</TD></TR>\n";
10) }
```

La función `count` (línea 7) devuelve la cantidad actual de elementos del array. La función `sort` (línea 2) ordena el array basándose en los valores de elemento que contiene (no en las claves). En el caso de los arrays asociativos, cada clave permanece asociada con el mismo valor de elemento después de la ordenación. Esto no ocurre cuando se ordenan arrays numéricos. Hay muchas otras funciones que pueden aplicarse a los arrays PHP, pero una explicación completa queda fuera del objetivo de nuestra presentación.

### 26.3.3 Funciones en PHP

Como ocurre en otros lenguajes de programación, en PHP pueden definirse funciones para estructurar mejor un programa complejo y para compartir secciones comunes de código que pueden reutilizar otras aplicaciones. La versión más moderna de PHP, PHP5, también tiene características de orientación a objetos, pero no las explicaremos aquí porque nos estamos centrando en los fundamentos de PHP. Las funciones PHP básicas pueden tener argumentos que se pasan por valor. Es posible acceder a las variables globales dentro de las funciones. A las variables que aparecen dentro de una función y dentro del código que llama a la función, se les aplican las reglas de ámbito estándar.

Vamos a ver dos sencillos ejemplos para ilustrar las funciones básicas de PHP. La Figura 26.6 muestra cómo puede reescribirse el segmento de código P1 de la Figura 26.3(a) utilizando funciones. El segmento de código P1' de la Figura 26.6 tiene dos funciones: `display_welcome()` (líneas 0 a 3) y `display_empty_form()` (líneas 5 a 13). Ninguna de estas funciones tiene argumentos, ni devuelven valores. Las líneas 14 a 19 muestran cómo se efectúa la llamada a estas funciones para producir el mismo efecto que el segmento de código P1 de la Figura 26.3(a). Como podemos ver en este ejemplo, las funciones pueden utilizarse para estructurar mejor el código PHP y facilitar su seguimiento.

En la Figura 26.7 dispone de un segundo ejemplo. Aquí utilizamos el array `$teaching` introducido en la Figura 26.5. La función `course_instructor`() de las líneas 0 a 8 de la Figura 26.7 tiene dos argumentos: `$course` (una cadena que almacena el nombre de un curso) y `$teaching_assignments` (un array asociativo que almacena las asignaciones de cursos, parecido al array `$teaching` de la Figura 26.5). La función localiza el nombre del profesor que imparte un curso en particular. Las líneas 9 a 14 de la Figura 26.7 muestra el uso de esta función.

**Figura 26.6.** Segmento de programa P1 reescrito como P1' utilizando funciones.

```
//Segmento de programa P1':
0)  function display_welcome() {
1)      print("Bienvenido, ") ;
2)      print($_POST['user_name']);
3)  }
4)
5)  function display_empty_form(); {
6)  print <<<_HTML_
7)  <FORM method="post" action="$_SERVER['PHP_SELF']">
8)  Introduzca su nombre: <INPUT type="text" name="user_name">
9)  <BR/>
10)  <INPUT type="submit" value="Enviar nombre">
11)  </FORM>
12)  _HTML_;
13)  }
14)  if ($_POST['user_name']) {
15)      display_welcome();
16)  }
17)  else {
18)      display_empty_form();
19)  }
```

**Figura 26.7.** Ilustración de una función con argumentos y devolución de un valor.

```
0)  function course_instructor ($course, $teaching_assignments) {
1)     if (array_key_exists($course, $teaching_assignments)) {
2)        $instructor = $teaching_assignments[$course];
3)        RETURN "$instructor imparte el curso de $course";
4)     }
5)     else {
6)        RETURN "no existe el curso $course";
7)     }
8)  }
9)  $teaching = array('Database' => 'Smith', 'OS' => 'Carrick','Graphics' => 'Kam');
10) $teaching['Graphics'] = 'Benson'; $teaching['Data Mining'] = 'Kam';
11) $x = course_instructor('Database', $teaching);
12) print($x);
13) $x = course_instructor('Computer Architecture', $teaching);
14) print($x);
```

La llamada de función de la línea 11 debe devolver la cadena: *Smith imparte el curso de Database*, porque la entrada de array con la clave 'Database' tiene el valor 'Smith' como profesor. Por el contrario, la llamada de función de la línea 13 debe devolver la cadena: *no existe el curso Computer Architecture* porque no existe una entrada de array con la clave 'Computer Architecture'. Hagamos unos cuantos comentarios sobre este ejemplo y sobre las funciones de PHP en general:

1. La función de array `array_key_exists($k, $a)` de PHP devuelve verdadero si el valor de la variable `$k` existe como clave en el array asociativo de la variable `$a`. En nuestro ejemplo, comprueba si el valor `$course` proporcionado existe como una clave en el array `$teaching_assignments` (línea 1 de la Figura 26.7).
2. Los argumentos de una función se pasan por valor. Por tanto, en este ejemplo, las llamadas de las líneas 11 y 13 no pueden cambiar el array `$teaching` proporcionado como argumento para la llamada. Los valores proporcionados en los argumentos se pasan (copian) a los argumentos de la función cuando se llama a ésta.
3. Los valores que devuelve una función se colocan después de la palabra clave `RETURN`. Una función puede devolver cualquier tipo. En este ejemplo, devuelve un tipo cadena. En nuestro ejemplo se pueden devolver dos cadenas diferentes, dependiendo de si el valor clave `$course` proporcionado existe o no en el array.
4. Las reglas de ámbito para los nombres de variable se aplican como en otros lenguajes de programación. Las variables globales que se utilizan fuera de la función no se pueden utilizar, a menos que se haga referencia a ellas utilizando el array PHP `$GLOBALS` integrado. Básicamente, `$GLOBALS['abc']` accederá al valor de una variable global `$abc` definida fuera de la función. En caso contrario, las variables que aparecen dentro de una función son locales aun en el caso de que exista una variable global con el mismo nombre.

La explicación anterior ofrece una panorámica breve de las funciones de PHP. No hemos ofrecido muchos detalles porque no es nuestro objetivo presentar PHP en profundidad.

### 26.3.4 Variables de servidor y formularios en PHP

En la variable de array autoglobal `$_SERVER` integrada de PHP hay varias entradas que ofrecen al programador información útil sobre el servidor donde se está ejecutando el intérprete PHP, además de otra información. Estos detalles pueden ser necesarios a la hora de generar el texto de un documento HTML (por ejemplo, consulte la línea 7 de la Figura 26.6). Aquí tiene algunas de las entradas que mencionamos:

1. **$_SERVER['SERVER_NAME']**. Proporciona el nombre de sitio web del servidor donde se está ejecutando el intérprete PHP. Por ejemplo, si el intérprete PHP se está ejecutando en el sitio web *http://www.uta.edu*, entonces esta cadena será el valor de $_SERVER['SERVER_NAME'].
2. **$_SERVER['REMOTE_ADDRESS']**. Es la dirección IP (Protocolo de Internet) del computador del usuario cliente que está accediendo al servidor; por ejemplo, 129.107.61.8.
3. **$_SERVER['REMOTE_HOST']**. Es el nombre de sitio web del computador del usuario cliente; por ejemplo, abc.uta.edu. En este caso, el servidor tendrá que convertir el nombre en una dirección IP para acceder al cliente.
4. **$_SERVER['PATH_INFO']**. Es la parte de una dirección URL que va después de la barra inclinada (/) al final del URL.
5. **$_SERVER['QUERY_STRING']**. Proporciona la cadena que almacena los parámetros de un URL después de un interrogante (?) al final del URL. Por ejemplo, puede albergar parámetros de búsqueda.
6. **$_SERVER['DOCUMENT_ROOT'].** Es el directorio raíz que almacena los archivos en el servidor web que son accesibles para los usuarios cliente.

Éstas y otras entradas del array $_SERVER son las que normalmente se necesitan cuando se crea el archivo HTML que se envía para la visualización.

Otra variable de array autoglobal integrada de PHP es $_POST, que proporciona al programador los valores de entrada enviados por el usuario a través de formularios HTML especificados en la etiqueta <input> de HTML y otras etiquetas similares. Por ejemplo, en la línea 14 de la Figura 26.6, la variable $_POST['user_name'] proporciona al programador el valor escrito por el usuario en el formulario HTML especificado a través de la etiqueta <input> de la línea 8. Las claves para este array son los nombres de los distintos parámetros de entrada proporcionados a través del formulario, por ejemplo utilizando el atributo name de la etiqueta <input> de HTML como en la línea 8. Cuando los usuarios introducen datos a través de formularios, los valores de datos pueden almacenarse en este array.

## 26.4 Visión general de la programación de bases de datos PHP

Hay varias técnicas para acceder a las bases de datos con un lenguaje de programación. En el Capítulo 9 explicamos algunas de ellas, cuando hablamos de cómo acceder a una base de datos SQL utilizando los lenguajes de programación C y Java. En particular, explicamos SQL incrustado, JDBC, SQL/CLI (parecido a ODBC) y SQLJ. En esta sección ofrecemos una visión general de cómo acceder a la base de datos utilizando el lenguaje de *scripting* PHP, que resulta muy adecuado para crear interfaces web para buscar en las bases de datos y actualizarlas, así como páginas web dinámicas.

Hay una librería de funciones de acceso a bases de datos PHP que forma parte de PHP Extension and Application Repository (PEAR), que es una colección de varias librerías de funciones destinadas a mejorar PHP. La librería PEAR DB proporciona funciones para el acceso a las bases de datos. Con esta librería es posible acceder a muchos sistemas de bases de datos, incluyendo Oracle, MySQL, SQLite y Microsoft SQL Server, entre otros.

Explicaremos varias funciones que forman parte de PEAR DB en el contexto de algunos ejemplos. La Sección 26.4.1 muestra cómo conectar con una base de datos utilizando PHP. La Sección 26.4.2 explica cómo se pueden utilizar los datos recopilados por los formularios HTML para insertar un registro nuevo en una tabla. La Sección 26.3 muestra cómo se pueden ejecutar las consultas de recuperación y que se visualicen sus resultados en una página web dinámica.

### 26.4.1 Conexión a una base de datos

Para utilizar las funciones de bases de datos en un programa PHP, debe cargarse el módulo de librería PEAR DB denominado DB.php. En la Figura 26.8 se realiza la carga en la línea 0. A partir de este momento se puede acceder a las funciones de librería DB utilizando `DB::<function_name>`. La función para conectar a una base de datos es `DB::connect('cadena')` donde el argumento `cadena` especifica la información de la base de datos. El formato de 'cadena' es:

```
<DBMS software>://<cuenta usuario>:<contraseña>@<servidor de base de datos>
```

En la Figura 26.8, la línea 1 conecta a la base de datos que está almacenada utilizando Oracle (se especifica mediante la cadena oci8). Algunos de los paquetes de software DBMS que son accesibles a través de PEAR DB son los siguientes:

1. **MySQL.** Se especifica como `mysql` para las versiones antiguas y como `mysqli` para las versiones más recientes, empezando por la versión 4.1.2.
2. **Oracle.** Se especifica como `oc8i` para las versiones 7, 8 y 9. Se utiliza en la línea 1 de la Figura 26.8.
3. **SQLite.** Se especifica como `sqlite`.
4. **Microsoft SQL Server.** Se especifica como `mssql`.
5. **Mini SQL.** Se especifica como `msql`.
6. **Informix.** Se especifica como `ifx`.
7. **Sybase.** Se especifica como `sybase`.
8. **Cualquier sistema compatible con ODBC.** Se especifica como `odbc`.

La lista anterior no es una lista global.

A continuación de `<DB software>` en el argumento de cadena pasado a `DB::connect` está el operador `://` seguido por el nombre de la cuenta de usuario `<cuenta usuario>`; a continuación va el separador `:` y la contraseña de la cuenta, `<contraseña>`. Todo esto va seguido por el operador `@` y el nombre del servidor y el directorio `<servidor de base de datos>` donde está almacenada la base de datos.

**Figura 26.8.** Conexión a una base de datos, creación de una tabla e inserción de un registro.

```
0)  require 'DB.php';
1)  $d = DB::connect('oci8://acct1:pass12@www.host.com/db1');
2)  if (DB::isError($d)) { die("imposible conectar – " . $d->getMessage());}
    ...
3)  $q = $d->query("CREATE TABLE EMPLEADO
4)     (Emp_id INT,
5)     Nombre VARCHAR(15),
6)     Puesto VARCHAR(10),
7)     Dno INT)" );
8)  if (DB::isError($q)) { die("error al crear la tabla – " .$q->getMessage()); }
    ...
9)  $d->setErrorHandling(PEAR_ERROR_DIE);
    ...
10) $eid = $d->nextID('EMPLEADO');
11) $q = $d->query("INSERT INTO EMPLEADO VALUES
12)    ($eid, $_POST['nombre_emp'], $_POST['puesto_emp'], $_POST['dno_emp'])" );
    ...
13) $eid = $d->nextID('EMPLEADO');
14) $q = $d->query('INSERT INTO EMPLEADO VALUES (?, ?, ?, ?)',
15) array($eid, $_POST['nombre_emp'], $_POST['puesto_emp'], $_POST['dno_emp']) );
```

En la línea 1 de la Figura 26.8, el usuario está conectando con el servidor que se ubica en www.host.com/db1 utilizando el nombre de cuenta `acct1` y la contraseña `pass12` almacenados bajo el DBMS de Oracle `oci8`. La cadena entera se pasa utilizando `DB::connect`. La información de conexión se guarda en la variable `$d` de conexión de base de datos, que se utiliza siempre que se aplique una operación a esta base de datos en particular.

La línea 2 de la Figura 26.8 muestra cómo puede comprobarse si se ha establecido satisfactoriamente o no la conexión con la base de datos. PEAR DB tiene una función `DB::isError`, que puede determinar si cualquier operación de acceso a la base de datos ha tenido o no éxito. El argumento para esta función es la variable de conexión a la base de datos (`$d` en este ejemplo). En general, el programador PHP puede comprobar tras cada llamada a la base de datos si la última operación sobre la base de datos fue o no satisfactoria, y terminar el programa (utilizando la función `die`) si se da el último caso. Desde la base de datos también se devuelve un mensaje de error a través de la operación `$d->get_message()`. Este mensaje también puede visualizarse como en la línea 2 de la Figura 26.8.

En general, la mayoría de los comandos SQL se pueden enviar a la base de datos una vez establecida la conexión. La función `$d->query` toma un comando SQL como su argumento de cadena y lo envía al servidor de bases de datos para su ejecución. En la Figura 26.8, las líneas 3 a 7 envían un comando `CREATE TABLE` para crear una tabla denominada `EMPLEADO` con cuatro atributos. Siempre que se ejecuta una consulta, el resultado de ésta es asignado a una variable de consulta, `$q` en nuestro ejemplo. La línea 8 comprueba si la consulta se ejecutó satisfactoriamente o no.

La librería PHP PEAR DB ofrece una alternativa para comprobar los errores después de cada comando de base de datos. La función:

```
$d->setErrorHandling(PEAR_ERROR_DIE)
```

terminará el programa e imprimirá los mensajes de error predeterminados si se produce cualquier error subsiguiente cuando se accede a la base de datos a través de la conexión `$d` (consulte la línea 9 de la Figura 26.8).

## 26.4.2 Recopilación de los datos de formularios e inserción de registros

En las aplicaciones de bases de datos es frecuente recopilar información a través de HTML u otros tipos de formularios web. Por ejemplo, cuando se adquiere un billete de una compañía aérea o se realiza una compra con una tarjeta de crédito, el usuario tiene que introducir información personal, como el nombre, la dirección y el número de teléfono. Esta información se recopila y almacena normalmente en un registro de base de datos en un servidor de bases de datos.

Las líneas 10 a 12 de la Figura 26.8 ilustran cómo puede hacerse esto. En este ejemplo, omitimos el código para crear el formulario y recopilar los datos, que puede ser una variante del ejemplo de la Figura 26.3. Asumimos que el usuario introdujo valores correctos en los parámetros de entrada denominados `nombre_emp`, `puesto_emp` y `dno_emp`. Éstos serían accesibles a través del array `$_POST` autoglobal de PHP, como explicamos al final de la Sección 26.3.4.

En el comando `INSERT` de SQL que aparece en las líneas 11 y 12 de la Figura 26.8, las entradas de array `$POST['nombre_emp']`, `$POST['puesto_emp']` y `$POST['dno_emp']` almacenarán los valores recopilados del usuario a través del formulario de entrada de HTML. Después se insertan como un registro nuevo de empleado en la tabla de empleados.

Este ejemplo también ilustra otra característica de PEAR DB. Es frecuente en algunas aplicaciones crear un identificador de registro único para cada registro nuevo insertado en la base de datos.[4]

[4] Esto sería parecido al OID generado por el sistema explicado en los Capítulos 20 a 22 para los sistemas de bases de datos de objetos y de objetos relacionales.

PHP tiene una función `$d->nextID` para crear una secuencia de valores únicos para una tabla en particular. En nuestro ejemplo, creamos el campo `Emp_id` de la tabla de empleados (consulte la línea 4 de la Figura 26.8) con este objetivo. La línea 10 muestra cómo recuperar el siguiente valor único de la secuencia para la tabla `EMPLEADO`, y las líneas 11 y 12 muestran cómo insertarlo como parte del registro nuevo.

El código de inserción de las líneas 10 a 12 de la Figura 26.8 puede permitir la introducción de cadenas malintencionadas que pueden alterar el comando `INSERT`. Una forma más segura de realizar las inserciones y otras consultas es mediante el uso de marcadores (especificados por el símbolo `?`). En las líneas 13 a 15 se ilustra un ejemplo, donde se inserta otro registro. En esta forma de la función `$d->query()`, hay dos argumentos. El primero es la sentencia SQL, con uno o más símbolos `?` (marcadores). El segundo argumento es un array, cuyos elementos de valor se utilizarán para reemplazar los marcadores en el orden en que se especifican.

### 26.4.3 Consultas de recuperación de datos de las tablas de una base de datos

En la Figura 26.9 hay tres ejemplos de consultas de recuperación mediante PHP. Las líneas 0 a 3 establecen una variable `$d` de conexión con la base de datos y establecen que la manipulación de errores sea la predeterminada, como explicamos en la sección anterior. La primera consulta (líneas 4 a 7) recupera el nombre y el número de departamento de todos los registros de empleado. La variable de consulta `$q` se utiliza para referirnos al resultado de la consulta. En las líneas 5 a 7, un bucle `while` permite recorrer las filas del resultado. La función `$q->fetchRow()` de la línea 5 sirve para recuperar el siguiente registro del resultado de la consulta y para controlar el bucle, que empieza en el primer registro.

El segundo ejemplo de consulta aparece en las líneas 8 a 13 e ilustra una consulta dinámica, donde las condiciones para seleccionar las filas están basadas en los valores introducidos por el usuario. Se trata de recuperar los nombres de los empleados que desempeñan un trabajo determinado y trabajan para un departamento en particular. El trabajo concreto y el número de departamento se introducen mediante un formulario en las variables `$POST['puesto_emp']` y `$POST['dno_emp']`. Si el usuario ha introducido 'Ingeniero' como puesto de

**Figura 26.9.** Ilustración de consultas de recuperación de base de datos.

```
0)  require 'DB.php';
1)  $d = DB::connect('oci8://acct1:pass12@www.host.com/dbname');
2)  if (DB::isError($d)) { die("imposible conectar – " . $d->getMessage()); }
3)  $d->setErrorHandling(PEAR_ERROR_DIE);
...
4)  $q = $d->query('SELECT Nombre, Dno FROM EMPLEADO');
5)  while ($r = $q->fetchRow()) {
6)     print "el empleado $r[0] trabaja para el departamento $r[1] \n" ;
7)  }
...
8)  $q = $d->query('SELECT Nombre FROM EMPLEADO WHERE Puesto = ? AND Dno = ?',
9)     array($_POST['puesto_emp'], $_POST['dno_emp']) );
10) print "empleados del dpto $_POST['dno_emp'] cuyo trabajo es $_POST['puesto_emp']: \n"
11) while ($r = $q->fetchRow()) {
12) print "empleado $r[0] \n" ;
13) }
...
14) $allresult = $d->getAll('SELECT Nombre, Puesto, Dno FROM EMPLEADO');
15) foreach ($allresult as $r) {
16)    print "el empleado $r[0] tiene el puesto $r[1] y trabaja para el dpto $r[2] \n" ;
17) }
...
```

trabajo y 5 como número de departamento, la consulta seleccionaría los nombres de todos los ingenieros que trabajaron en el departamento 5. Como podemos ver, se trata de una consulta dinámica cuyo resultado difiere en función de las opciones que el usuario introduce como entrada. En este ejemplo utilizamos dos marcadores `?`, como explicamos al final de la Sección 26.4.2.

La última consulta (líneas 14 a 17) muestra otra forma de especificar una consulta y de recorrer sus filas. En este ejemplo, la función `$d=>getAll` almacena todos los registros del resultado de una consulta en una sola variable, `$allresult`. Para recorrer todos los registros individuales, podemos utilizar un bucle `foreach`, con la variable de fila `$r` iterando por cada fila de `$allresult`.[5]

# 26.5 Resumen

En este capítulo empezamos en la Sección 26.1 hablando de algunas de las diferencias entre los distintos tipos de datos, clasificándolos en tres tipos principales: estructurados, semiestructurados y no estructurados. Los datos estructurados están almacenados en bases de datos tradicionales.

Los datos no estructurados se refieren a la información visualizada en la Web, especificada a través de HTML, en la que ha desaparecido la información sobre los tipos de los elementos de los datos. Los datos semiestructurados mezclan nombres de tipos de datos y valores de datos, pero no todos los datos tienen que seguir una estructura predefinida fija. En el Capítulo 27 explicaremos XML, que es un ejemplo de modelo de datos semiestructurado.

A continuación, explicamos cómo podemos convertir algunos datos estructurados de una base de datos, en elementos que se puedan introducir y visualizar en una página web. Nos centramos en el lenguaje de *scripting* PHP, que se está haciendo muy popular en la programación de bases de datos web. La Sección 26.2 presentó algunos de los principios básicos de PHP en la programación web, utilizando para ello un ejemplo básico. La Sección 26.3 ofreció algunos de los principios del lenguaje PHP, incluyendo sus tipos de datos de array y cadena, que se utilizan mucho. La Sección 26.4 presentó una visión general de cómo puede utilizarse PHP para especificar distintos tipos de comandos de bases de datos, incluyendo los que permiten crear tablas, insertar registros nuevos y recuperar registros de una base de datos. PHP se ejecuta en el servidor, en comparación con otros lenguajes de *scripting* que se ejecutan en el computador cliente.

Sólo ofrecimos una introducción básica de PHP. Hay muchos libros y sitios web, tanto a nivel básico como avanzado, dedicados a la programación en PHP. También existen muchas librerías de funciones para PHP, como producto de código fuente que es.

## Preguntas de repaso

**26.1** ¿Cuáles son las diferencias entre datos estructurados, semiestructurados y no estructurados? Ofrezca algunos ejemplos de los modelos de datos para cada tipo de datos.

**26.2** ¿Qué tipo de lenguaje de programación es PHP?

**26.3** Explique las distintas formas de especificar cadenas en PHP.

**26.4** Explique los diferentes tipos de arrays en PHP.

**26.5** ¿Qué son las variables autoglobales de PHP? Ofrezca algunos ejemplos de arrays autoglobales de PHP, y explique cómo se utiliza normalmente cada uno de ellos.

**26.6** ¿Qué es PEAR? ¿Qué es PEAR DB?

**26.7** Explique las principales funciones para acceder a una base de datos en PEAR DB, y cómo se utiliza cada una de ellas.

[5] La variable $r es parecida a los cursores y las variables iteradoras explicados en los Capítulos 9, 20 y 21.

**26.8** Explique las diferentes formas de recorrer el resultado de una consulta en PHP.

**26.9** ¿Qué son los marcadores? ¿Cómo se utilizan en la programación de bases de datos PHP?

## Ejercicios

**26.10** A partir del esquema de la base de datos BIBLIOTECA de la Figura 6.14, escriba el código PHP que permita crear las tablas de dicho esquema.

**26.11** Escriba un programa PHP que cree formularios web para introducir la información relativa a una nueva entidad PRESTATARIO. Repítalo con una nueva entidad LIBRO.

**26.12** Escriba interfaces web PHP para las consultas especificadas en el Ejercicio 6.18.

## Ejercicios de práctica

**26.13.** Consideremos la base de datos CONFERENCIA_ENTREVISTA del Ejercicio 3.34 para la que creamos y rellenamos la base de datos relacional en el Ejercicio 8.28 utilizando MySQL. Hay tres tipos de usuarios del sistema: *autor*, *revisor* y *administrador*. El autor debe firmar y enviar un documento a la conferencia; el revisor debe enviar las revisiones de documentos que se le han asignado; y el administrador debe asignar revisores a los documentos (al menos tres revisores a cada documento), así como imprimir un informe de todos los documentos ordenados por la valoración media asignada por los revisores. Los revisores y los administradores tienen asignados un id de usuario y una contraseña; no obstante, los autores deben registrarse antes de poder utilizar el sistema. Escriba una aplicación basada en PHP que implemente las siguientes funciones:

- **a.** Una página de registro de usuario que permita a un autor registrar una dirección de correo electrónico y demás información. El autor elige una contraseña en la página de registro.
- **b.** Una página de inicio de sesión de usuario con cuadros de texto propios de una GUI para aceptar los ids de usuario y las contraseñas, y botones de radio para elegir entre autor, revisor y administrador. Después de la autenticación debe visualizarse un menú separado por cada usuario.
- **c.** El autor debe poder introducir detalles del envío de un documento y de la carga de un archivo.
- **d.** El revisor debe poder elegir un documento asignado y enviar una revisión.
- **e.** El administrador debe poder asignar al menos tres revisores a cada documento. También debe poder visualizar un informe de todos los documentos ordenado por la valoración media (de la más alta a la más baja).

**26.14.** Consideremos la base de datos LIBRO_CALIF del Ejercicio 4.28 para la que creamos y rellenamos una base de datos relacional en el Ejercicio 8.29 utilizando Oracle y MySQL.

- **a.** Escriba y pruebe un programa Java que interactúe con la base de datos Oracle e imprima un informe de calificaciones para cada curso. En el informe deben aparecer el número de calificaciones A, B, C, D y F obtenidas en cada uno de los cursos. El informe aparecerá ordenado por el número de curso y su formato será el siguiente:

```
Informe de calificaciones
N. curso    Curso        A    B      C      D      F
_____________________________________________________
CSc 2010    Intro a CS   12   8      15     5      2
CSc 2310    Java         15   2      12     8      1
...
...
_____________________________________________________
```

**b.** Escriba y pruebe *scripts* PHP que accedan a la base de datos MySQL e implementen las siguientes funciones para una sección de curso dada. La información relativa al número de curso, periodo y nivel se proporcionará como parámetros `GET` en el propio URL.

i El profesor debe poder introducir las puntuaciones/notas de un determinado módulo a evaluar. El profesor también debe poder elegir un módulo a evaluar mediante una lista desplegable.

ii El profesor debe poder ver las puntuaciones de todos los módulos de todos los estudiantes en un formato tabulado, con una fila por estudiante y una columna por módulo evaluado.

**26.15.** Consideremos la base de datos `SUBASTA_ONLINE` del Ejercicio 4.29; posteriormente, en el Ejercicio 8.30 utilizamos MySQL para crear y rellenar la base de datos relacional. Escriba *scripts* en PHP para implementar un sitio web de subastas *online*. Un miembro se puede registrar y elegir una contraseña, y después iniciar sesión en el sistema. Debe implementar las siguientes funciones:

**a.** Un comprador debe ser capaz de buscar elementos introduciendo una palabra clave. El resultado de la búsqueda debe ser una lista de elementos, cada uno de ellos con un hipervínculo. Al hacer clic en este último, debe visualizarse una página detallada describiendo el elemento. El usuario puede pujar por el elemento en esta página.

**b.** Un vendedor debe ser capaz de colocar un elemento a la venta proporcionando los detalles del mismo, la fecha límite para realizar pujas, etcétera.

**c.** Compradores y vendedores deben poder ver una lista de todos los elementos para los que se ha terminado la subasta.

**d.** Compradores y vendedores deben poder colocar una tasación y ver los detalles de las transacciones en las que están involucrados.

## Bibliografía seleccionada

Muchas son las fuentes dedicadas a la programación en PHP, tanto impresas como *online* en la Web. Mencionamos un par de libros a modo de ejemplo. Una buena introducción a PHP es la obra Sklar (2004). Si lo que desea es el desarrollo avanzado de sitios web, el libro Schlossnagle (2004) ofrece ejemplos muy detallados.

CAPÍTULO 27

# XML: Lenguaje de marcado extensible

El lenguaje XML se ha erigido como el estándar para estructurar e intercambiar datos por la Web. XML se puede utilizar para proporcionar información sobre la estructura y el significado de ciertos componentes de los datos visualizados en una página web, en lugar de especificar cómo ha de visualizarse la información, que es lo que hace HTML. El formateo de la visualización se puede especificar por separado, por ejemplo mediante XSL (Lenguaje de hojas de estilo extensible, *Extensible Stylesheet Language*). Recientemente, también se ha propuesto a XML como un posible modelo para el almacenamiento y la recuperación de datos, aunque hasta el momento sólo se han desarrollado unos pocos sistemas de bases de datos experimentales basados en XML.

En este capítulo nos centraremos en describir el modelo de datos de XML, y en cómo pueden formatearse los datos extraídos de las bases de datos relacionales como documentos XML que luego puede intercambiarse por la Web. La Sección 27.1 presenta el modelo de datos de XML, que está basado en estructuras en árbol (jerárquicas), al contrario que las estructuras planas del modelo de datos relacional. La Sección 27.2 se centra en la estructura de los documentos XML, y en los lenguajes para especificar la estructura de estos árboles, como DTD (Definición del tipo de documento, *Document Type Definition*) y XML Schema. La Sección 27.3 muestra la relación entre XML y las bases de datos relacionales y explica cómo pueden formatearse los datos extraídos de las bases de datos relacionales como documentos XML. La Sección 27.4 ofrece una introducción a los lenguajes de consulta XML: XPath y XQuery. Por último, la Sección 27.5 es el resumen del capítulo.

## 27.1 Modelo de datos jerárquico (árbol) de XML

Vamos a hacer una introducción al modelo de datos utilizado en XML. El objeto básico es XML en el documento XML. En la construcción de un documento XML se utilizan dos conceptos de estructuración principales: **elementos** y **atributos**. Es importante saber que el término atributo en XML no se utiliza de la misma forma que en la terminología de bases de datos, pero sí como se utiliza en los lenguajes de descripción de documentos, como HTML y SGML.[1] Los atributos en XML proporcionan información adicional que describe los elementos. En XML hay conceptos adicionales, como entidades, identificadores y referencias, pero primero nos concentraremos en describir los elementos y los atributos que muestran la esencia del modelo XML.

[1] SGML (Lenguaje de marcado generalizado estándar, *Standard Generalized Markup Language*) es un lenguaje más general para describir documentos y proporciona capacidades para especificar etiquetas nuevas. Sin embargo, es más complejo que HTML y XML.

**Figura 27.1.** Un elemento XML complejo denominado <Proyectos>.

```
<?xml version= “1.0” standalone=“yes”?>
<Proyectos>
  <Proyecto>
    <Nombre>ProductoX</Nombre>
    <Número>1</Número>
    <Ubicación>Valencia</Ubicación>
    <NumDpto>5</NumDpto>
    <Trabajador>
      <Dni>123456789</Dni>
      <Apellido1>Pérez</Apellido1>
      <Horas>32,5</Horas>
    </Trabajador>
    <Trabajador>
      <Dni>453453453</Dni>
      <NombreP>Aurora</NombreP>
      <Horas>20,0</Horas>
    </Trabajador>
  </Proyecto>
  <Proyecto>
    <Nombre>ProductoY</Nombre>
    <Número>2</Número>
    <Ubicación>Sevilla</Ubicación>
    <NumDpto>5</NumDpto>
    <Trabajador>
      <Dni>123456789</Dni>
      <Horas>7,5</Horas>
    </Trabajador>
    <Trabajador>
      <Dni>453453453</Dni>
      <Horas>20,0</Horas>
    </Trabajador>
    <Trabajador>
      <Dni>333445555</Dni>
      <Horas>10,0</Horas>
    </Trabajador>
  </Proyecto>
. . .
</Proyectos>
```

La Figura 27.1 muestra un ejemplo de elemento XML, denominado <Proyectos>. Como en HTML, los elementos están identificados dentro del documento por su etiqueta inicial y su etiqueta final. Los nombres de las etiquetas aparecen encerrados entre corchetes angulares, < ... >, y las etiquetas de cierre quedan identificadas además por una barra inclinada, </ ... >.[2]

[2] Los corchetes angulares (< y >) son caracteres especiales, como lo son el *ampersand* (&), el apóstrofe (’) y la comilla sencilla (‘). Para incluirlos dentro del texto de un documento, deben codificarse como <, >, &, ' y ", respectivamente.

Los **elementos complejos** se construyen jerárquicamente a partir de otros elementos, mientras que los **elementos simples** contienen valores de datos. Una diferencia importante entre XML y HTML es que los nombres de etiqueta de XML se definen para describir el significado de los elementos de datos del documento, y no para describir cómo se visualiza el texto. Esto hace posible que los programas de computador puedan procesar automáticamente los elementos de datos del documento XML.

Es muy fácil ver la correspondencia entre la representación textual de XML de la Figura 27.1 y la estructura en árbol de la Figura 26.1. En esta última, los nodos internos representan los elementos complejos, mientras que los nodos hoja representan los elementos sencillos. Es por esto que el modelo XML se denomina modelo de árbol o modelo jerárquico. En la Figura 27.1 los elementos sencillos son los que tienen los nombres de etiqueta <Nombre>, <Número>, <Ubicación>, <NumDpto>, <Dni>, <Apellido1>, <NombreP> y <Horas>. Los elementos complejos son los que tienen los nombres de etiqueta <Proyectos>, <Proyecto> y <Trabajador>. En general, no existe un límite en los niveles de anidamiento de elementos.

Es posible distinguir tres tipos principales de documentos XML:

- **Documentos XML centrados en los datos.** Estos documentos tienen muchos elementos de datos pequeños que respetan una estructura específica y, por tanto, pueden extraerse de una base de datos estructurada. Se formatean como documentos XML para intercambiarlos o visualizarlos por la Web.
- **Documentos XML centrados en el documento.** Son documentos con grandes cantidades de texto, como los artículos y los libros. En estos documentos hay pocos o ningún elemento de datos estructurado.
- **Documentos XML híbridos.** Estos documentos pueden tener partes que contienen datos estructurados y otras partes que son principalmente textuales o no estructuradas.

Es importante saber que los documentos XML centrados en los datos se pueden considerar como datos semiestructurados o como datos estructurados, como se define en la Sección 26.1. Si un documento XML obedece a un esquema XML predefinido o DTD (consulte la Sección 27.2), entonces el documento puede considerarse como de *datos estructurados*. Por el contrario, XML permite documentos que no obedecen a ningún esquema; estos documentos pueden considerarse como datos semiestructurados, que también se conocen como *documentos XML sin esquema*. Cuando el valor del atributo standalone de un documento XML es yes, como en la primera línea de la Figura 27.1, el documento es independiente y sin esquema.

Los atributos XML se utilizan generalmente de una forma parecida a como se utilizan en HTML (véase la Figura 26.2); por ejemplo, para describir las propiedades y las características de los elementos (etiquetas) dentro de las que aparecen. También es posible utilizar los atributos XML para conservar los valores de los elementos de datos sencillos; sin embargo, esto definitivamente no es recomendable.

En la Sección 27.2 explicamos los atributos XML, al explicar el esquema XML y DTD.

# 27.2 Documentos XML, DTD y XML Schema

## 27.2.1 Documentos XML bien formados y válidos y XML DTD

En la Figura 27.1 vimos el aspecto que puede tener un documento XML sencillo. Se dice que un documento XML está **bien formado** si cumple unas cuantas condiciones. En particular, debe empezar con una **declaración XML** para indicar la versión de XML que se está utilizando, así como cualesquiera otros atributos relevantes, como se muestra en la primera línea de la Figura 27.1. También debe obedecer a las directrices sintácticas del modelo en árbol. Esto significa que sólo debe haber un *elemento raíz*, y todos los elementos deben incluir un par de etiquetas inicial y final coincidentes, dentro de las etiquetas de inicio y cierre del *elemento padre*. Esto garantiza que los elementos anidados constituyen una estructura en árbol bien formada.

**Figura 27.2.** Un archivo XML DTD denominado *Proyectos*.

```
<!DOCTYPE Proyectos [
  <!ELEMENT Proyectos (Proyecto+)>
  <!ELEMENT Proyecto (Nombre, Número, Ubicación, NumDpto?, Trabajadores)>
  <!ELEMENT Nombre (#PCDATA)>
  <!ELEMENT Número (#PCDATA)>
  <!ELEMENT Ubicación (#PCDATA)>
  <!ELEMENT NumDpto (#PCDATA)>
  <!ELEMENT Trabajadores (Trabajador*)>
  <!ELEMENT Trabajador (Dni, Apellido1?, NombreP?, Horas)>
  <!ELEMENT Dni (#PCDATA)>
  <!ELEMENT Apellido1 (#PCDATA)>
  <!ELEMENT NombreP (#PCDATA)>
  <!ELEMENT Horas (#PCDATA)>
] >
```

Un documento XML bien formado es sintácticamente correcto. Esto permite que sea procesado por los procesadores genéricos que recorren el documento y crean una representación interna en forma de árbol. Un conjunto estándar de funciones API (Interfaz de programación de aplicaciones) denominado **DOM** (Modelo de objeto de documento) permite a los programas manipular la representación en árbol resultante correspondiente a un documento XML bien formado. No obstante, es preciso analizar de antemano el documento entero utilizando el DOM. Otra API, denominada **SAX**, permite el procesado de documentos XML sobre la marcha notificando al programa de procesamiento siempre que se encuentra una etiqueta de inicio o de cierre. Esto facilita el procesamiento de documentos grandes y permite procesar los denominados **documentos XML de *streaming***, donde el programa de procesamiento puede procesar las etiquetas a medida que se encuentran.

Un documento XML bien formado puede tener cualquier nombre de etiqueta para los elementos del documento. No hay un conjunto predefinido de elementos (nombres de etiqueta) que un programa que procesa el documento deba esperar encontrarse. Gracias a ello, el creador del documento tiene libertad para especificar elementos nuevos, pero se limita las posibilidades de interpretar automáticamente los elementos del documento.

Un criterio más estricto que un documento XML debe cumplir es el de la **validez**. En este caso, el documento debe estar bien formado, y los nombres de los elementos utilizados en los pares de etiquetas de inicio y fin deben seguir la estructura especificada en un archivo **DTD** (**Definición de tipo de documento**, *Document Type Definition*) separado o en un archivo de esquema XML. Vamos a explicar primero XML DTD, y después ofreceremos una visión general de XML Schema en la Sección 27.2.2. La Figura 27.2 muestra un archivo XML DTD sencillo, que especifica los elementos (nombres de etiqueta) y sus estructuras anidadas. Cualquier documento válido conforme a su DTD debe obedecer la estructura especificada. Existe una sintaxis especial para especificar archivos DTD, como se ilustra en la Figura 27.2. En primer lugar, se asigna un nombre a la **etiqueta raíz** del documento, que se denomina Proyectos en la primera línea de la Figura 27.2. Después se especifican los elementos y su estructura anidada.

Al especificar los elementos, se utiliza la siguiente notación:

- Un * a continuación del nombre del elemento significa que el elemento puede repetirse ninguna o más veces en el documento. Este tipo de elemento se conoce como *elemento (repetitivo) multivalor opcional*.
- Un + a continuación del nombre del elemento significa que el elemento puede repetirse una o más veces en el documento. Este tipo de elemento es un *elemento (repetitivo) multivalor obligatorio*.

- Un ? a continuación del nombre del elemento significa que el elemento puede repetirse ninguna o una vez. Este tipo es un *elemento (no repetitivo) monovalor opcional*.
- Un elemento que aparece sin ninguno de los tres símbolos anteriores debe aparecer exactamente una vez en el documento. Es un *elemento (no repetitivo) monovalor obligatorio*.
- El **tipo** del elemento se especifica mediante unos paréntesis a continuación del elemento. Si los paréntesis incluyen nombres de otros elementos, estos últimos son los *hijos* del elemento en la estructura en forma de árbol. Si los paréntesis incluyen la palabra clave #PCDATA o uno de los otros tipos de datos disponibles en XML DTD, el elemento es un nodo hoja. PCDATA se refiere a *datos de carácter analizados*, que son muy parecidos a un tipo de datos de cadena.
- Los paréntesis se pueden anidar al especificar los elementos.
- Un símbolo de barra ( *e*1 | *e*2 ) especifica que en el documento puede aparecer *e*1 o *e*2.

Podemos ver que la estructura en árbol de la Figura 26.1 y el documento XML de la Figura 27.1 son conformes al XML DTD de la Figura 27.2. Para forzar la comprobación de la conformidad de un documento XML con un DTD, debemos especificarlo en la declaración del documento. Por ejemplo, podríamos modificar la primera línea de la Figura 27.1 por lo siguiente:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE Proyectos SYSTEM "proj.dtd">
```

Cuando el valor del atributo standalone de un documento XML es "no", debe verificarse el documento respecto a un documento DTD independiente. El archivo DTD de la Figura 27.2 debe almacenarse en el mismo sistema de archivos que el documento XML, y se le debe asignar el nombre de archivo proj.dtd. Como alternativa, podríamos incluir el texto del documento DTD al principio del propio documento XML para permitir la verificación.

Aunque XML DTD es bastante adecuado para especificar estructuras en árbol con elementos obligatorios, opcionales y repetitivos, tiene varias limitaciones. En primer lugar, los tipos de datos en DTD no son muy generales. En segundo lugar, DTD tiene su propia sintaxis especial y, por tanto, requiere procesadores especializados. Sería beneficioso especificar documentos XML Schema utilizando las reglas sintácticas de XML para que los mismos procesadores utilizados para los documentos XML puedan procesar descripciones de XML Schema. En tercer lugar, todos los elementos DTD están obligados a seguir siempre el orden especificado del documento, por lo que no están permitidos los elementos desordenados. Estos inconvenientes llevaron al desarrollo de XML Schema, un lenguaje más general que sirve para especificar la estructura y los elementos de los documentos XML.

### 27.2.2 XML Schema

El **lenguaje XML Schema** es un estándar para especificar la estructura de los documentos XML. Utiliza las mismas reglas sintácticas que los documentos XML convencionales, por lo que pueden utilizarse los mismos procesadores en ambos. Para distinguir los dos tipos de documentos, utilizaremos el término *documento de instancia XML* o *documento XML* para un documento XML normal, y *documento de esquema XML* para un documento que especifica un esquema XML. La Figura 27.3 muestra un documento de esquema XML (en inglés) correspondiente a la base de datos EMPRESA de las Figuras 3.2 y 5.5. Aunque es improbable que queramos visualizar la base de datos entera como un documento simple, ha habido propuestas para almacenar datos en formato *XML nativo* como una alternativa a almacenar los datos en bases de datos relacionales. El esquema de la Figura 27.3 serviría para el propósito de especificar la estructura de la base de datos EMPRESA si se almacenara en un sistema XML nativo. Explicamos este tema en la Sección 27.3.

Como con XML DTD, XML Schema está basado en el modelo de datos en árbol, con los elementos y los atributos como los conceptos de estructuración principales. Sin embargo, toma prestados conceptos adicionales

de los modelos de bases de datos y de objetos, como las claves, las referencias y los identificadores. Aquí describimos por pasos las características de XML Schema, refiriéndonos al ejemplo de documento de esquema XML de la Figura 27.3. Introducimos y describimos algunos de los conceptos de esquema siguiendo el orden de utilización en la mencionada figura.

1. **Descripciones de esquema y espacios de nombres XML.** Es necesario identificar el conjunto específico de elementos del lenguaje XML Schema (etiquetas) que se utilizan especificando un archivo almacenado en una ubicación web. La segunda línea de la Figura 27.3 especifica el archivo utilizado en este ejemplo, que es http://www.w3.org/2001/XMLSchema. Es un estándar que se utiliza con frecuencia para los comandos de XML Schema. Toda definición semejante se denomina **espacio de nombres XML**, porque define el conjunto de comandos (nombres) que pueden utilizarse. El nombre de archivo se asigna a la variable xsd (descripción de XML Schema) utilizando el atributo xmlns (espacio de nombres XML), y esta variable se utiliza como prefijo para todos los comandos de XML Schema (nombres de etiqueta). Por ejemplo, en la Figura 27.3, cuando escribimos xsd:element o xsd:sequence, nos estamos refiriendo a las definiciones del elemento y la secuencia de etiquetas como se define en el archivo http://www.w3.org/2001/XMLSchema.

2. **Anotaciones, documentación y lenguaje utilizado.** Las dos siguientes líneas de la Figura 27.3 ilustran los elementos de XML Schema (etiquetas) xsd:annotation y xsd:documentation, que se utilizan para proporcionar comentarios y otras anotaciones en el documento XML. El atributo xml:lang del elemento xsd:documentation especifica el lenguaje que se está utilizando, donde en se refiere al idioma inglés.

3. **Elementos y tipos.** A continuación, especificamos el *elemento raíz* de nuestro esquema XML. En XML Schema, el atributo name de la etiqueta xsd:element especifica el nombre del elemento, que se denomina company para el elemento raíz en nuestro ejemplo (véase la Figura 27.3). La estructura del elemento raíz company puede especificarse después, que en nuestro ejemplo es xsd:complexType. Esto se define después para que sea una secuencia de departamentos, empleados y proyectos utilizando la estructura xsd:sequence de XML Schema. Es importante observar que no es la única forma de especificar un esquema XML para la base de datos EMPRESA. Veremos otras opciones en la Sección 27.3.

4. **Elementos de primer nivel en la base de datos EMPRESA.** A continuación, especificamos los tres elementos de primer nivel bajo el elemento raíz company de la Figura 27.3. Estos elementos son employee, department y project, y cada uno se especifica en una etiqueta xsd:element. Si una etiqueta sólo tiene atributos y ningún subelemento o dato en su interior, puede darse por finalizada directamente con el símbolo (/>) en lugar de tener la etiqueta de cierre correspondiente. Son los conocidos como **elementos vacíos**; ejemplos de ellos son los elementos xsd:element denominados department y project de la Figura 27.3.

5. **Especificar el tipo de elemento y las ocurrencias mínima y máxima.** En XML Schema, los atributos type, minOccurs y maxOccurs de la etiqueta xsd:element especifican el tipo y la multiplicidad de cada elemento en cualquier documento conforme a las especificaciones del esquema. Si especificamos un atributo type en xsd:element, la estructura del elemento debe describirse por separado, normalmente con el elemento xsd:complexType de XML Schema. En la Figura 27.3 se ilustra con los elementos employee, department y project. Por el contrario, si no especificamos el atributo type, podemos definir directamente la estructura del elemento a continuación de la etiqueta, como se ilustra con el elemento raíz company en la Figura 27.3. Las etiquetas minOccurs y maxOccurs se utilizan para especificar los límites inferior y superior del número de ocurrencias de un elemento en cualquier documento conforme a las especificaciones del esquema. Si no los especificamos, lo predeterminado es exactamente una ocurrencia. Es parecido a los símbolos *, + y ? de XML DTD, y las restricciones (mín, máx) del modelo ER (consulte la Sección 3.7.4).

**Figura 27.3.** Un archivo de XML Schema denominado *company*.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">Company Schema (Element Approach) - Prepared by Babak
        Hojabri</xsd:documentation>
  </xsd:annotation>
<xsd:element name="company">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="department" type="Department" minOccurs="0" maxOccurs=
          "unbounded" />
      <xsd:element name="employee" type="Employee" minOccurs="0" maxOccurs=
          "unbounded">
        <xsd:unique name="dependentNameUnique">
          <xsd:selector xpath="employeeDependent" />
          <xsd:field xpath="dependentName" />
        </xsd:unique>
      </xsd:element>
      <xsd:element name="project" type="Proyecto" minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:unique name="departmentNameUnique">
    <xsd:selector xpath="department" />
    <xsd:field xpath="departmentName" />
  </xsd:unique>
  <xsd:unique name="projectNameUnique">
    <xsd:selector xpath="project" />
    <xsd:field xpath="projectName" />
  </xsd:unique>
  <xsd:key name="projectNumberKey">
    <xsd:selector xpath="project" />
    <xsd:field xpath="projectNumber" />
  </xsd:key>
  <xsd:key name="departmentNumberKey">
    <xsd:selector xpath="department" />
    <xsd:field xpath="departmentNumber" />
  </xsd:key>
  <xsd:key name="employeeSSNKey">
    <xsd:selector xpath="employee" />
    <xsd:field xpath="employeeSSN" />
  </xsd:key>
  <xsd:keyref name="departmentManagerSSNKeyRef" refer="employeeSSNKey">
    <xsd:selector xpath="department" />
    <xsd:field xpath="departmentManagerSSN" />
  </xsd:keyref>
```

**Figura 27.3. (Continuación)** Un archivo de XML Schema denominado *company*.

```
    <xsd:keyref name="employeeDepartmentNumberKeyRef"
      refer="departmentNumberKey">
      <xsd:selector xpath="employee" />
      <xsd:field xpath="employeeDepartmentNumber" />
    </xsd:keyref>
    <xsd:keyref name="employeeSupervisorSSNKeyRef" refer="employeeSSNKey">
      <xsd:selector xpath="employee" />
      <xsd:field xpath="employeeSupervisorSSN" />
    </xsd:keyref>
    <xsd:keyref name="projectDepartmentNumberKeyRef" refer="departmentNumberKey">
      <xsd:selector xpath="project" />
      <xsd:field xpath="projectDepartmentNumber" />
    </xsd:keyref>
    <xsd:keyref name="projectWorkerSSNKeyRef" refer="employeeSSNKey">
      <xsd:selector xpath="project/projectWorker" />
      <xsd:field xpath="SSN" />
    </xsd:keyref>
    <xsd:keyref name="employeeWorksOnProjectNumberKeyRef"
      refer="projectNumberKey">
      <xsd:selector xpath="employee/employeeWorksOn" />
      <xsd:field xpath="projectNumber" />
    </xsd:keyref>
  </xsd:element>
  <xsd:complexType name="Department">
    <xsd:sequence>
      <xsd:element name="departmentName" type="xsd:string" />
      <xsd:element name="departmentNumber" type="xsd:string" />
      <xsd:element name="departmentManagerSSN" type="xsd:string" />
      <xsd:element name="departmentManagerStartDate" type="xsd:date" />
      <xsd:element name="departmentLocation" type="xsd:string" minOccurs="0"
        maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Employee">
    <xsd:sequence>
      <xsd:element name="employeeName" type="Name" />
      <xsd:element name="employeeSSN" type="xsd:string" />
      <xsd:element name="employeeSex" type="xsd:string" />
      <xsd:element name="employeeSalary" type="xsd:unsignedInt" />
      <xsd:element name="employeeBirthDate" type="xsd:date" />
      <xsd:element name="employeeDepartmentNumber" type="xsd:string" />
      <xsd:element name="employeeSupervisorSSN" type="xsd:string" />
      <xsd:element name="employeeAddress" type="Address" />
      <xsd:element name="employeeWorksOn" type="WorksOn" minOccurs="1"
```

**Figura 27.3. (Continuación)** Un archivo de XML Schema denominado *company*.

```
        maxOccurs="unbounded" />
      <xsd:element name="employeeDependent" type="Dependent" minOccurs="0"
        maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Project">
    <xsd:sequence>
      <xsd:element name="projectName" type="xsd:string" />
      <xsd:element name="projectNumber" type="xsd:string" />
      <xsd:element name="projectLocation" type="xsd:string" />
      <xsd:element name="projectDepartmentNumber" type="xsd:string" />
      <xsd:element name="projectWorker" type="Worker" minOccurs="1"
        maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Dependent">
    <xsd:sequence>
      <xsd:element name="dependentName" type="xsd:string" />
      <xsd:element name="dependentSex" type="xsd:string" />
      <xsd:element name="dependentBirthDate" type="xsd:date" />
      <xsd:element name="dependentRelationship" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Address">
    <xsd:sequence>
      <xsd:element name="number" type="xsd:string" />
      <xsd:element name="street" type="xsd:string" />
      <xsd:element name="city" type="xsd:string" />
      <xsd:element name="state" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Name">
    <xsd:sequence>
      <xsd:element name="firstName" type="xsd:string" />
      <xsd:element name="middleName" type="xsd:string" />
      <xsd:element name="lastName" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Worker">
    <xsd:sequence>
      <xsd:element name="SSN" type="xsd:string" />
      <xsd:element name="hours" type="xsd:float" />
    </xsd:sequence>
  </xsd:complexType>
```

**Figura 27.3. (Continuación)** Un archivo de XML Schema denominado company.

```
<xsd:complexType name="WorksOn">
  <xsd:sequence>
    <xsd:element name="projectNumber" type="xsd:string" />
    <xsd:element name="hours" type="xsd:float" />
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

6. **Especificación de claves.** En XML Schema, es posible especificar restricciones que se correspondan con las restricciones de clave única y principal en una base de datos relacional (consulte la Sección 5.2.2), así como con las restricciones de claves externas (o integridad referencial) (consulte la Sección 5.2.4). La etiqueta xsd:unique especifica los elementos que corresponden a los atributos únicos en una base de datos relacional que no son claves primarias. A estas restricciones de unicidad podemos asignarles un nombre, y debemos especificar las etiquetas xsd:selector y xsd:field para ellas a fin de identificar el tipo de elemento que contiene el elemento único y el nombre del elemento que dentro de él es único a través del atributo xpath. Es lo que se ilustra con los elementos departmentNameUnique y projectNameUnique en la Figura 27.3. Para especificar las **claves principales** se utiliza la etiqueta xsd:key en sustitución de xsd:unique, como se ilustra con los elementos projectNumberKey, departmentNumberKey y employeeSSNKey de la Figura 27.3. Para especificar las **claves externas**, se utiliza la etiqueta xsd:keyref, como se ilustra con los seis elementos xsd:keyref de la Figura 27.3. Al especificar una clave externa, el atributo referido por la etiqueta xsd:keyref especifica la clave primaria referenciada, mientras que las etiquetas xsd:selector y xsd:field especifican el tipo de elemento referenciado y la clave externa (véase la Figura 27.3).

7. **Especificar las estructuras de los elementos complejos mediante tipos complejos.** La siguiente parte de nuestro ejemplo especifica las estructuras de los elementos complejos Department, Employee, Proyecto y Dependent, utilizando la etiqueta xsd:complexType (véase la Figura 27.5). Especificamos cada una de éstas como una secuencia de subelementos correspondientes a los atributos de base de datos de cada tipo de entidad (véanse las Figuras 3.2 y 5.7) mediante las etiquetas xsd:sequence y xsd:element de XML Schema. A cada elemento se le asigna un nombre y un tipo mediante los atributos name y type de xsd:element. También podemos especificar los atributos minOccurs y maxOccurs si necesitamos cambiar el valor predeterminado de exactamente una ocurrencia. Para los atributos de la base de datos (opcionales) en los que está permitido el valor null, tenemos que especificar minOccurs = 0, mientras que para los atributos multivalor de base de datos debemos especificar maxOccurs = "unbounded" en el elemento correspondiente. Si no vamos a especificar restricciones de clave, tendríamos que incrustar los subelementos directamente dentro de las definiciones de elemento padre sin tener que especificar los tipos complejos. No obstante, tenemos que especificar las restricciones de clave única y principal, y las de clave externa; debemos definir los tipos complejos para especificar las estructuras de elemento.

8. **Atributos compuestos.** Los atributos compuestos de la Figura 3.2 también se especifican como tipos complejos en la Figura 27.5, como se ilustra con los tipos complejos Direcc, Nombre, Trabajador y TrabajaEn. Se podrían incrustar directamente dentro de sus elementos padre.

Este ejemplo ilustra algunas de las características principales de XML Schema. Hay otras muchas características, pero quedan fuera del objetivo de nuestra presentación. En la siguiente sección explicamos los diferentes métodos para crear documentos XML a partir de bases de datos relacionales y almacenar documentos XML.

# 27.3 Documentos XML y bases de datos

Ahora vamos a ver cómo podemos almacenar y recuperar distintos tipos de documentos XML. La Sección 27.3.1 ofrece una visión general de los distintos métodos que hay para almacenar documentos XML. La Sección 27.3.2 explica en detalle uno de estos métodos, en el que extraemos documentos XML basados en datos de las bases de datos existentes. En particular, hablaremos de cómo se pueden crear documentos estructurados en árbol a partir de las bases de datos estructuradas como un gráfico. La Sección 27.3.3 explica el problema de los ciclos y de cómo tratarlos.

## 27.3.1 Métodos para almacenar documentos XML

Se han propuesto varios métodos para organizar el contenido de los documentos XML a fin de facilitar su posterior consulta y recuperación. Los métodos más comunes son los siguientes:

1. **Uso de un DBMS para almacenar los documentos como texto.** Es posible utilizar un DMBS de objetos o relacional para almacenar documentos XML enteros como campos de texto dentro de los objetos o registros del DBMS. Este método se puede utilizar si el DBMS tiene un módulo especial para el procesamiento de documentos, y funcionará para almacenar documentos XML sin esquema y centrados en el documento. Las funciones de indexación del módulo de procesamiento del documento (consulte el Capítulo 22) pueden utilizarse para indexar y acelerar la búsqueda y la recuperación de los documentos.
2. **Uso de un DBMS para almacenar el contenido del documento como elementos de datos.** Este método funcionaría para almacenar una colección de documentos que obedecen un esquema XML específico o XML DTD. Como todos los documentos tienen la misma estructura, podemos diseñar una base de datos relacional (o de objetos) para almacenar los elementos de datos a nivel de hoja dentro de los documentos XML. Este método requiere algoritmos de mapeado para diseñar un esquema de base de datos compatible con la estructura del documento XML tal como está especificada en el esquema XML o en el DTD y para recrear los documentos XML a partir de los datos almacenados. Estos algoritmos se pueden implementar como un módulo DBMS interno o como *middleware* independiente que no forma parte del DBMS.
3. **Diseño de un sistema especializado para almacenar datos XML nativos.** Podría diseñarse e implementarse un nuevo tipo de sistema de bases de datos basado en el modelo jerárquico (árbol). Dichos sistemas se denominan **DBMSs XML nativos**. El sistema debería incluir técnicas de indexación y consulta especializadas, y debería funcionar para todos los tipos de documentos XML. También podría incluir técnicas de compresión de datos para reducir el tamaño de los documentos de cara a su almacenamiento. Tamino de Software AG y Dynamic Application Platform de eXcelon son dos conocidos productos que ofrecen la funcionalidad de DBMS XML nativo.
4. **Creación o publicación de documentos XML personalizados a partir de bases de datos relacionales pre-existentes.** Como en las bases de datos relacionales hay cantidades enormes de datos almacenados, puede darse la necesidad de tener que formatear parte de estos datos como documentos para intercambiarlos o visualizarlos por la Web. Este método utilizaría una capa software *middleware* separada para llevar a cabo las conversiones necesarias entre los documentos XML y la base de datos relacional.

Todos estos métodos han recibido una atención considerable durante los últimos años. En la siguiente subsección nos ocuparemos del cuarto método, porque ofrece una buena noción conceptual de las diferencias entre el modelo de datos en árbol de XML y los modelos de bases de datos tradicionales basados en los ficheros planos (modelo relacional) y las representaciones gráficas (modelo ER).

### 27.3.2 Extracción de documentos XML a partir de bases de datos relacionales

Esta sección explica los problemas de representación que surgen al convertir datos de un sistema de bases de datos en documentos XML. Como hemos explicado, XML utiliza un modelo jerárquico (árbol) para representar los documentos. Los sistemas de bases de datos utilizan ampliamente el modelo de datos relacional plano. Al añadir restricciones de integridad referencial, podemos considerar que un esquema relacional es una estructura gráfica (por ejemplo, véase la Figura 5.7). De forma parecida, el modelo ER representa los datos utilizando estructuras parecidas a gráficos (por ejemplo, véase la Figura 3.2). En el Capítulo 7 vimos que hay unos mapeados directos entre los modelos ER y relacional, por lo que podemos representar conceptualmente un esquema de base de datos relacional utilizando el esquema ER correspondiente. Aunque utilizaremos el modelo ER en nuestra explicación y algunos ejemplos para aclarar las diferencias conceptuales entre los modelos en árbol y gráfico, los mismos problemas se pueden aplicar a la conversión de datos relacionales en XML.

Utilizaremos el esquema ER UNIVERSIDAD de la Figura 27.4 para ilustrar nuestra explicación. Supongamos que una aplicación necesita extraer documentos XML con información de estudiante, curso y calificación a partir de la base de datos UNIVERSIDAD. Los datos necesarios para estos documentos están contenidos en los atributos de la base de datos de los tipos de entidad CURSO, NIVEL y ESTUDIANTE (véase la Figura 27.4), y las relaciones E-N y C-N entre ellos. En general, la mayoría de los documentos extraídos de una base de datos sólo utilizarán un subconjunto de los atributos, tipos de entidad y relaciones de la base de datos. En este ejemplo, el subconjunto de la base de datos necesario se muestra en la Figura 27.5.

**Figura 27.4.** Esquema ER para una base de datos UNIVERSIDAD simplificada.

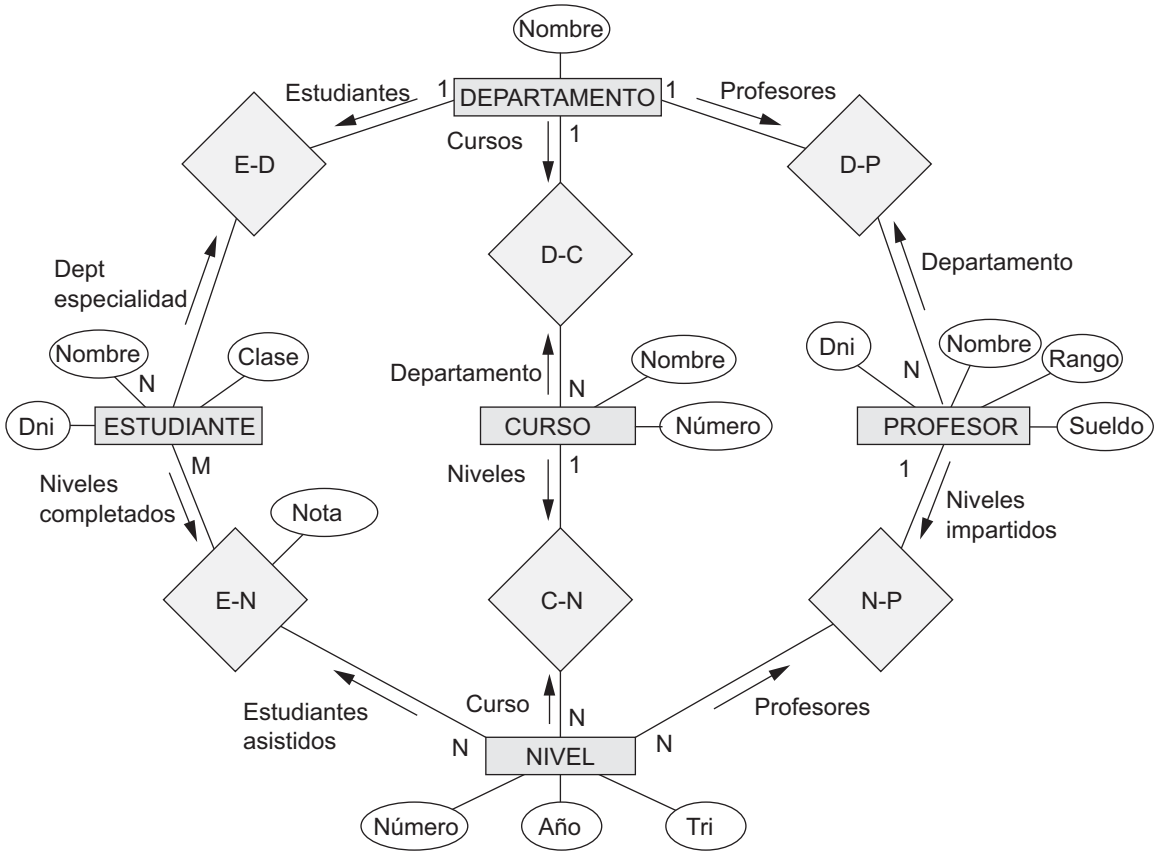


**Figura 27.5.** Subconjunto del esquema de la base de datos UNIVERSIDAD necesario para la extracción del documento XML.

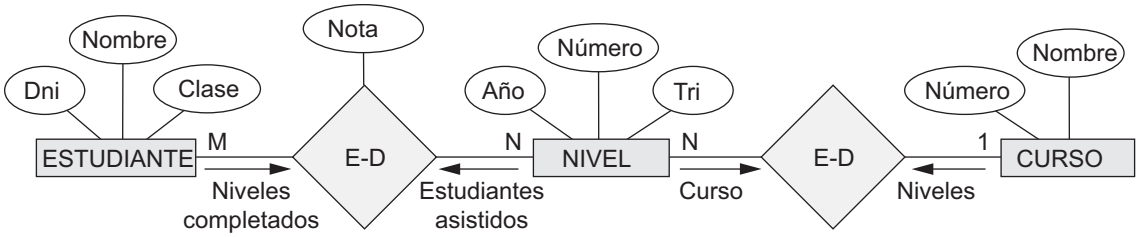
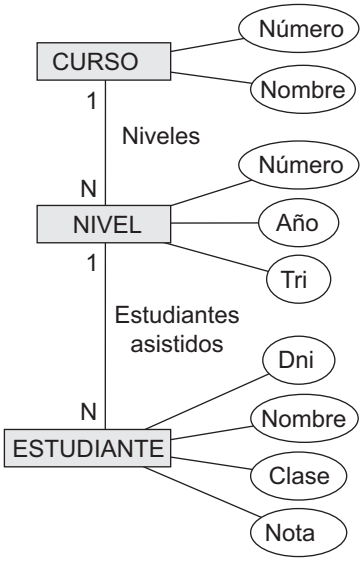



**Figura 27.6.** Vista jerárquica (árbol) con CURSO como raíz.




Del subconjunto de base de datos de la Figura 27.5 se pueden extraer al menos tres posibles jerarquías de documento. Primero, podemos seleccionar CURSO como la raíz (véase la Figura 27.6). Aquí, cada entidad curso tiene el conjunto de sus niveles como subelementos, y cada nivel tiene sus estudiantes como subelementos. Podemos ver una consecuencia de modelar la información en una estructura de árbol jerárquica. Si un estudiante aparece en varios niveles, la información de ese estudiante aparecerá varias veces en el documento; una por cada nivel. Un posible esquema XML simplificado para esta vista se muestra en la Figura 27.7. El atributo Nota de la base de datos en la relación E-N se migra al elemento ESTUDIANTE. Esto es así porque ESTUDIANTE se convierte en hijo de NIVEL en esta jerarquía, por lo que cada elemento ESTUDIANTE bajo un elemento NIVEL específico puede tener una nota específica en ese nivel. En esta jerarquía de documento, un estudiante que aparece en más de un nivel tendrá varias réplicas, una bajo cada nivel, y cada réplica tendrá la nota específica de ese nivel en particular.

En la segunda vista de documento jerárquico, podemos elegir ESTUDIANTE como raíz (véase la Figura 27.8). En esta vista jerárquica cada estudiante tiene un conjunto de niveles como elementos hijo, y cada nivel está relacionado con un curso como su hijo, porque la relación entre NIVEL y CURSO es N:1. Podemos, por tanto, mezclar los elementos CURSO y NIVEL en esta vista (véase la Figura 27.8). Además, el atributo de base de datos NOTA se puede migrar al elemento NIVEL. En esta jerarquía, la información CURSO/NIVEL combinada se duplica bajo cada estudiante que haya completado el nivel. Un posible esquema XML simplificado para esta vista es el mostrado en la Figura 27.9.

**Figura 27.7.** Documento de esquema XML con *curso* como raíz.

```
<xsd:element name="root">
<xsd:sequence>
<xsd:element name="curso" minOccurs="0" maxOccurs="unbounded">
  <xsd:sequence>
    <xsd:element name="nombrec" type="xsd:string" />
    <xsd:element name="númeroc" type="xsd:unsignedInt" />
    <xsd:element name="nivel" minOccurs="0" maxOccurs="unbounded">
      <xsd:sequence>
        <xsd:element name="númeronivel" type="xsd:unsignedInt" />
        <xsd:element name="anio" type="xsd:string" />
        <xsd:element name="trimestre" type="xsd:string" />
        <xsd:element name="estudiante" minOccurs="0" maxOccurs="unbounded">
          <xsd:sequence>
            <xsd:element name="dni" type="xsd:string" />
            <xsd:element name="nombree" type="xsd:string" />
            <xsd:element name="clase" type="xsd:string" />
            <xsd:element name="nota" type="xsd:string" />
          </xsd:sequence>
        </xsd:element>
      </xsd:sequence>
    </xsd:element>
  </xsd:sequence>
</xsd:element>
</xsd:sequence>
</xsd:element>
```

La tercera forma posible es elegir NIVEL como raíz (véase la Figura 27.10). De forma parecida a la segunda vista jerárquica, la información de CURSO se puede mezclar en el elemento NIVEL. El atributo de base de datos NOTA se puede migrar al elemento ESTUDIANTE. Como podemos ver, incluso en este sencillo ejemplo, puede haber muchas vistas jerárquicas de un documento, cada una correspondiente a una raíz diferente y una estructura de documento XML distinta.

## 27.3.3 Rotura de ciclos para convertir gráficos en árboles

En los ejemplos anteriores el subconjunto de la base de datos no tenía ciclos. Es posible tener un subconjunto más complejo con uno o más ciclos, indicando las muchas relaciones entre las entidades. En este caso, es más complicado decidir cómo crear las jerarquías del documento. Es posible que tengamos que duplicar entidades adicionales para representar las distintas relaciones. Ilustraremos esto con un ejemplo que utiliza el esquema ER de la Figura 27.4.

Supongamos que necesitamos la información de todos los tipos de entidad y relaciones de la Figura 27.4 para un documento XML concreto, con ESTUDIANTE como elemento raíz. La Figura 27.11 ilustra cómo podemos crear una posible estructura jerárquica en árbol para este documento. En primer lugar, obtenemos un entramado con ESTUDIANTE como raíz (véase la Figura 27.11[a]). No es una estructura en árbol debido a los ciclos. Una forma de romper los ciclos es replicar los tipos de entidad implicados en ellos. Primero, duplicamos PROFESOR como se muestra en la Figura 27.11(b), y denominamos a la copia de la derecha PROFESOR1. La

**Figura 27.8.** Vista jerárquica (árbol) con ESTUDIANTE como raíz.

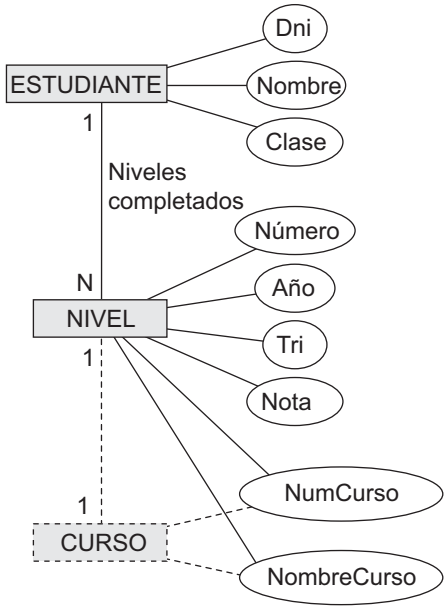



**Figura 27.9.** Documento de esquema XML con *estudiante* como raíz.

```
<xsd:element name=”root”>
<xsd:sequence>
<xsd:element name=”estudiante” minOccurs=”0” maxOccurs=”unbounded”>
  <xsd:sequence>
    <xsd:element name=”dni” type=”xsd:string” />
    <xsd:element name=”nombree” type=”xsd:string” />
    <xsd:element name=”clase” type=”xsd:string” />
    <xsd:element name=”nivel” minOccurs=”0” maxOccurs=”unbounded”>
      <xsd:sequence>
        <xsd:element name=”numeronivel” type=”xsd:unsignedInt” />
        <xsd:element name=”anio” type=”xsd:string” />
        <xsd:element name=”trimestre” type=”xsd:string” />
        <xsd:element name=”númeroc” type=”xsd:unsignedInt” />
        <xsd:element name=”nombrec” type=”xsd:string” />
        <xsd:element name=”nota” type=”xsd:string” />
      </xsd:sequence>
    </xsd:element>
  </xsd:sequence>
</xsd:element>
</xsd:sequence>
</xsd:element>
```

réplica PROFESOR de la izquierda representa la relación entre los profesores y los niveles que imparten, mientras que la réplica PROFESOR1 de la derecha representa la relación entre los profesores y el departa-

**Figura 27.10.** Vista jerárquica (árbol) con NIVEL como raíz.

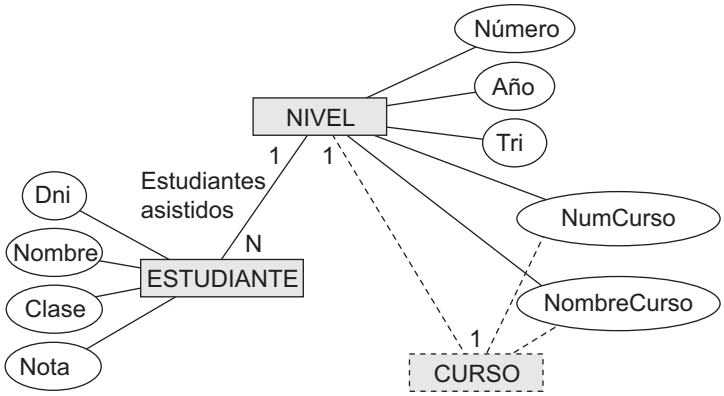
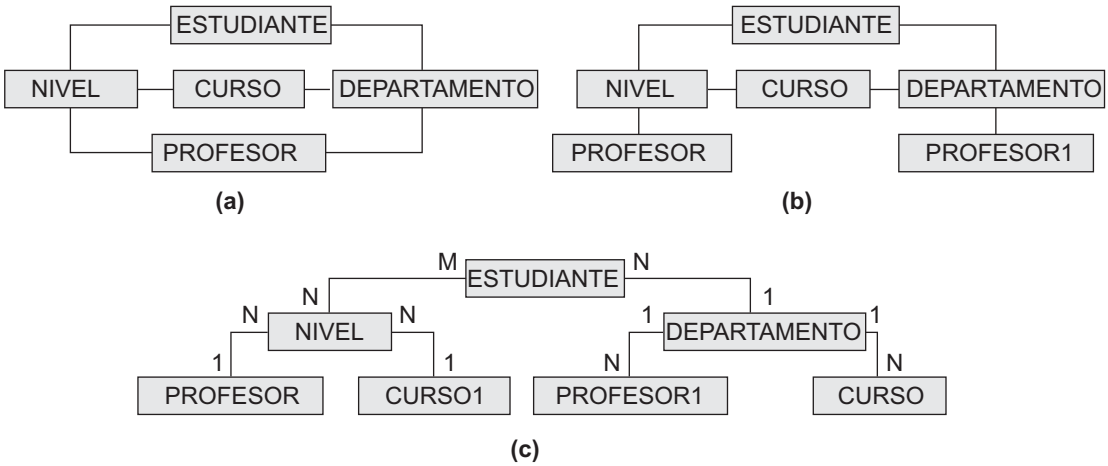



**Figura 27.11.** Conversión de un gráfico con ciclos en una estructura jerárquica (árbol).




mento en el que trabajan. Después de esto, todavía tenemos el ciclo que implica el CURSO, por lo que podemos duplicar CURSO de una forma parecida, lo que nos lleva a la jerarquía de la Figura 27.11(c). La réplica CURSO1 de la izquierda representa la relación entre los cursos y sus niveles, mientras que la réplica CURSO de la derecha representa la relación entre los cursos y el departamento que ofrece cada curso.

En la Figura 27.11(c), hemos convertido el gráfico inicial en una jerarquía. Si lo deseamos, podemos seguir mezclando (como en nuestro ejemplo anterior) antes de crear la jerarquía final y la estructura de esquema XML correspondiente.

### 27.3.4 Otros pasos para extraer documentos XML a partir de bases de datos

Además de crear la jerarquía XML apropiada y el documento de esquema XML correspondiente, necesitamos más pasos para extraer un documento XML concreto a partir de una base de datos.

1. Es necesario crear la consulta correcta en SQL para extraer la información deseada para el documento XML.

2. Una vez ejecutada la consulta, su resultado debe estructurarse para pasar de la forma relacional plana a la estructura de árbol XML.
3. La consulta debe personalizarse para seleccionar uno o varios objetos en el documento. Por ejemplo, en la vista de la Figura 27.9, la consulta puede seleccionar una sola entidad de estudiante y crear un documento correspondiente a ese único estudiante, o puede seleccionar varios (o incluso todos) los estudiantes y crear un documento con varios estudiantes.

# 27.4 Consulta XML

Hay varias propuestas como lenguajes de consulta XML, pero destacan dos estándares. El primero es XPath, que proporciona estructuras de lenguaje para especificar las expresiones de ruta que permitan identificar ciertos nodos (elementos) dentro de un documento XML que coincidan con patrones específicos. El segundo es XQuery, que es un lenguaje de consulta más general. XQuery utiliza expresiones XPath, pero ofrece construcciones adicionales. En esta sección ofrecemos una visión general de cada uno de estos lenguajes.

## 27.4.1 XPath: especificar expresiones de ruta en XML

Una expresión XPath devuelve una colección de nodos elemento que satisfacen los patrones especificados en la expresión. Los nombres que aparecen en la expresión XPath son nombres de nodo del árbol del documento XML, que son los nombres de las etiquetas (elemento) o los nombres de los atributos, posiblemente con **condiciones de calificador** adicionales para restringir después los nodos que cumplen el patrón. Al especificar una ruta se suelen utilizar dos **separadores**: una sola barra inclinada (/) y una barra inclinada doble (//). Una sola barra inclinada delante de una etiqueta indica que la etiqueta debe aparecer como un hijo directo de la etiqueta anterior (padre), mientras que una barra inclinada doble indica que la etiqueta puede aparecer como un descendiente de la etiqueta anterior *en cualquier nivel*. Ofreceremos algunos ejemplos de XPath (véase la Figura 27.12).

La primera expresión XPath de la Figura 27.12 devuelve el nodo raíz company y todos sus nodos descendientes, es decir, devuelve el documento XML entero. Debemos saber que es costumbre incluir el nombre del fichero en la consulta XPath. Esto nos permite especificar cualquier nombre de fichero local o, incluso, cualquier ruta de acceso correspondiente a un fichero en la Web. Por ejemplo, si el documento XML EMPRESA está almacenado en la siguiente ubicación:

```
www.company.com/info.XML
```

La primera expresión XPath de la Figura 27.12 puede escribirse como:

```
doc(www.company.com/info.XML)/company
```

Este prefijo también debería incluirse en los demás ejemplos.

El segundo ejemplo de la Figura 27.12 devuelve todos los nodos de departamento (elementos) y sus subárboles descendientes. Los nodos (elementos) de un documento XML están ordenados, por lo que el resultado de XPath que devuelva varios nodos lo hará siguiendo el mismo orden en que aparecen los nodos en el árbol del documento.

La tercera expresión XPath de la Figura 27.12 ilustra el uso de //, que es conveniente utilizar si no conocemos la ruta de acceso completa que estamos buscando, pero conocemos el nombre de algunas etiquetas de interés dentro del documento XML. Esto es particularmente útil para los documentos XML sin esquema o para los documentos con muchos niveles de nodos anidados.[3]

[3] Aquí utilizamos indistintamente los términos nodo, etiqueta y elemento.

**Figura 27.12.** Algunos ejemplos de expresiones XPath en los documentos XML que obedecen al fichero de esquema XML *company* de la Figura 27.3.

1. /company
2. /company/department
3. //employee [employeeSalary gt 70000]/employeeName
4. /company/employee [employeeSalary gt 70000]/employeeName
5. /company/project/projectWorker [hours ge 20.0]

---

**Figura 27.13.** Algunos ejemplos de consultas XQuery sobre documentos XML que obedecen al fichero de esquema XML *company* de la Figura 27.3.

**1.** FOR $x IN
doc(www.company.com/info.xml)
//employee [employeeSalary gt 70000]/employeeName
RETURN <res> $x/firstName, $x/lastName </res>

**2.** FOR $x IN
doc(www.company.com/info.xml)/company/employee
WHERE $x/employeeSalary gt 70000
RETURN <res> $x/employeeName/firstName, $x/employeeName/lastName </res>

**3.** FOR $x IN
doc(www.company.com/info.xml)/company/project[projectNumber = 5]/projectWorker,
$y IN doc(www.company.com/info.xml)/company/employee
WHERE $x/hours gt 20.0 AND $y.ssn = $x.ssn
RETURN <res> $y/employeeName/firstName, $y/employeeName/lastName, $x/hours </res>

---

La expresión devuelve todos los nodos employeeName que son hijos directos de un nodo employee, tal que el nodo employee tiene otro elemento hijo employeeSalary cuyo valor es mayor que 70.000. Esto ilustra el uso de las condiciones de calificador, que restringen los nodos seleccionados por la expresión XPath a aquellos que satisfacen la condición. XPath tiene varias operaciones de comparación que podemos utilizar en las condiciones de calificador, incluyendo operaciones estándar para cálculos aritméticos, manipulación de cadenas y comparación de conjuntos.

La cuarta expresión de XPath debe devolver el mismo resultado que la anterior, solo que hemos especificado la ruta de acceso completa. La quinta expresión de la Figura 27.12 devuelve todos los nodos projectWorker y sus nodos descendientes que son hijos y que están bajo una ruta de acceso /company/project y tienen un nodo hijo hours con un valor superior a 20.0 horas.

## 27.4.2 XQuery: especificación de consultas en XML

XPath permite escribir expresiones que seleccionan nodos a partir de un documento XML estructurado en forma de árbol. XQuery permite especificar consultas más generales sobre uno o más documentos XML. La forma típica de una consulta en XQuery se conoce como **expresión FLWR**, que hace referencia a las cuatro cláusulas principales de XQuery y que tienen la siguiente forma:

FOR <asociaciones de variables a nodos (elementos) individuales>
LET <asociaciones de variables a colecciones de nodos (elementos)>

WHERE <condiciones de calificador>
RETURN <especificación del resultado de la consulta>

La Figura 27.13 incluye algunos ejemplos de consultas en XQuery que se pueden especificar en documentos de instancia XML y que obedezcan al documento de esquema XML de la Figura 27.3. La primera consulta recupera el nombre del primer y del último empleado que ganan más de 70.000 dólares. La variable $x está asociada a cada elemento employeeName hijo de un elemento employee, pero sólo para los elementos employee que satisfacen el calificador de que su valor employeeSalary es mayor que 70.000 dólares. El resultado recupera los elementos hijo firstName y lastName de los elementos employeeName seleccionados. La segunda consulta es una forma alternativa de recuperar los mismos elementos que recupera la primera consulta.

La tercera consulta ilustra cómo puede realizarse la operación de concatenación con más de una variable. Aquí, la variable $x está asociada a cada elemento projectWorker hijo del proyecto número 5, mientras que la variable $y se asocia a cada elemento empleado. La condición de concatenación empareja valores de ssn con el objetivo de recuperar los nombres de los empleados.

Esto concluye nuestra breve introducción a XQuery. El lector interesado puede consultar www.w3.org, que contiene documentos que describen los últimos estándares relacionados con XML.

# 27.5 Resumen

Este capítulo ofrece una visión general del estándar de representación e intercambio de datos por Internet. Hemos descrito el estándar XML y su modelo de datos (jerárquico) estructurado en forma de árbol, así como los documentos XML y los lenguajes que permiten especificar la estructura de dichos documentos, en particular, XML DTD (Definición del tipo de documento) y XML Schema. También hemos ofrecido una visión general de los distintos métodos de almacenamiento de documentos XML, tanto en su formato nativo (texto), como en su formato comprimido o relacional (y otros tipos de bases de datos). También hemos explicado los problemas de asignación que surgen cuando es necesario convertir los datos almacenados en las bases de datos tradicionales en documentos XML. Por último, ofrecimos una panorámica de los lenguajes XPath y XQuery propuestos para consultar los datos XML.

## Preguntas de repaso

**27.1.** ¿Cuáles son las diferencias entre datos estructurados, semiestructurados y no estructurados?

**27.2.** ¿Bajo cuál de las categorías anteriores podemos encuadrar los documentos XML? ¿Qué puede decir sobre los datos autodescriptivos?

**27.3.** ¿Cuáles son las diferencias entre el uso de etiquetas en XML frente a HTML?

**27.4.** ¿Cuál es la diferencia entre documentos XML centrados en los datos y los centrados en los documentos?

**27.5.** ¿Cuál es la diferencia entre atributos y elementos en XML? Enumere algunos de los atributos más importantes que se utilizan para especificar los elementos en XML Schema.

**27.6.** ¿Cuál es la diferencia entre XML Schema y XML DTD?

## Ejercicios

**27.7.** Cree parte de un documentos de instancia XML que se corresponda con los datos almacenados en la base de datos relacional de la Figura 5.6, de forma parecida al documento XML Schema de la Figura 27.3.

**27.8.** Cree documentos de esquema XML y XML DTDs correspondientes a las jerarquías mostradas en las Figuras 27.10 y 27.11(c).

**27.9.** Considerando el esquema de la base de datos relacional BIBLIOTECA de la Figura 6.14, cree un documento de esquema XML que se corresponda con este esquema de base de datos.

**27.10.** Especifique las siguientes vistas y consultas en XQuery sobre el esquema XML company de la Figura 27.3.

**a.** Una vista que muestre el nombre del departamento, el nombre del director y el sueldo del director por cada departamento.

**b.** Una vista que muestre el nombre de empleado, el nombre de su supervisor y el sueldo de dicho empleado por cada empleado que trabaje en el departamento "Research" (investigación).

**c.** Una vista que muestre, por cada proyecto, el nombre de éste, el nombre del departamento que lo gestiona, el número de empleados y la cantidad total de horas trabajadas por semana en dicho proyecto.

**d.** Una vista que muestre, por cada proyecto en el que trabaje más de un empleado, el nombre de éste, el nombre del departamento que lo gestiona, el número de empleados y la cantidad total de horas trabajadas por semana en dicho proyecto.

## Bibliografía seleccionada

Hay tantos libros y artículos sobre los distintos aspectos de XML que sería imposible incluso hacer una lista modesta. Mencionaremos un libro: Chaudhri, Rashid y Zicari, eds (2003). Este libro explica varios aspectos de XML y contiene una lista de algunas referencias recientes que permiten investigar y practicar con XML.

CAPÍTULO 28

# Conceptos de minería de datos

A lo largo de las tres últimas décadas, muchas organizaciones han generado una gran cantidad de datos mecanizados en forma de ficheros y bases de datos. Para procesar estos datos, disponemos de la tecnología de bases de datos que proporciona lenguajes de consulta como SQL. El problema es que SQL es un lenguaje estructurado en el que se supone que el usuario tiene conocimientos del esquema de la base de datos. SQL soporta operaciones de álgebra relacional que permiten al usuario seleccionar filas y columnas de datos a partir de tablas o de información recopilada a partir de tablas que tienen campos comunes. En el siguiente capítulo, veremos que la *tecnología de almacenes de datos* proporciona varios tipos de funcionalidades: consolidación, agregación y resumen de los datos. Los almacenes de datos nos permiten ver la misma información bajo distintas ópticas. En este capítulo, fijaremos nuestra atención en otra área de interés muy conocida, la minería de datos. Como indica su nombre, la minería de datos tiene relación con la extracción o descubrimiento de nueva información en términos de patrones o reglas a partir de grandes cantidades de datos. Para que resulte útil en la práctica, la **minería de datos** debe ser puesta en práctica de modo eficiente sobre archivos y bases de datos de gran tamaño. Actualmente, *no* está bien integrada con los sistemas de gestión de bases de datos.

Haremos un breve repaso al estado actual de este amplio campo de la minería de datos, que utiliza técnicas tomadas de áreas como el aprendizaje artificial, la estadística, las redes neuronales y los algoritmos genéticos. Fijaremos nuestra atención en la naturaleza de la información que se obtiene, los tipos de problemas con los que nos encontramos al hacer minería en las bases de datos y los tipos de aplicaciones de minería de datos. También repasaremos el estado actual de un gran número de herramientas comerciales disponibles (consulte la Sección 28.7) y describiremos varios avances necesarios para que esta área sea viable.

## 28.1 Repaso a la tecnología de minería de datos

En algunos informes, como el conocido Informe Gartner,[1] la minería de datos aparece como una de las tecnologías de más éxito en un futuro próximo. En esta sección relacionaremos la minería de datos con otra área más extensa llamada descubrimiento del conocimiento y compararemos ambas mediante un ejemplo ilustrativo.

[1] El Informe Gartner es un ejemplo de las diferentes publicaciones de estudios en los que se basan los gestores de las corporaciones para tomar sus decisiones sobre tecnología.

### 28.1.1 Comparación entre minería de datos y almacenes de datos

El objetivo de un almacén de datos (consulte el Capítulo 29) es ayudar a la toma de decisiones basadas en los datos. La minería de datos puede ser usada en combinación con un almacén de datos para ayudar a la toma de determinados tipos de decisiones. La minería de datos se puede aplicar a las bases de datos operacionales en las transacciones individuales. Para que la minería de datos sea más eficaz, el almacén de datos debería disponer de un conjunto de datos agregado o resumido. La minería de datos ayuda en la extracción de nuevos patrones con significado que no se mostrarían sólo con efectuar consultas o procesar los datos o metadatos del almacén de datos. Según esto, las aplicaciones de minería de datos deberían ser tenidas en cuenta en las primeras fases del diseño de un almacén de datos. Además, las herramientas de minería de datos deberían ser diseñadas para facilitar su uso en conjunción con los almacenes de datos. De hecho, en el caso de bases de datos de gran tamaño que trabajan con terabytes de datos, el uso provechoso de las aplicaciones de minería de datos dependerá principalmente de la construcción de un almacén de datos.

### 28.1.2 La minería de datos como parte del proceso del descubrimiento de conocimiento

El **Descubrimiento de conocimiento en bases de datos**, abreviado normalmente como **KDD** (*Knowledge Discovery in Databases*), es por lo general un tema más amplio que la minería de datos. El proceso de descubrimiento de conocimiento comprende seis fases:[2] selección de datos, purgado de datos, enriquecimiento, transformación de datos o codificación, minería de datos, y obtención de informes y visualización de la información recopilada.

Como ejemplo, tomemos una base de datos transaccional gestionada por un suministrador de bienes de consumo de especialidades alimenticias. Supongamos que los datos del cliente incluyen su nombre, código postal, número de teléfono, fecha de compra, código de producto, precio, número de productos y precio total. Mediante el proceso de descubrimiento de conocimientos sobre esta base de datos de clientes es posible obtener diversa información. Durante la *selección de datos*, es posible seleccionar datos en relación con productos en concreto, con tipos de productos, o con tiendas situadas en una determinada zona del país. El proceso de *purgado de datos* podría, a continuación, corregir códigos postales no válidos o eliminar registros con prefijos telefónicos incorrectos. El *enriquecimiento* mejora, por lo general, los datos a partir de fuentes de información adicionales. Por ejemplo, dados los nombres de los clientes y sus números de teléfono, la tienda podría comprar otros datos sobre edad, nivel de ingresos y valoración crediticia, y agregarlos a cada registro. Se puede utilizar la *transformación de datos* y la codificación para reducir la cantidad de datos. Por ejemplo, se podrían agrupar los códigos de producto en audio, vídeo, componentes, dispositivos electrónicos, cámaras, accesorios, etc. Es posible agregar códigos postales a las zonas geográficas, se puede dividir el nivel de ingresos en franjas, etc. En la Figura 28.1, mostraremos un paso llamado *limpieza* como precursor de la creación de los almacenes de datos. Si la minería de datos se basa en un almacén ya existente de esta cadena de tiendas, tendríamos que suponer que ya se ha aplicado la limpieza. Sólo cuando se ha realizado este preproceso, se utilizarán las técnicas de *minería de datos* para aplicar distintas reglas y patrones.

El resultado de la minería podría ser la obtención de los siguientes tipos de *nueva* información:

- **Reglas de asociación**: por ejemplo, siempre que un cliente compre equipos de vídeo, también comprará accesorios electrónicos.
- **Patrones secuenciales**: por ejemplo, supongamos que un cliente compra una cámara de fotos y pasados tres meses compra accesorios de fotografía, entonces el cliente tiene predisposición a comprar accesorios en el plazo de seis meses. Esto define un patrón secuencial de transacciones. Un cliente que

[2] Este comentario se basa principalmente en Adriaans y Zantinge (1996).

compre más de dos veces en períodos de rebajas tendrá bastante probabilidad de comprar al menos una vez en el período navideño.

- **Árboles de clasificación**: por ejemplo, los clientes se pueden clasificar por frecuencia de visitas, tipo de financiación usada, importe de compra o afinidad de tipos de productos; se pueden obtener datos estadísticos reveladores a partir de esas clases.

Podemos observar que existen muchas posibilidades de descubrimiento de nuevos conocimientos sobre qué y cuánto compran los clientes a partir de los patrones de compra y factores relacionados como edad, grupo de nivel de ingresos y lugar de residencia. Esta información se podrá utilizar posteriormente para planificar la instalación de nuevas tiendas en función de datos demográficos, la ejecución de promociones de venta, la unión de varios productos en publicidad o la planificación de estrategias de mercado estacionales. Según se muestra en esta información sobre una tienda, la minería de datos debe estar precedida de una preparación de los datos antes de que pueda generar información útil que pueda influenciar directamente las decisiones del negocio.

Los resultados de la minería de datos se pueden proporcionar en distintos formatos, como listados, gráficos, tablas resumen o visualizaciones.

### 28.1.3 Objetivos de la minería de datos y el descubrimiento de conocimiento

La minería de datos se suele realizar habitualmente con ciertos objetivos o para determinadas aplicaciones finales. En términos generales, estos objetivos se pueden clasificar en los tipos siguientes: predicción, identificación, clasificación y optimización.

- **Predicción.** La minería de datos puede mostrar cómo se comportarán en el futuro determinados atributos de los datos. Entre los ejemplos de predicciones de datos podemos incluir el análisis de transacciones de compra para predecir qué comprarán los consumidores ante determinados descuentos, qué volumen de ventas se generará en una tienda en un periodo determinado y si la eliminación de una línea de productos generará más beneficios. En aplicaciones de este tipo, la lógica de negocio se utiliza unida a la minería de datos. En un contexto científico, determinados patrones de ondas sísmicas podrían predecir un terremoto con una probabilidad alta.

- **Identificación.** Los patrones de datos se pueden utilizar para identificar la existencia de un objeto, un evento o una actividad. Por ejemplo, es posible identificar a los intrusos que intentan introducirse en un sistema mediante los programas que han sido ejecutados, los archivos que han sido accedidos y el tiempo de CPU por sesión. En las aplicaciones de biología, es posible identificar la existencia de un gen a partir de determinadas secuencias de símbolos nucleótidos en la cadena del ADN. El área conocida como *autenticación* es una forma de identificación. Se encarga de asegurar si un usuario es en realidad un usuario en concreto o uno perteneciente a una clase autorizada, e implica una comparación de parámetros, de imágenes o de señales con una base de datos.

- **Clasificación.** La minería de datos puede dividir los datos de forma que sea posible identificar distintas clases o categorías basándose en combinaciones de parámetros. Por ejemplo, los clientes de un supermercado se pueden clasificar en buscadores de descuentos, compradores con poco tiempo, compradores fieles a la tienda, compradores de productos de marca conocida y compradores no frecuentes. Esta clasificación se puede utilizar en diferentes análisis de transacciones de compra de los clientes como actividad post-minería. A veces, la clasificación basada en el conocimiento de dominio común se utiliza como entrada para descomponer el problema de minería y hacerlo más sencillo. Por ejemplo, alimentos saludables, alimentos para fiestas y alimentos para comedores escolares son categorías diferentes en el negocio de los supermercados. Tiene sentido analizar las relaciones entre las categorías y

dentro de ellas como problemas independientes. Se puede usar esta categorización para codificar los datos de forma apropiada antes de someterlos posteriormente a la minería de datos.

- **Optimización.** Un posible objetivo de la minería de datos podría ser la optimización del uso de recursos limitados como el tiempo, el espacio, el dinero o los bienes materiales y la maximización de variables de salida como las ventas o los beneficios obtenidos con determinadas limitaciones.

En realidad, este objetivo de la minería de datos se parece a la función objetivo que se usa en los problemas de investigación operativa para la optimización con restricciones. El término minería de datos se usa habitualmente en un sentido muy amplio. En algunas situaciones, incluye el análisis estadístico y la optimización con restricciones además del aprendizaje automático. No existe una línea clara de separación entre estas disciplinas en la minería de datos. Queda fuera de nuestro ámbito, por tanto, revisar en detalle todo el conjunto de aplicaciones que forman este amplio ámbito de trabajo. Para una comprensión más detallada de este tema, los lectores deberán dirigirse a los libros especializados dedicados a la minería de datos.

### 28.1.4 Tipos de conocimiento adquirido durante la minería de datos

El término *conocimiento* se entiende por lo general como algo que implica cierto grado de inteligencia. Existe una progresión desde los simples datos a la información y después al conocimiento a medida que vamos aplicando más procesamiento. A veces se clasifica al conocimiento como algo inductivo en lugar de algo deductivo. El **conocimiento deductivo** deduce la nueva información basándose en la aplicación de reglas lógicas de deducción pre-especificadas sobre unos datos en concreto. La minería de datos trata con el **conocimiento inductivo**, mediante el cual se descubren nuevas reglas y patrones a partir de los datos suministrados. El conocimiento se puede representar de varias formas. En un sentido no estructurado, se puede representar mediante reglas o lógica proposicional. En forma estructurada, se puede representar mediante árboles de decisión, redes semánticas, redes neuronales o jerarquías de clases o marcos. Es habitual describir el conocimiento adquirido durante la minería de datos según se muestra a continuación:

- **Reglas de asociación.** Estas reglas relacionan la presencia de un conjunto de objetos con otro rango de valores en otro conjunto de variables. Ejemplos: (1) Cuando un cliente de sexo femenino compra un bolso, es probable que compre zapatos. (2) Una imagen de rayos X que contenga ciertas características a y b, contendrá probablemente también la característica c.
- **Jerarquías de clasificación.** El objetivo es trabajar a partir de un conjunto existente de eventos o transacciones para crear una jerarquía de clases. Ejemplos: (1) Se puede dividir una población en cinco rangos de valoración de crédito basándose en el historial de las transacciones de crédito anteriores. (2) Es posible desarrollar un modelo para calcular los factores que determinan la idoneidad de la ubicación de una tienda en una escala de 1 a 10. (3) Los fondos de inversión se pueden clasificar basándose en los datos de resultados utilizando características como crecimiento, ingresos y estabilidad.
- **Patrones secuenciales.** Aquí se busca una secuencia de acciones o eventos. Ejemplo: Si un paciente ha sido sometido a cirugía cardiaca por tener arterias bloqueadas y aneurisma y posteriormente se le detecta un nivel alto de urea en sangre transcurrido un año desde la operación, es posible que sufra de fallos renales en los siguientes 18 meses. La detección de patrones secuenciales es equivalente a la detección de asociaciones entre eventos con determinadas relaciones temporales.
- **Patrones dentro de series temporales.** Es posible detectar parecidos dentro de las posiciones de una **serie temporal** de datos, que es una secuencia de datos tomados a intervalos regulares como, por ejemplo, las ventas diarias o los precios de las acciones al cierre de la bolsa. Ejemplos: (1) Las acciones de una empresa de suministros, ABC Power, y de una compañía financiera, XYZSecurities, han seguido el mismo patrón durante el año 2002 en términos de precio al cierre de la sesión de bolsa. (2) Dos pro-

ductos siguen el mismo patrón de venta durante el verano, pero distinto en el invierno. (3) Un patrón detectado en los vientos solares magnéticos podría utilizarse para predecir cambios en las condiciones atmosféricas de la Tierra.

- **Agrupamiento.** Una población determinada de eventos o elementos puede ser dividida (segmentada) en conjuntos de elementos "similares". Ejemplos: (1) Una población entera de datos de tratamientos para una enfermedad se puede dividir en grupos basándose en el parecido de los efectos secundarios producidos. (2) La población adulta de los Estados Unidos se puede dividir en grupos desde *el que tiene mayor probabilidad de comprar* hasta *el que tiene menor probabilidad de comprar* un nuevo producto. (3) Los accesos a la Web realizados por un grupo de usuarios en un conjunto de documentos (por ejemplo, en una biblioteca digital) pueden ser analizados en términos de palabras clave de documentos para obtener agrupamientos o categorías de usuarios.

En la mayoría de las aplicaciones, el conocimiento deseado es una combinación de los tipos anteriores. Profundizaremos en cada uno de los anteriores tipos de conocimiento en las siguientes secciones.

# 28.2 Reglas de asociación

## 28.2.1 El modelo de la cesta de la compra, soporte y confianza

Una de las principales tecnologías dentro de la minería de datos tiene relación con el descubrimiento de reglas de asociación. La base de datos se ve como un conjunto de transacciones, cada una de ellas relacionada con un conjunto de elementos. Un ejemplo habitual es el de la **cesta de la compra**. En este caso, la cesta de la compra se corresponde con el conjunto de elementos que compra un consumidor en un supermercado durante su visita. Tomemos cuatro transacciones de ese tipo en el ejemplo tomado al azar de la Figura 28.1.

Una **regla de asociación** tiene el aspecto de $X => Y$, donde $X = \{x_1, x_2, ..., x_n\}$, e $Y = \{y_1, y_2, ..., y_m\}$ son conjuntos de elementos, siendo $x_i$ e $y_j$ elementos distintos para todo $i$ y todo $j$. Esta asociación significa que si un cliente compra $X$, también comprará probablemente $Y$. En general, cualquier regla de asociación tiene la forma LHS (término izquierdo) => RHS (término derecho), donde LHS y RHS son conjuntos de elementos. Al conjunto LHS ∪ RHS se le denomina **conjunto de elementos** o ***itemset***, el conjunto de elementos adquiridos por los compradores. Para que una regla de asociación resulte interesante en la minería de datos, la regla debería satisfacer alguna medida de interés. Dos medidas de interés habituales son el soporte y la confianza.

El **soporte** de una regla LHS => RHS se calcula con respecto al conjunto; indica con qué frecuencia aparece un conjunto determinado en la base de datos. Es decir, el soporte es el porcentaje de transacciones que contienen todos los elementos del conjunto, LHS ∪ RHS. Si el soporte es bajo, implica que no hay una evidencia abrumadora de que los elementos en LHS ∪ RHS aparezcan juntos, ya que el conjunto aparece en sólo un pequeño número de transacciones. Otro nombre que se le da al soporte es *cobertura* de la regla.

La **confianza** tiene relación con la implicación mostrada en la regla. La confianza de la regla LHS => RHS se calcula como soporte(LHS ∪ RHS)/soporte(LHS). Podemos considerarlo como la probabilidad de que los elementos de RHS sean comprados en el supuesto de que los elementos de LHS sean comprados por un cliente. A la confianza también se le denomina *fuerza* de la regla.

Como ejemplo de soporte y confianza, tomemos las dos reglas siguientes: leche => zumo y pan => zumo. Echando un vistazo a nuestras cuatro transacciones de ejemplo en la Figura 28.1, veremos que el soporte de {leche, zumo} es 50% y que el soporte de {pan, zumo} es solo 25%. La confianza de leche => zumo es 66,7% (lo cual significa que de cada tres transacciones en las que aparece la leche, en dos de ellas aparece también el zumo) y la confianza de pan => zumo es 50% (significa que en una de cada dos transacciones en las que aparece el pan también aparece el zumo).

Según se puede observar, el soporte y la confianza no van acompañados necesariamente. El objetivo de las reglas de asociación en la minería es, por tanto, generar todas las reglas posibles que superen algún umbral

**Figura 28.1.** Transacciones de ejemplo en el modelo de la cesta de la compra.

| Id_transaccion | Hora | Artículos_comprados |
|---|---|---|
| 101 | 6:35 | leche, pan, galletas, zumo |
| 792 | 7:38 | leche, zumo |
| 1130 | 8:05 | leche, huevos |
| 1735 | 8:40 | pan, galletas, café |

mínimo de soporte y confianza especificado por el usuario. El problema se descompone, según esto, en dos subproblemas.

1. Generación de todos los conjuntos que tienen un soporte que supere el umbral. A estos conjuntos de elementos se les denomina **conjuntos grandes** (o **frecuentes**). Observe que grande significa, en este caso, de soporte grande.
2. Para cada conjunto frecuente, todas las reglas que tienen un valor mínimo de confianza se generan según se muestra a continuación: Para un conjunto frecuente $X$ y siendo $Y \subset X$, tomemos $Z = X - Y$; entonces si soporte ($X$)/soporte ($Z$) > confianza mínima, la regla $Z => Y$ (es decir, $X - Y => Y$) es una regla válida.

La generación de reglas utilizando todos los conjuntos frecuentes y sus soportes es relativamente fácil. Sin embargo, conocer cuáles son todos los conjuntos frecuentes junto con el valor de sus soportes es un serio problema cuando la cardinalidad del conjunto de elementos es muy alta. Un supermercado típico tiene miles de artículos. El número de conjuntos distintos es $2^m$, siendo $m$ el número de artículos, y por tanto la obtención del soporte para todos los posibles conjuntos requiere un gran cálculo computacional. Para reducir el espacio combinatorio de búsqueda, los algoritmos de búsqueda de reglas de asociación utilizan las siguientes propiedades:

Un subconjunto de un conjunto frecuente debe ser también frecuente (es decir, todos los subconjuntos de un conjunto grande superan el mínimo soporte requerido).

De igual modo, un superconjunto de un conjunto infrecuente es también pequeño (lo cual implica que no tiene suficiente soporte).

A la primera propiedad se le denomina **clausura inferior**. La segunda propiedad, llamada propiedad de **no-monotonicidad**, ayuda a reducir el espacio de búsqueda de soluciones posibles. Es decir, una vez que se encuentra un conjunto infrecuente (no grande), entonces cualquier extensión a ese conjunto, formada al añadir uno o más elementos al conjunto, será también un conjunto infrecuente.

## 28.2.2 El Algoritmo Apriori

El primer algoritmo que utilizó las propiedades de clausura inferior y no monotonicidad fue el algoritmo **Apriori**, que se muestra en el Algoritmo 28.1.

**Algoritmo 28.1. Algoritmo Apriori para la búsqueda de conjuntos frecuentes (grandes)**

**Entrada:** Base de datos $D$ con $m$ transacciones y un soporte mínimo, *mins*, representado como fracción de $m$.

**Salida:** Conjuntos frecuentes, $L_1, L_2, ..., L_k$

**Inicio** /* Los pasos o sentencias están numerados para facilitar la lectura */

1. Calcular soporte($i_j$) = count($i_j$)/$m$ para cada elemento individual, $i_1, i_2, ..., i_n$ explorando la base de datos una vez y contando el número de transacciones en las que aparece el elemento $i_j$ (es decir, count($i_j$));
2. El candidato frecuente 1-conjunto, $C_1$, será el conjunto de elementos $i_1, i_2, ..., i_n$.

3. El subconjunto de elementos de $C_1$ que contienen $i_j$ en los que el soporte($i_j$) >= mins pasa a ser el 1-conjunto frecuente, $L_1$;
4. $k = 1$;
   terminado = false;
   **repetir**
1. $L_{k+1} =$ ;
2. Crear el candidato frecuente ($k$+1)-conjunto, $C_{k+1}$, combinando los miembros de $L_k$ que tienen $k$–1 elementos en común; (con esto se forman los *itemsets* candidatos frecuentes ($k$+1) ampliando selectivamente los $k$ conjuntos frecuentes en un elemento).
3. Además, considerar como elementos de $C_{k+1}$ sólo los $k$+1 elementos de forma que cada subconjunto de tamaño $k$ aparezca en $L_k$;
4. Explorar la base de datos una vez y calcular el soporte para cada miembro de $C_{k+1}$; si el soporte para un miembro de $C_{k+1}$ >= mins entonces añadir ese miembro a $L_{k+1}$;
5. Si $L_{k+1}$ está vacío, entonces terminado = true
   si no, $k = k + 1$;
   **hasta terminado;**

**Fin;**

Ilustraremos el Algoritmo 28.1 utilizando los datos de transacción de la Figura 28.1 con un soporte mínimo de 0,5. Los 1-conjuntos candidatos son {leche, pan, zumo, galletas, huevos, café} y sus soportes respectivos son 0,75 0,5 0,5 0,5 0,25 y 0,25. Los primeros cuatro elementos se incluyen en $L_1$ ya que cada soporte es mayor o igual que 0,5. En la primera iteración del bucle repetir, ampliamos los 1-conjuntos frecuentes para crear los 2-conjuntos frecuentes candidatos, $C_2$. $C_2$ contiene {leche, pan}, {leche, zumo}, {pan, zumo}, {leche, galletas}, {pan, galletas} y {zumo, galletas}. Observe que, por ejemplo, {leche, huevos} no aparece en $C_2$ ya que {huevos} es pequeño (de acuerdo con la propiedad de no-monotonicidad) y no aparece en $L_1$. Los soportes de los seis conjuntos incluidos en $C_2$ son 0,25 0,5 0,25 0,25 0,5 y 0,25 y se calculan explorando el conjunto de transacciones. Sólo el segundo 2-conjunto {leche, zumo} y el quinto 2-conjunto {pan, galletas} tienen un soporte mayor o igual que 0,5. Estos dos 2-conjuntos forman los 2-conjuntos frecuentes, $L_2$.

En la siguiente iteración del bucle, construiremos los 3-conjuntos frecuentes añadiendo elementos adicionales a los conjuntos de $L_2$. Sin embargo, en ninguna ampliación de los conjuntos de $L_2$ se cumplirá que todos los subconjuntos de 2 elementos vayan a estar contenidos en $L_2$. Por ejemplo, tomemos {leche, zumo, pan}; el 2-conjunto {leche, pan} no está en $L_2$, por tanto, {leche, zumo, pan} no puede ser un 3-conjunto frecuente debido a la propiedad de clausura inferior. En este punto, el algoritmo finaliza con $L_1$ igual a {{leche}, {pan}, {zumo}, {galletas}} y $L_2$ igual a {{leche, zumo}, {pan, galletas}}.

Se han propuesto otros algoritmos como reglas de asociación para la minería de datos. Se diferencian principalmente en cómo se generan los conjuntos candidatos, y cómo se contabilizan los soportes para los conjuntos candidatos. Algunos algoritmos utilizan estructuras de datos del tipo mapas de bits y árboles *hash* para guardar la información relacionada con los conjuntos. Otros algoritmos utilizan exploraciones múltiples en la base de datos, ya que el número potencial de conjuntos, $2^m$, podría ser demasiado grande para ser almacenado en contadores durante una única exploración. Revisaremos tres algoritmos mejorados (en comparación con el algoritmo Apriori) para la minería mediante reglas de asociación: un algoritmo de muestreo, el algoritmo de árbol de patrón frecuente y el algoritmo de particionado.

## 28.2.3 Algoritmo de muestreo

La idea principal del **algoritmo de muestreo** es seleccionar una pequeña muestra, una que quepa en memoria principal, de la base de datos de transacciones y determinar los conjuntos frecuentes en esa muestra. Si

esos conjuntos frecuentes forman un superconjunto de los conjuntos frecuentes de la base de datos completa, entonces podemos determinar los conjuntos frecuentes reales explorando el resto de la base de datos para calcular los valores exactos de soporte para los conjuntos superconjunto. Por lo general, es posible encontrar un superconjunto de los conjuntos frecuentes a partir de la muestra utilizando, por ejemplo, el algoritmo Apriori con el soporte mínimo más pequeño.

En algunos casos extraños, es posible que falten algunos conjuntos frecuentes y se necesitará una segunda exploración de la base de datos. Para decidir si faltan algunos conjuntos frecuentes se utiliza el concepto de borde negativo. El borde negativo con respecto a un conjunto frecuente $S$ y un conjunto de elementos $l$ son los conjuntos más pequeños contenidos en PowerSet($I$) y no en $S$. La idea básica es que el borde negativo de un conjunto de conjuntos frecuentes contiene los conjuntos más cercanos que podrían ser también frecuentes. Tomemos el caso en el cual un conjunto $X$ no está contenido en los conjuntos frecuentes. Si todos los subconjuntos de X están contenidos en el conjunto de los conjuntos frecuentes, entonces $X$ estaría en el borde negativo.

Ilustraremos esto con el siguiente ejemplo. Tomemos el conjunto de elementos $I$ = {A, B, C, D, E} y supongamos que los conjuntos frecuentes combinados de tamaño 1 a 3 son $S$ = {{A}, {B}, {C}, {D}, {AB}, {AC}, {BC}, {AD}, {CD}, {ABC}}. El borde negativo es {{E}, {BD}, {ACD}}. El conjunto {E} es el único 1-conjunto que no está contenido en $S$, {BD} es el único 2-conjunto que no está en $S$ pero cuyos subconjuntos de 1-conjuntos lo están, y {ACD} es el único 3-conjunto cuyos subconjuntos 2-conjunto están todos en $S$. El borde negativo es importante ya que es necesario para determinar el soporte para aquellos conjuntos que se encuentran en el borde negativo para asegurar que no falta ningún conjunto grande tras analizar los datos de muestra.

El soporte para el borde negativo se determina cuando se explora el resto de la base de datos. Si descubrimos que un conjunto X en el borde negativo se encuentra en el conjunto de todos los conjuntos frecuentes, entonces existe la posibilidad de que un superconjunto de X sea también frecuente. Si sucede esto, se necesita una segunda pasada en la base de datos para asegurarnos de encontrar todos los conjuntos frecuentes.

### 28.2.4 Algoritmo de árbol de patrón frecuente

El **algoritmo de árbol de patrón frecuente** (árbol FP) viene dado por el hecho de que los algoritmos Apriori pueden generar y comprobar un número muy grande de conjuntos candidatos. Por ejemplo, con 1.000 1-conjunto frecuentes, el algoritmo Apriori tendría que generar:

$$\binom{1000}{2}$$

o 499,500 2-conjuntos candidatos. El algoritmo de crecimiento de patrones frecuentes es un modelo que elimina la generación de un gran número de conjuntos candidatos.

El algoritmo genera, en primer lugar, una versión comprimida de la base de datos en términos de árbol de patrón frecuente. El árbol FP almacena información relevante sobre los conjuntos y permite la búsqueda eficiente de conjuntos frecuentes. El proceso actual en la minería de datos adopta una estrategia de "divide y vencerás" en la cual el proceso de minería se descompone en un conjunto de tareas más pequeñas que operan, cada una de ellas, sobre un árbol FP condicional, un subconjunto (proyección) del árbol original. En primer lugar, veremos cómo se construye el árbol FP. Se comienza examinando la base de datos y se calculan los 1-conjuntos frecuentes y sus soportes. Con este algoritmo, el soporte es el *contador* de transacciones que contienen el elemento en lugar de la fracción de transacciones que contienen el elemento. Los 1-conjuntos frecuentes son ordenados a continuación en orden de su soporte no creciente. Posteriormente, se crea la raíz del árbol FP con una etiqueta NULL. La base de datos se explora una segunda vez y para cada transacción $T$ de la base de datos, los 1-conjuntos frecuentes en $T$ se sitúan en el mismo orden que los 1-conjuntos frecuentes. Podríamos decir que esta lista ordenada a partir de $T$ está formada por un primer elemento, la cabecera, y

los elementos restantes, la cola. La información de conjunto (cabecera, cola) se inserta en el árbol FP de forma recursiva, comenzando por el nodo raíz, según se muestra a continuación:

1. Si el nodo actual, *N*, del árbol FP tiene un hijo con nombre de elemento = cabecera, entonces incrementar el contador asociado con el nodo *N* en 1 unidad, en caso contrario crear un nuevo nodo, *N*, con contador igual a 1, enlazar *N* con su padre y enlazar *N* con la tabla de cabeceras de elemento (para recorrer el árbol de forma eficiente).
2. Si la cabeza no está vacía, entonces repetir el paso (1) utilizando como lista ordenada sólo la cola, es decir, la cabecera antigua se elimina y la nueva cabecera será el primer elemento a partir de la cola y los elementos restantes serán la nueva cola.

La tabla de cabeceras de elemento, creada durante el proceso de construcción del árbol FP, contiene tres campos por cada elemento frecuente: el identificador de elemento, el contador de soporte y el enlace a nodo. El identificador de elemento y el contador de soporte son autoexplicativos. El enlace a nodo es un puntero a una ocurrencia de ese elemento en el árbol FP. Ya que pueden aparecer múltiples ocurrencias de un único elemento en el árbol FP, estos elementos se enlazan entre sí en forma de lista en la cual el comienzo de lista está apuntado por el enlace a nodo de la tabla de cabeceras de elemento. Ilustraremos la construcción del árbol FP utilizando los datos de transacción de la Figura 28.1. Utilicemos un soporte mínimo con valor igual a 2. Una pasada por las cuatro transacciones nos da como resultado los siguientes 1-conjuntos frecuentes con su soporte asociado: {{(leche, 3)}, {(pan, 2)}, {(galletas, 2)}, {(zumo, 2)}}. La base de datos se explora una segunda vez y cada transacción será procesada de nuevo.

Para la primera transacción crearemos la lista ordenada $T$ = {leche, pan galletas, zumo}. Los elementos de $T$ son los 1-conjuntos frecuentes en la primera transacción. Los elementos se ordenan basándose en el orden no creciente del contador de los 1-conjuntos encontrados en la primera pasada, (es decir, leche en primer lugar, pan en segundo lugar, etc). Crearemos un nodo raíz NULL para el árbol FP e insertaremos *leche* como hijo de la raíz, *pan* como hijo de *leche*, *galletas* como hijo de *pan* y *zumo* como hijo de *galletas*. Ajustaremos las entradas para los elementos frecuentes en la tabla de cabeceras de elemento.

Para la segunda transacción tenemos la lista ordenada {leche, zumo}. Comenzando en la raíz, vemos que existe un nodo hijo cuya etiqueta es *leche*; por tanto, nos posicionamos en ese nodo y actualizamos su contador (para contabilizar las segundas transacciones que contienen *leche*). Podemos observar que no hay ningún hijo del nodo actual con la etiqueta *zumo*, así que creamos un nuevo nodo con la etiqueta *zumo*. Se ajusta la tabla de cabeceras de elemento.

La tercera transacción sólo tiene un elemento frecuente, {leche}. Una vez más, comenzando por la raíz, vemos que existe el nodo con la etiqueta *leche*, así que nos posicionamos en ese nodo, incrementamos su contador y ajustamos la tabla de cabeceras de elemento. La última transacción contiene elementos frecuentes, {pan, galletas}. En el nodo raíz, vemos que no existe ningún hijo con la etiqueta *pan*. Según esto, crearemos un nuevo hijo de la raíz, inicializamos su contador y, a continuación, insertamos *galletas* como hijo de este nodo e inicializamos su contador. Una vez que hayamos actualizado la tabla de cabeceras de elemento, terminamos obteniendo el árbol FP y la tabla de cabeceras de elemento que se muestra en la Figura 28.2. Si revisamos este árbol FP, veremos que representa fielmente las transacciones originales en formato comprimido (es decir, sólo aparecen los elementos de cada transacción que son 1-conjuntos grandes).

### Algoritmo 28.2. Algoritmo de crecimiento FP para la búsqueda de *itemsets* frecuentes

**Entrada:** árbol FP y soporte mínimo, mins

**Salida:** patrones frecuentes (conjuntos)

procedimiento de crecimiento-FP (árbol, alpha);
**Inicio**
IF árbol contiene un único camino P THEN
para cada combinación, beta, de los nodos en el camino

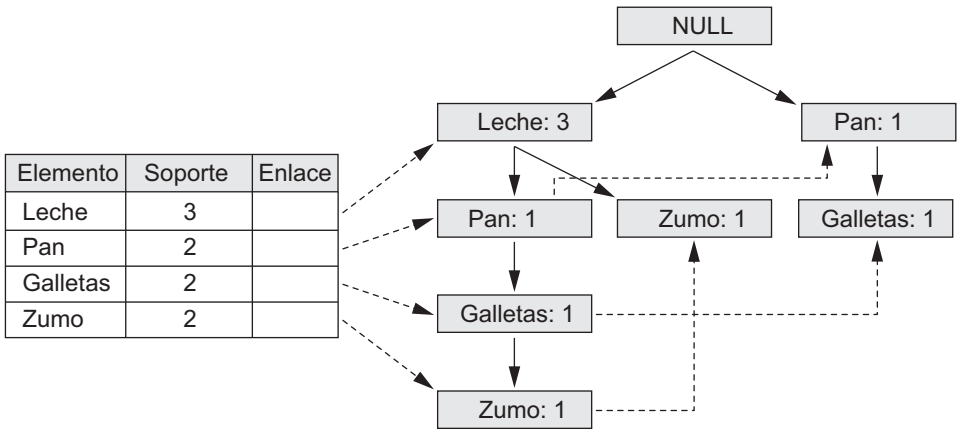
```
        generar patrón (beta ∪ alpha)
        con soporte = mínimo soporte de los nodos en beta
    ELSE
      para cada elemento, i, de la cabecera del árbol hacer
        Inicio
          generar patrón beta = (i ∪ alpha) con soporte = i.soporte;
          construir la base de patrón condicional de beta;
          construir el árbol FP condicional de beta, arbol_beta;
          IF arbol_beta no está vacío THEN
            crecimiento-FP(arbol_beta, beta);
        fin;
Fin;
```

El algoritmo 28.2 se utiliza para explorar el árbol FP en busca de patrones frecuentes. Mediante el árbol FP, es posible encontrar todos los patrones frecuentes que contienen un elemento frecuente dado comenzando por la tabla de cabeceras de elemento para ese elemento y recorriendo los enlaces a nodo en el árbol FP. El algoritmo comienza con un 1-conjunto frecuente (patrón sufijo), construye su base patrón condicional y, a continuación, su árbol FP condicional. La base patrón condicional está formada por un conjunto de caminos prefijo, es decir, en donde el elemento frecuente es un sufijo. Por ejemplo, si tomamos el elemento zumo, observamos en la Figura 28.2 que hay dos caminos en el árbol FP que finalizan en zumo: (leche, pan, galletas, zumo) y (leche, zumo). Los dos caminos prefijo asociados son (leche, pan, galletas) y (leche). El árbol FP condicional se construye a partir de los patrones de la base de patrones condicionales. La minería se ejecuta de forma recursiva sobre este árbol FP. Los patrones frecuentes se forman concatenando el patrón sufijo con los patrones frecuentes obtenidos de un árbol FP condicional.

Ilustraremos el algoritmo utilizando los datos de la Figura 28.1 y el árbol de la Figura 28.2. Al procedimiento de crecimiento FP se le llama con dos parámetros: el árbol FP original y NULL para la variable alpha. Ya que el árbol FP original tiene más de un camino único, ejecutaremos la parte ELSE de la primera sentencia IF. Comenzamos con el elemento frecuente, zumo. Revisaremos los elementos frecuentes siguiendo el orden del soporte más bajo (es decir, desde la última entrada de la tabla hasta la primera). A la variable beta se le asigna zumo con soporte igual a 2.

Siguiendo el enlace a nodo de la tabla de cabeceras de elemento, la base de patrón condicional formada por dos caminos (con zumo como sufijo). Estos son (leche, pan, galletas: 1) y (leche: 1). El árbol FP condicional está formado sólo por un único nodo, leche: 2. Esto se debe a un soporte igual a sólo 1 para el nodo pan y galletas, que está bajo el soporte mínimo igual a 2. Se llama al algoritmo de forma recursiva con un árbol FP

**Figura 28.2.** Árbol FP y tabla de cabeceras de elemento.



formado sólo por un nodo único (es decir, leche: 2) y un valor beta igual a zumo. Ya que este árbol FP sólo tiene un camino, se generarán todas las combinaciones de beta y los nodos del camino (es decir, {leche, zumo}) con soporte igual a 2.

A continuación, se utiliza el elemento frecuente galletas. A la variable beta se le asigna galletas con soporte = 2. Siguiendo el enlace a nodo de la tabla de cabeceras de elemento, construiremos la base de patrón condicional formada por dos caminos. Estos son (leche, pan: 1) y (pan: 1). El árbol FP condicional está formado por un nodo único, pan: 2. Se llama al algoritmo de forma recursiva con un árbol FP formado sólo por un nodo único (es decir, pan: 2) y un valor de beta igual a galletas. Ya que este árbol FP sólo tiene un camino, se generarán todas las combinaciones de beta y los nodos del camino, es decir, {pan, galletas} con soporte igual a 2. Se tomará seguidamente el elemento frecuente, pan. A la variable beta se le asigna pan con soporte = 2. Siguiendo el enlace a nodo de la tabla de cabeceras de elemento, construiremos la base de patrón condicional formada por un camino, que es (leche: 1). El árbol FP condicional está vacío, ya que el contador es menor que el soporte mínimo. Puesto que el árbol FP condicional está vacío, no se generarán patrones frecuentes.

El último elemento frecuente a considerar es leche. Éste es el elemento de más arriba en la tabla de cabeceras de elemento y como tal tiene una base de patrón condicional vacía y un árbol FP condicional vacío. Como resultado de esto, no se añadirán patrones frecuentes. El resultado de la ejecución del algoritmo son los siguientes patrones frecuentes (o conjuntos) con su soporte: {{leche: 3}, {pan: 2}, {galletas: 2}, {zumo: 2}, {leche, zumo: 2}, {pan, galletas: 2}}.

### 28.2.5 Algoritmo de particionado

Otro algoritmo, llamado **algoritmo de particionado**,[3] se resume a continuación. Si tenemos una base de datos con un número pequeño de conjuntos potencialmente grandes, por ejemplo, de unos pocos miles, entonces se puede comprobar el soporte de todos ellos con una sola pasada utilizando una técnica de particionado. El particionado divide la base de datos en subconjuntos que no se solapan entre sí; éstos se consideran de forma individual como bases de datos independientes y todos los conjuntos grandes de cada partición, llamados *conjuntos frecuentes locales*, se generan en una pasada. El algoritmo Apriori podrá ser utilizado posteriormente de forma eficiente sobre cada partición si cabe en su totalidad en memoria principal. Las particiones se eligen de forma que cada partición se pueda colocar en memoria principal. Como consecuencia de esto, una partición se leerá sólo una vez en cada pasada. Lo único a tener en cuenta con el método de particionado es que el soporte mínimo utilizado para cada partición tiene un significado ligeramente distinto al del valor original. El soporte mínimo se basa en el tamaño de la partición en lugar de en el tamaño de la base de datos para determinar los conjuntos frecuentes (grandes) locales. El valor del umbral de soporte real es el mismo que se dio anteriormente, pero el soporte se calcula sólo para una partición.

Al final de la primera pasada, tomaremos la unión de todos los conjuntos frecuentes de cada partición. Esto formará los conjuntos frecuentes candidatos para la totalidad de la base de datos. Cuando se mezclen estas listas, podrán contener algunos falsos positivos. Es decir, es posible que algunos de los conjuntos que son frecuentes (grandes) en una de las particiones no lo sean en algunas de las otras particiones y, por tanto, no superen el soporte mínimo cuando se trate de la base de datos original. Observe que no hay falsos negativos; no faltará ninguno de los conjuntos grandes. Los conjuntos candidatos grandes globales identificados en la primera pasada serán verificados en la segunda pasada, es decir, su soporte real se medirá en la *totalidad* de la base de datos. Al final de la fase dos, todos los conjuntos grandes globales serán identificados. El algoritmo de particionado se adecua a una implementación distribuida o en paralelo para mejorar la eficiencia. Se han propuesto varias mejoras a este algoritmo.[4]

---

[3] Consulte Savasere y otros (1995) si desea más información sobre el algoritmo, las estructuras de datos usadas para implementarlo y sus comparativas de rendimiento.

[4] Consulte Cheung y otros. (1996) y Lin y Dunham (1998).

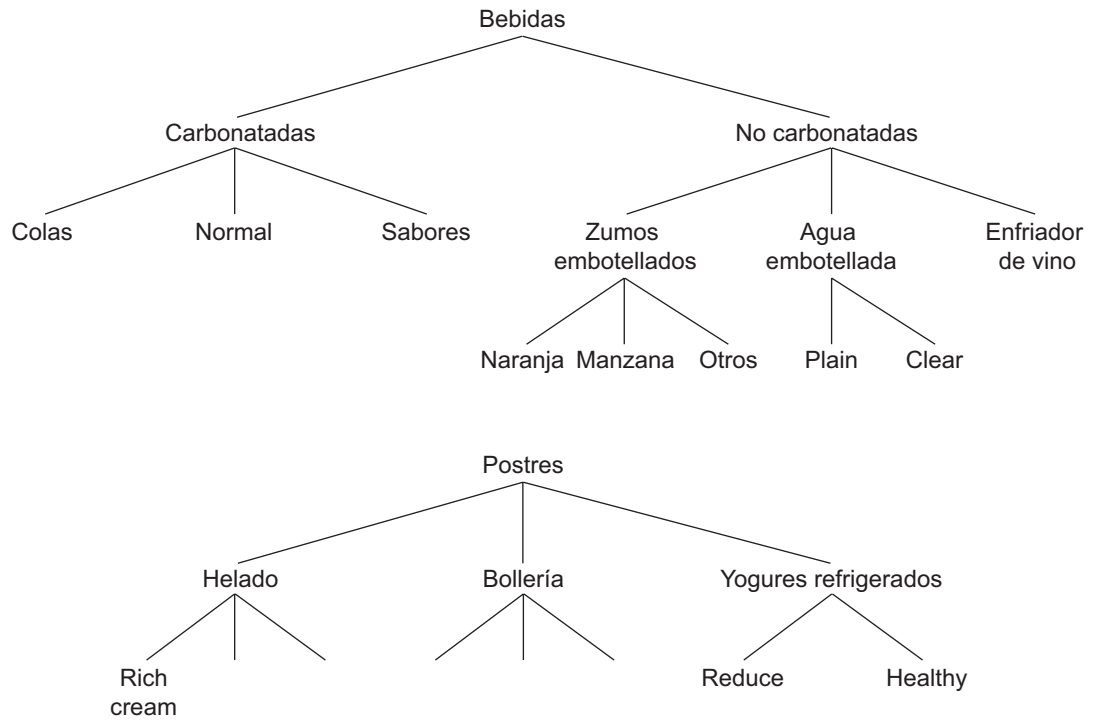
### 28.2.6 Otros tipos de reglas de asociación

**Reglas de asociación entre jerarquías.** Existen determinados tipos de asociaciones que son particularmente interesantes por un motivo especial. Estas asociaciones se producen entre jerarquías de elementos. Por lo general, es posible dividir los elementos entre jerarquías disjuntas basándose en la naturaleza del dominio. Por ejemplo, los alimentos en un supermercado, los artículos en unos grandes almacenes o los artículos en una tienda de deportes se pueden dividir en clases y subclases que den origen a las jerarquías. Observe la Figura 28.3, en la que se muestra la clasificación de elementos en un supermercado. La figura muestra dos jerarquías: bebidas y postres. Los grupos completos podrían no generar asociaciones de la forma bebidas => postres, o postres => bebidas. Sin embargo, las asociaciones del tipo Yogur refrigerado de marca dietética => agua embotellada, o helado de la marca Richcream => enfriador de vinos podrían producir suficiente confianza y soporte para ser reglas de asociación válidas de interés.

Por tanto, si en el área de aplicación existe una clasificación natural de los elementos en jerarquías, la búsqueda de asociaciones *dentro* de las jerarquías no resulta de interés. Las que son de interés específico son las asociaciones *entre* jerarquías. Se pueden producir entre agrupaciones de elementos en diferentes niveles.

**Asociaciones multidimensionales.** El descubrimiento de reglas de asociación implica la búsqueda de patrones en un archivo. Al comienzo de la sección de minería de datos, vimos un ejemplo de un archivo de transacciones de cliente con tres dimensiones: Id_transacción, Hora y Artículos_comprados. Sin embargo, los procesos de minería de datos presentados hasta este momento sólo actúan sobre una dimensión: Artículos_comprados. La siguiente regla es un ejemplo donde incluimos la etiqueta de la única dimensión: Artículos_comprados(leche) => Artículos_comprados(zumo). Podría resultar de interés encontrar reglas de asociación que actúen sobre varias dimensiones, por ejemplo, Hora(6:30 . .8:00) => Artículos_comprados(leche). Las reglas de este tipo se denominan reglas de asociación multidimensional. Las dimensiones representan atributos de registros de un archivo o, en términos de relaciones, las columnas de las filas de una

**Figura 28.3.** Clasificación de los artículos en un supermercado.



relación, y pueden ser categóricos o cuantitativos. Los atributos categóricos tienen un conjunto finito de valores que no muestran relación de orden. Los atributos cuantitativos son numéricos y con valores que muestran una relación de orden, por ejemplo, < Artículos_comprados es un ejemplo de atributo categórico e Id_transacción y Hora son cuantitativos.

Un mecanismo para tratar los atributos cuantitativos es particionar sus valores en intervalos que no se solapen a los cuales se les asigna etiquetas. Esto se puede hacer de una forma estática basándose en el conocimiento específico del dominio. Por ejemplo, una jerarquía conceptual podría agrupar los valores de salario en tres clases distintas: ingresos bajos (0 < Salario < 29.999), ingresos medios (30.000 < Salario < 74.999) e ingresos altos (Salario > 75.000). A partir de esto, se puede utilizar el algoritmo normal de tipo Apriori o una de sus variantes para la regla de minería ya que ahora los atributos cuantitativos tienen el aspecto de atributos categóricos. Otro modelo posible para realizar el particionado es agrupar los valores de atributo basándose en la distribución de los datos, por ejemplo, el método de partición equitativa y la asignación de valores enteros a cada partición. El particionado a este nivel puede estar relativamente afinado, es decir, con un gran número de intervalos. Posteriormente, durante el proceso de minería, se pueden combinar estas particiones con otras particiones adyacentes si su soporte es menor que algún valor máximo predefinido. También se puede utilizar aquí un algoritmo del tipo Apriori para la minería de datos.

**Asociaciones negativas.** El problema de la búsqueda de una asociación negativa es mucho más complicado que en el caso de una asociación positiva. Una asociación negativa es una asociación del tipo siguiente: *el 60% de los clientes que compran patatas fritas no compran agua embotellada*. (Aquí, el 60% se refiere a la confianza en la regla de asociación negativa.) En una base de datos con 10.000 artículos existen $2^{10000}$ combinaciones posibles de artículos, la mayoría de los cuales no aparecen ni una sola vez en la base de datos. Si la ausencia de determinada combinación de artículos va a ser tomada como una asociación negativa, entonces potencialmente tendríamos millones y millones de reglas de asociación negativa con RHSs que no son de interés en absoluto. El problema, entonces, será encontrar sólo las reglas negativas interesantes. En general, lo que nos interesa son los casos en los que dos conjuntos de elementos determinados aparecen con muy poca frecuencia en la misma transacción. Esto provoca dos problemas.

1. Para un inventario total de artículos de 10.000 elementos, la probabilidad de que dos de ellos sean comprados a la vez es $(1/10.000) * (1/10.000) = 10^{-8}$. Si descubrimos que el soporte real cuando esos dos artículos aparecen juntos es 0, no significa una desviación de los resultados esperados y, por tanto, no se trata de una asociación (negativa) interesante.
2. El otro problema es más serio. Estamos buscando combinaciones de artículos con soporte muy bajo y hay millones y millones con soporte bajo o incluso igual a 0. Por ejemplo, a un conjunto de datos de 10 millones de transacciones le faltan la mayoría de los 2.500 millones de combinaciones de pares de 10.000 elementos. Esto generaría miles de millones de reglas inútiles.

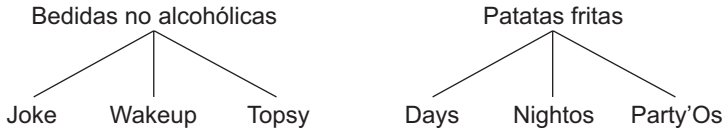
Por tanto, para obtener reglas de asociación negativas de interés, debemos utilizar el conocimiento previo sobre los conjuntos. Una posibilidad es la utilización de jerarquías. Supongamos que utilizamos las jerarquías de bebidas no alcohólicas y patatas fritas que aparecen en la Figura 28.4.

Aparece una fuerte asociación positiva entre las bebidas no alcohólicas y las patatas fritas. Si encontrásemos un soporte grande para el hecho de que cuando los clientes compran patatas Days comprarán preferentemente Topsy y *no* Joke y *tampoco* Wakeup, resultaría interesante ya que normalmente lo que se esperaría es que si existe una fuerte asociación entre Days y Topsy, debería también existir una fuerte asociación entre Days y Joke o entre Days y Wakeup.[5]

En los agrupamientos entre yogur refrigerado y agua embotellada que aparecen en la Figura 28.3 supongamos que la división entre las marcas Reduce y Healthy es 80–20 y que la división entre las marcas Plain y Clear es 60–40 entre las categorías respectivas. Esto nos daría una probabilidad de asociación de que el yogurt refri-

[5] Por simplicidad, suponemos una distribución uniforme de las transacciones entre los miembros de una jerarquía.

**Figura 28.4.** Jerarquía simple de bebidas no alcohólicas y patatas fritas.




gerado fuese comprado junto con el agua embotellada Plain del 48% entre las transacciones que contuviesen yogurt refrigerado y agua embotellada. En el caso de que este soporte resultase ser sólo del 20%, nos indicaría una asociación negativa significativa entre el yogurt Reduce y el agua embotellada Plain; de nuevo, esto resultaría de interés.

El problema de la búsqueda de asociaciones negativas es importante en las situaciones anteriores dado el conocimiento de dominio en forma de jerarquías de generalización de artículos (es decir, las bebidas y postres que se muestran en la Figura 28.3), las asociaciones positivas existentes (como por ejemplo la que existe entre los grupos de yogurt refrigerado y agua embotellada) y la distribución de artículos (como la de los nombres de marcas dentro de los grupos mencionados). El grupo de bases de datos de Georgia Tech ha realizado informes sobre investigaciones en este contexto (consulte la Bibliografía seleccionada de este capítulo). El ámbito de descubrimiento de asociaciones negativas se encuentra limitado por el conocimiento de las jerarquías y la distribución de los artículos. El crecimiento exponencial de las asociaciones negativas es todavía un reto.

### 28.2.7 Consideraciones adicionales en las reglas de asociación

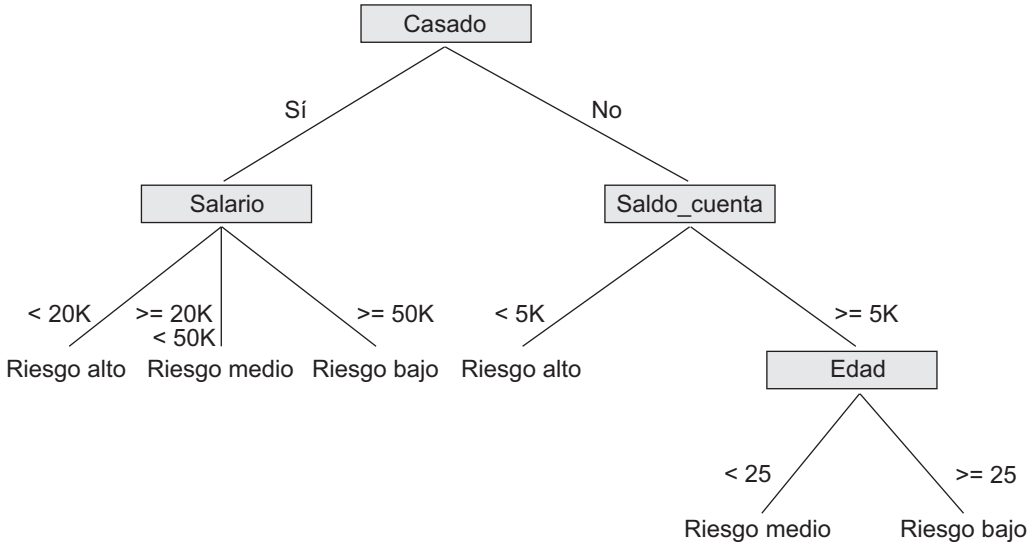
La ejecución de la minería con reglas de asociación en las bases de datos de la vida real resulta complicada debido a los siguientes factores:

- La cardinalidad de los conjuntos de elementos en la mayoría de las situaciones es extremadamente grande, así como el volumen de transacciones. Algunas bases de datos operacionales en el negocio de la distribución y de las comunicaciones recopilan decenas de millones de transacciones al día.
- Las transacciones muestran una gran variación en factores como la ubicación geográfica y la temporalidad, haciendo que el muestreo resulte difícil.
- Las clasificaciones de artículos aparecen en múltiples dimensiones. Según esto, el tratamiento del proceso de búsqueda a partir del conocimiento del dominio, en particular para el caso de las reglas negativas, es extremadamente difícil.
- La calidad de los datos es variable; existen problemas significativos en relación con datos que faltan, datos erróneos, conflictivos y redundantes en muchas de las áreas de negocio.

## 28.3 Clasificación

La clasificación es el proceso de aprendizaje de un modelo que describe diferentes clases de datos. Las clases están predeterminadas. Por ejemplo, en una aplicación bancaria, los clientes que solicitan una tarjeta de crédito se podrían clasificar como de bajo riesgo, riesgo medio y alto riesgo. Debido a lo anterior, a esta actividad se la denomina también **aprendizaje supervisado**. Una vez que se ha construido el modelo se puede utilizar para clasificar nuevos datos. El primer paso de aprendizaje del modelo se lleva a cabo utilizando un conjunto de datos de entrenamiento que ya haya sido clasificado. Cada registro de los datos de entrenamiento contiene un atributo, llamado etiqueta de *clase*, que indica a qué clase pertenece el registro. El modelo resultante tiene por lo general la forma de un árbol de decisión o de un conjunto de reglas. Algunos de los temas importantes en relación con el modelo y con el algoritmo que genera el modelo son la capacidad del

**Figura 28.5.** Ejemplo de árbol de decisión para aplicaciones de tarjeta de crédito.




modelo para predecir la clasificación correcta de los nuevos datos, el coste computacional asociado al algoritmo y la escalabilidad del algoritmo.

Revisaremos el modelo cuando éste tiene la forma de un árbol de decisión. Un árbol de decisión es simplemente una representación gráfica de la descripción de cada una de las clases o, en otras palabras, una representación de las reglas de clasificación. En la Figura 28.5 se muestra un ejemplo de árbol de decisión. Podemos observar en la Figura 28.5 que si un cliente está *casado* y su salario es >= 50K, entonces es de bajo riesgo para la tarjeta de crédito del banco. Ésta es una de las reglas que describen la clase *bajo riesgo*. Al recorrer el árbol desde la raíz hasta cada uno de los nodos hoja, se forman otras reglas para esta clase y las otras dos clases. El Algoritmo 28.3 muestra el procedimiento de construcción de un árbol de decisión a partir de un conjunto de datos de entrenamiento. Inicialmente, todas las muestras de entrenamiento se encuentran en la raíz del árbol. Las muestras se dividen de forma recursiva basándose en los atributos seleccionados. El atributo usado en un nodo para dividir las muestras es aquel con el mejor criterio de división, por ejemplo, aquel que maximiza la medida de ganancia de información.

### Algoritmo 28.3: Algoritmo para generación de un árbol de decisión

**Entrada:** Conjunto de registros de datos de entrenamiento: $R_1$, $R_2$, ..., $R_m$ y conjunto de atributos: $A_1$, $A_2$, ..., $A_n$

**Salida:** Árbol de decisión

procedimiento Construir_árbol (registros, atributos);
**Inicio**

crear un nodo *N*;
IF todos los registros pertenecen a la misma clase, *C* THEN
  retornar *N* como nodo hoja con etiqueta de clase *C*;
IF atributos está vacío THEN
  retornar *N* como nodo hoja con etiqueta de clase *C*, de forma que la mayoría de los registros pertenezcan a él;
seleccionar atributo $A_i$ (*con la mayor ganancia de información*) en los atributos;
etiquetar el nodo *N* con $A_i$;

```
    para cada valor conocido, v_j, de A_i hacer
      inicio
        añadir una rama al nodo N para la condición A_i = v_j;
        S_j = subconjunto de registros donde A_i = v_j;
        IF S_j está vacío THEN
          Añadir una hoja, L, con etiqueta de clase, de forma que la mayoría de los
            registros pertenezcan a ella y devolver L
        ELSE añadir el nodo devuelto por Construir_árbol(S_j, atributos – A_i);
      fin;
  Fin;
```

Antes de ilustrar el algoritmo 28.3, explicaremos la medida de **ganancia de información** con más detalle. El uso de la **entropía** como medida de ganancia de información se debe al objetivo existente de minimizar la información necesaria para clasificar los datos de muestra en las particiones resultantes y, de este modo, minimizar el número esperado de pruebas condicionales para clasificar un nuevo registro. La información esperada para clasificar unos datos de entrenamiento de $s$ muestras, donde el atributo Clase tiene $n$ valores ($v_1$, ..., $v_n$) y $s_i$ es el número de muestras que pertenecen a la etiqueta de clase $v_i$, viene dado por:

$$I(S_1, S_2, \ldots, S_n) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

siendo $p_i$ la probabilidad de que una muestra aleatoria pertenezca a la clase etiquetada como $v_i$. Una estimación para $p_i$ podría ser $s_i/s$. Tomemos un atributo $A$ con valores $\{v_1, ..., v_m\}$ como atributo de prueba para realizar las divisiones en el árbol de decisión. El atributo $A$ divide las muestras en los subconjuntos $S_1$, ..., $S_m$ donde las muestras en cada $S_j$ tienen un valor de $v_j$ para el atributo $A$. Cada uno de los $S_j$ puede contener muestras que pertenezcan a cualquiera de las clases. El número de muestras en $S_j$ que pertenecen a la clase i se puede indicar como $s_{ij}$. La entropía asociada al uso del atributo $A$ como atributo de prueba se define como

$$E(A) = \sum_{j=1}^{m} \frac{S_{1j} + \ldots + S_{nj}}{S} * I\left(S_{1j}, \cdots, S_{nj}\right)$$

$I(s_{1j}, ..., s_{nj})$ se puede definir utilizando la fórmula para $I(s_1, ... s_n)$ y sustituyendo $p_i$ por $p_{ij}$ siendo $p_{ij} = s_{ij}/s_j$. En este momento, la ganancia de información al particionar sobre el atributo $A$, Ganancia($A$), se define como $I(s_1, ..., \mathbf{s}_n) - E(A)$. Podemos utilizar los datos de entrenamiento de muestra de la Figura 28.6 para ilustrar el algoritmo.

El atributo RID representa el identificador de registro usado para identificar un registro individual y se trata de un atributo interno. Lo utilizamos para identificar un atributo en particular en nuestro ejemplo. En primer lugar, calculamos la información esperada necesaria para clasificar los datos de entrenamiento formados por 6 registros como $I(s_1, s_2)$, donde existen dos clases: el valor de la primera etiqueta de clase corresponde a *sí* y el segundo a *no*. Por tanto,

$$I(3,3) = -0{,}5\log_2 0{,}5 - 0{,}5\log_2 0{,}5 = 1.$$

A continuación, calculamos la entropía para cada uno de los cuatro atributos según se muestra a continuación. Para Casado = sí, tenemos $s_{11}=2$, $s_{21}=1$ y $I(s_{11}, s_{21})=0{,}92$. Para Casado = no, tenemos $s_{12}=1$, $s_{22}=2$ y $I(s_{12}, s_{22})=0{,}92$. Por tanto, la información esperada necesaria para clasificar una muestra usando el atributo Casado como atributo de particionado es

$$E(\text{Casado}) = 3/6\ I(s_{11}, s_{21}) + 3/6\ \text{I}(s_{12}, s_{22}) = 0{,}92.$$

La ganancia en información, Ganancia(Casado), sería $1 - 0{,}92 = 0{,}08$. Si seguimos los mismos pasos para calcular la ganancia con respecto a los otros tres atributos obtendremos

**Figura 28.6.** Datos de entrenamiento de ejemplo para el algoritmo de clasificación.

| RID | Casado | Salario | Saldo_cuenta | Edad | Concederprestamo |
|---|---|---|---|---|---|
| 1 | No | >=50K | <5K | >=25 | Sí |
| 2 | Sí | >=50K | >=5K | >=25 | Sí |
| 3 | Sí | 20K. . .50K | <5K | <25 | No |
| 4 | No | <20K | >=5K | <25 | No |
| 5 | No | <20K | <5K | >=25 | No |
| 6 | Sí | 20K. . .50K | >=5K | >=25 | Sí |

$E$(Salario) = 0,33 y Ganancia(Salario) = 0,67

$E$(Saldo_cuenta) = 0,92 y Ganancia(Saldo_cuenta) = 0,08

$E$(Edad) = 0,54 y Ganancia(Edad) = 0,46

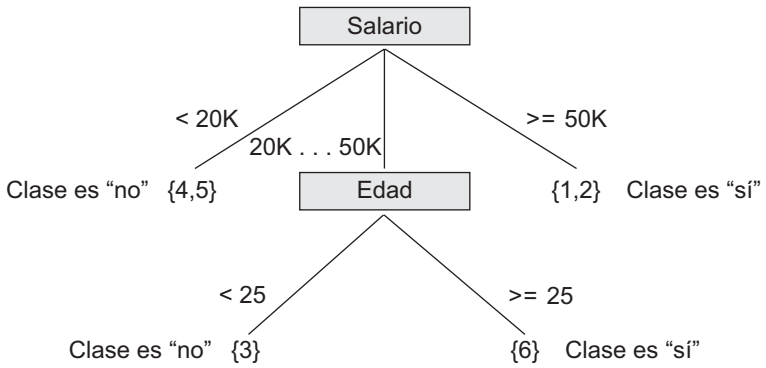
Ya que la ganancia mayor es la del atributo Salario, será el elegido como atributo de particionado. La raíz del árbol se crea con la etiqueta *Salario* y tiene tres ramas, una para cada valor de Salario. Para dos de los tres valores, es decir, <20K y >=50K, todas las muestras que serán divididas de acuerdo con esto (los registros con RID 4 y 5 para <20K y los registros con RID 1 y 2 para >=50K) entrarán en la misma clase *conceder-préstamo no* y *concederpréstamo sí* respectivamente para esos dos valores. Por tanto, crearemos un nodo hoja para cada uno. La única clase que necesita ser expandida es la correspondiente al valor 20K. . .50K con dos muestras, los registros con RID 3 y 6 de los datos de entrenamiento. Al continuar el proceso usando estos dos registros, descubrimos que Ganancia(Casado) es 0, Ganancia(Saldo_cuenta) es 1 y Ganancia(Edad) es 1.

Podemos elegir o bien Edad o bien Sado_cuenta, ya que ambos tienen la ganancia más grande. Tomemos Edad como atributo de particionado. Añadiremos un nodo con la etiqueta Edad y con dos ramas, menor que 25 y mayor o igual que 25. Cada rama divide los restantes datos de ejemplo de forma que cada registro de ejemplo pertenecerá a una rama y, por tanto, a una clase. Se crearán dos nodos hoja y habremos terminado. El árbol de decisión final se muestra en la Figura 28.7.

# 28.4 Agrupamiento

La tarea de minería de datos para la clasificación anterior realiza el particionado de los datos basándose en el uso de una muestra de ejemplo clasificada previamente. Sin embargo, a veces resulta útil particionar los datos sin disponer de una muestra de entrenamiento; a esto también se lo conoce como **aprendizaje no supervisado**. Por ejemplo, en temas de negocio, puede ser importante establecer grupos de clientes que tengan los mismos patrones de compra o, en medicina, puede ser importante establecer grupos de pacientes que muestren las mismas reacciones ante los medicamentos prescritos. El objetivo del agrupamiento es situar los

**Figura 28.7.** Árbol de decisión basado en datos de entrenamiento de ejemplo, donde los nodos hoja se representan mediante un conjunto de RIDs de los registros particionados.



registros en grupos, de forma que los registros de un grupo sean similares a los demás y distintos a los registros de otros grupos. Los grupos son, por lo general, *disjuntos*.

Una faceta importante del agrupamiento es la función de semejanza utilizada. Cuando se trata de datos numéricos, se utiliza habitualmente una función de semejanza basada en la distancia. Por ejemplo, se puede usar la distancia euclídea para medir la semejanza. Tomemos dos puntos de datos *n*-dimensionales (registros) $r_j$ y $r_k$. Podemos asumir que el valor de la i-ésima dimensión es $r_{ji}$ y $r_{ki}$ para los dos registros. La distancia euclídea entre los puntos $r_j$ y $r_k$ en el espacio *n*-dimensional se calcula como:

$$\text{Distancia}(r_j, r_k) = \sqrt{|r_{j1} - r_{k1}|^2 + |r_{j2} - r_{k2}|^2 + \cdots + |r_{jn} - r_{kn}|^2}$$

Cuanto más pequeña sea la distancia entre dos puntos, mayor es la semejanza. Un algoritmo clásico de agrupamiento es el algoritmo de las *k*-medias (*k*-Means), el Algoritmo 28.4.

### Algoritmo 28.4. Algoritmo de agrupamiento *k*-medias

**Entrada:** una base de datos *D*, con *m* registros, $r_1$, ..., $r_m$ y un número deseado de agrupaciones *k*

**Salida:** conjunto de *k* agrupaciones que minimiza el criterio de error cuadrado

**Inicio**
elegir *k* registros al azar como centroides para las *k* agrupaciones;
REPETIR
Asignar cada registro, $r_i$, a una agrupación de forma que la distancia entre $r_i$ y el centroide de la agrupación (media) sea la más pequeña entre las *k* agrupaciones;
recalcular el centroide (media) para cada agrupación basándose en los registros asignados a la agrupación;
HASTA que no haya cambios;
**Fin;**

El algoritmo comienza eligiendo al azar *k* registros para representar los centroides (medias), $m_1$, ..., $m_k$, de las agrupaciones, $C_1$, ..., $C_k$. Todos los registros se colocan en una agrupación en concreto basándose en la distancia entre el registro y la media de la agrupación. Si la distancia entre $m_i$ y el registro $r_j$ es la más pequeña entre todas las medias de las agrupaciones, entonces el registro $r_j$ se sitúa en la agrupación $C_i$. Una vez que todos los registros han sido situados en una agrupación, se recalcula la media de cada agrupación. A continuación, se repite el proceso examinando nuevamente cada registro y situándolo en la agrupación con la media más cercana. Es posible que se necesiten varias iteraciones, pero el algoritmo terminará convergiendo, aunque podría finalizar en un valor óptimo local. La condición de finalización es, por lo general, el criterio de error cuadrado. Para las agrupaciones $C_1$, ..., $C_k$ con medias $m_1$, ..., $m_k$, el error se define como:

$$\text{Error} = \sum_{i=1}^{k} \sum_{\forall r_j \in C_i} \text{Distancia}(r_j, m_i)^2$$

Veremos cómo funciona el Algoritmo 28.4 con los registros bidimensionales de la Figura 28.8. Supongamos que el número deseado de agrupaciones *k* es 2. Dejemos que el algoritmo elija los registros con RID 3 para la agrupación $C_1$ y RID 6 para la agrupación $C_2$ como centroides de la agrupación inicial. Los registros restantes serán asignados a una de esas agrupaciones durante la primera iteración del bucle REPETIR. El registro con RID 1 tiene una distancia a $C_1$ igual a 22,4 y una distancia a $C_2$ igual a 32,0; por tanto, se une a la agrupación $C_1$. El registro con RID 2 tiene una distancia a $C_1$ igual a 10,0 y una distancia a $C_2$ igual a 5,0; por tanto, se une a la agrupación $C_2$. El registro con RID 4 tiene una distancia a $C_1$ igual a 25,5 y una distancia a $C_2$ igual a 36,6; por tanto, se une a la agrupación $C_1$. El registro con RID 5 tiene una distancia a $C_1$ igual a 20,6 y una distancia a $C_2$ igual a 29,2; por tanto, se une a la agrupación $C_1$. En este momento, se calculan las nuevas medias (centroides) de las dos agrupaciones. La media de una agrupación, $C_i$, con *n* registros de *m* dimensiones es el vector:

**Figura 28.8.** Registros bidimensionales de muestra para el ejemplo de agrupamiento (no se tiene en cuenta la columna RID).

| RID | Edad | Años_de_servicio |
|---|---|---|
| 1 | 30 | 5 |
| 2 | 50 | 25 |
| 3 | 50 | 15 |
| 4 | 25 | 5 |
| 5 | 30 | 10 |
| 6 | 55 | 25 |

$$\bar{C}_i = \left( \frac{1}{n} \sum_{\forall r_j \in C_i} r_{ji}, \cdots, \frac{1}{n} \sum_{\forall r_j \in C_i} r_{jm} \right)$$

La nueva media de $C_1$ es (33,75, 8,75) y la nueva media de $C_2$ es (52,5, 25). Se realiza una segunda iteración y se sitúa a los seis registros en las dos agrupaciones según se indica a continuación: los registros con RID 1, 4, 5 se sitúan en $C_1$ y los registros con RID 2, 3, 6 se sitúan en $C_2$. Se recalcula la media de $C_1$ y $C_2$ como (28,3, 6,7) y (51,7, 21,7), respectivamente. En la siguiente iteración, todos los registros se mantienen en sus agrupaciones anteriores y finaliza el algoritmo.

Tradicionalmente, en los algoritmos de agrupamiento se asume que todos los datos caben en la memoria principal. Recientemente, los investigadores han estado desarrollando algoritmos eficientes y escalables para las bases de datos de tamaño grande. Uno de estos algoritmos se llama BIRCH. BIRCH es un modelo híbrido que utiliza un modelo de agrupamiento jerárquico, que construye una representación en árbol de los datos, y varios métodos adicionales de agrupamiento, que se aplican a los nodos hoja del árbol. En el algoritmo BIRCH se utilizan dos parámetros de entrada. Uno de ellos especifica la cantidad de memoria principal disponible y el otro es un umbral inicial como radio de cualquier agrupación. La memoria principal se utiliza para almacenar información descriptiva de la agrupación como el centro (media) de una agrupación y el radio de la agrupación (se supone que las agrupaciones son de forma esférica). El umbral de radio afecta al número de agrupaciones a generar. Por ejemplo, si el valor del umbral de radio es grande, se formarán pocas agrupaciones con muchos registros. El algoritmo intenta mantener un número de agrupaciones de forma que su radio quede por debajo del umbral de radio. Si la memoria disponible resulta insuficiente, se aumentará el umbral de radio.

El algoritmo BIRCH lee los registros de datos de forma secuencial y los inserta en una estructura en árbol en memoria que intenta mantener la estructura de agrupamiento de los datos. Los registros se insertan en los nodos hoja adecuados (agrupaciones potenciales) basándose en la distancia entre el registro y el centro de la agrupación. Podría ser necesario dividir el nodo hoja donde se produce la inserción dependiendo de la actualización del centro y del radio de la agrupación y del parámetro umbral de radio. Adicionalmente, al realizar la división, se almacenará información extra sobre la agrupación, y si la memoria resulta insuficiente, se aumentará el umbral de radio. El incremento del umbral de radio podría producir un efecto colateral de reducción del número de agrupaciones, ya que se podrían mezclar algunos nodos.

En general, BIRCH es un método de agrupamiento eficiente con una complejidad computacional lineal en términos del número de registros a agrupar.

# 28.5 Planteamiento de otras cuestiones en minería de datos

## 28.5.1 Búsqueda de patrones secuenciales

La búsqueda de patrones secuenciales se basa en el concepto de secuencia de conjuntos. Suponemos que las transacciones del tipo de las de la cesta de la compra que vimos con anterioridad están ordenadas por la hora

en que se realizó la compra. Esa ordenación mantiene una secuencia de conjuntos. Por ejemplo, {leche, pan, zumo}, {pan, huevos}, {galletas, leche, café} podría ser una **secuencia de conjuntos** basada en las tres visitas del mismo cliente a la tienda. El soporte de una secuencia $S$ de conjuntos es el porcentaje de un conjunto dado U de secuencias de las cuales $S$ es una subsecuencia. En este ejemplo, {leche, pan, zumo} {pan, huevos} y {pan, huevos} {galletas, leche, café} se consideran **subsecuencias**. El problema de la identificación de patrones secuenciales es, por tanto, la búsqueda de todas las subsecuencias a partir de los conjuntos de secuencias dados que tienen un soporte mínimo definido por el usuario. La secuencia $S_1$, $S_2$, $S_3$, ... es una **predicción** del hecho de que un cliente que compra el artículo $S_1$ comprará con probabilidad el artículo $S_2$, posteriormente el $S_3$, etc. Esta predicción se basa en la frecuencia (soporte) de esta secuencia en el pasado. Se han investigado varios algoritmos de detección de secuencias.

### 28.5.2 Búsqueda de patrones en series temporales

Las series temporales son secuencias de eventos; cada evento puede ser un tipo fijo determinado de transacción. Por ejemplo, el precio de cierre de una acción en bolsa o de un fondo de inversión es un evento que se produce cada día laborable para cada acción o fondo. La secuencia de estos valores por acción o fondo constituye una serie temporal. En una serie temporal, se puede buscar una variedad de patrones analizando secuencias y subsecuencias del modo que se mostró anteriormente. Por ejemplo, podríamos buscar el periodo durante el cual las acciones subieron o se mantuvieron durante $n$ días, o podríamos buscar el cuarto de hora durante el cual las acciones sufrieron una fluctuación de no más del 1% sobre el precio anterior de cierre, o podríamos buscar el cuarto de hora durante el cual las acciones sufrieron el mayor porcentaje de ganancia o de pérdida. Podemos comparar las series temporales con el establecimiento de medidas de semejanza para identificar las compañías cuyas acciones se comportan de forma similar. El análisis y la minería de series temporales es una funcionalidad extendida de la gestión de datos temporales (consulte el Capítulo 24).

### 28.5.3 Regresión

La regresión es una aplicación especial de la regla de clasificación. Si pensamos en una regla de clasificación como una función sobre las variables que relaciona a estas variables con una variable de clase objetivo, la regla tendrá por nombre **regla de regresión**. Se produce una aplicación genérica de la regresión cuando, en lugar de relacionar una tupla de datos de una relación con una clase determinada, se predice el valor de una variable en base a esa tupla. Por ejemplo, tomemos una relación

PRUEBAS_LAB (ID paciente, prueba 1, prueba 2, ..., prueba_n)

que contiene valores que son resultados de una serie de $n$ pruebas a un paciente. La variable objetivo que queremos predecir es $P$, la probabilidad de supervivencia del paciente. Según esto, la regla de regresión toma esta forma:

(prueba 1 en rango$_1$) y (prueba 2 en rango$_2$) y ... (prueba $n$ en rango$_n$) $\Rightarrow P = x$, o $x < P \leqslant y$

La elección depende de si podemos predecir un valor único de $P$ o un rango de valores para $P$. Si pensamos en $P$ como una función:

$P = f$(prueba 1, prueba 2, ..., prueba $n$)

la función se llamará **función de regresión** para predecir $P$. En general, si la función aparece como

$Y = f(X_1, X_2, ..., X_n)$,

y $f$ es lineal en las variables de dominio $x_i$, el proceso de derivar $f$ a partir de un conjunto dado de tuplas para $<X_1, X_2, ..., X_n, y>$ se llama **regresión lineal**. La regresión lineal es una técnica estadística utilizada habitualmente para ajustar un conjunto de observaciones o puntos en $n$ dimensiones a la variable objetivo $y$.

El análisis de regresión es una herramienta muy habitual para el análisis de los datos en muchos ámbitos de investigación. La búsqueda de la función para predecir la variable objetivo es equivalente a la operación de minería de datos.

### 28.5.4 Redes neuronales

Una **red neuronal** es una técnica derivada de la investigación en inteligencia artificial que utiliza la regresión generalizada y proporciona un método iterativo para llevarla a cabo. Las redes neuronales usan un modelo de ajuste de curvas para deducir una función a partir de un conjunto de muestras. Esta técnica proporciona un *modelo de aprendizaje*; funciona mediante una muestra de prueba que se utiliza para la inferencia inicial y el aprendizaje. Con este tipo de método de aprendizaje, es posible interpolar las respuestas a nuevas entradas a partir de las muestras conocidas. Esta interpolación depende, sin embargo, del modelo de conocimiento (representación interna del dominio de problema) desarrollado por el método de aprendizaje.

Se puede hacer una clasificación amplia de las redes neuronales en dos categorías: redes supervisadas y redes no supervisadas. Los métodos adaptativos en los que se intenta reducir el error de salida son métodos de **aprendizaje supervisado**, mientras que los que desarrollan representaciones internas sin generar muestras en salida se denominan métodos de **aprendizaje no supervisado**. Las redes neuronales realizan auto adaptaciones; es decir, aprenden a partir de la información existente sobre un problema determinado. Se ejecutan con efectividad en tareas de clasificación y se usan, por tanto, en la minería de datos. Sin embargo, no están exentas de problemas. Aunque aprenden, no proporcionan una buena representación de *lo que* han aprendido. Sus salidas son muy cuantitativas y difíciles de interpretar. Otra de sus limitaciones es que las representaciones internas desarrolladas por las redes neuronales no son únicas. Además, por lo general, las redes neuronales tienen problemas con el modelado de datos de series temporales. A pesar de estos inconvenientes, son muy populares y varios proveedores comerciales las utilizan con frecuencia.

### 28.5.5 Algoritmos genéticos

Los **algoritmos genéticos** (GA) son un tipo de procedimientos de búsqueda aleatoria aptos para realizar búsquedas sólidas y adaptativas en un amplio rango de topologías de espacios de búsqueda. Fueron modelados de acuerdo con la aparición adaptativa de las especies biológicas mediante mecanismos evolutivos, y fueron introducidos por Holland.[6]

Los GA se han aplicado con éxito en diversos campos como el análisis de imágenes, la planificación y el diseño en ingeniería.

Los algoritmos genéticos amplían la idea de la genética humana del alfabeto de cuatro letras (basado en los nucleótidos A,C,T,G.) del código del ADN humano. La construcción de un algoritmo genético implica el desarrollo de un alfabeto que codifique las soluciones al problema de decisión en términos de cadenas de caracteres de ese alfabeto. Las cadenas son equivalentes a individuos. Una función de ajuste define qué soluciones pueden sobrevivir y cuáles no. Se establece una analogía entre las formas en que se pueden combinar las soluciones y la operación cruzada de cortar y combinar cadenas a partir de un padre y una madre. Se proporciona una población inicial de población variada y se desarrolla el juego de la evolución en el cual se producen mutaciones entre las cadenas. Se combinan para producir una nueva generación de individuos; los individuos más aptos sobreviven y mutan hasta que se desarrolla una familia de soluciones apropiadas.

Las soluciones generadas por los algoritmos genéticos (GA) se distinguen de la mayoría de las otras técnicas por las siguientes características:

- Una búsqueda mediante GA utiliza un conjunto de soluciones durante cada generación en lugar de una solución única.

[6] El importante trabajo de Holland (1975) titulado "Adaptation in Natural and Artificial Systems" introdujo la idea de los algoritmos genéticos.

- La búsqueda en el espacio de cadenas representa una búsqueda en paralelo más grande en el espacio de soluciones codificadas.
- La memoria de la búsqueda realizada se representa únicamente mediante el conjunto de soluciones disponibles para una generación.
- Un algoritmo genético es un algoritmo aleatorio, ya que los algoritmos de búsqueda utilizan operadores de probabilidad.
- Al progresar de una generación a la siguiente, un GA busca el equilibrio casi óptimo entre el descubrimiento de conocimiento y el rendimiento mediante la manipulación de soluciones codificadas.

Los algoritmos genéticos se utilizan para la resolución de problemas y para los problemas de agrupamiento. Su capacidad para la resolución de problemas en paralelo proporciona una potente herramienta para la minería de datos. Entre las desventajas de los GA se encuentran la gran superproducción de soluciones individuales, el carácter aleatorio del proceso de búsqueda y el alto requerimiento de proceso de computación. En general, mediante los algoritmos genéticos es necesario una gran cantidad de proceso de computación para obtener resultados significativos

## 28.6 Aplicaciones de la minería de datos

Las tecnologías de minería de datos se pueden aplicar a una gran variedad de contextos de toma de decisión de negocio. En concreto, las áreas de aplicación de esas tecnologías son algunas de las que aparecen a continuación:

- **Estudios de mercado.** Entre estas aplicaciones se encuentran el análisis de comportamiento de los consumidores basado en los patrones de compra; la determinación de estrategias de mercado incluida la publicidad, la ubicación de tiendas y el envío de correspondencia publicitaria; la segmentación de clientes, tiendas o productos; y el diseño de catálogos, de disposición de tiendas y de campañas publicitarias.
- **Finanzas.** En estas aplicaciones se incluye el análisis de la valoración de crédito de los clientes, la segmentación de deudas en las cuentas, el análisis de resultados de inversiones financieras como acciones de bolsa, bonos y fondos de inversión; la evaluación de opciones financieras; y la detección del fraude.
- **Fabricación.** Aplicaciones relacionadas con la optimización de recursos como máquinas, mano de obra y materiales; el diseño óptimo de procesos de fabricación, la disposición de las plantas de montaje y el diseño de productos, como en el caso de automóviles construidos bajo petición de los clientes.
- **Temas sanitarios.** Entre estas aplicaciones se incluye la búsqueda de patrones en imágenes radiológicas, el análisis de datos experimentales de microarrays (biochips) para realizar agrupamiento de genes y obtener su relación con las enfermedades o con sus síntomas, el análisis de efectos secundarios de las medicinas y la efectividad de determinados tratamientos, la optimización de los procesos en un hospital y la relación entre los datos de bienestar del paciente con la cualificación del médico.

## 28.7 Herramientas comerciales de minería de datos

Actualmente, las herramientas comerciales de minería de datos utilizan varias técnicas comunes para la extracción del conocimiento. Entre estas se incluyen las reglas de asociación, el agrupamiento, las redes neu-

ronales, la secuenciación y el análisis estadístico. Ya hemos revisado estos temas con anterioridad. También se utilizan los árboles de decisión, que son una representación de las reglas utilizadas en la clasificación o en el agrupamiento, y los análisis estadísticos, que pueden incluir la regresión y muchas otras técnicas. Otros productos comerciales utilizan técnicas avanzadas como los algoritmos genéticos, los razonamientos basados en casos, las redes Bayesianas, la regresión no lineal, la optimización combinatoria, el emparejamiento de patrones y la lógica difusa. En este capítulo ya hemos visto algunas de ellas.

La mayoría de las herramientas de minería de datos utilizan la interfaz ODBC (*Open Database Connectivity*, Conectividad de base de datos abierta). ODBC es un estándar de la industria que trabaja con bases de datos; permite el acceso a los datos en la mayoría de los programas de bases de datos más conocidos como Access, dBASE, Informix, Oracle y SQL Server. Algunos de estos paquetes de software proporcionan interfaces a programas de bases de datos específicas; las más comunes son Oracle, Access y SQL Server. La mayoría de las herramientas funcionan en el entorno de Microsoft Windows y unas pocas bajo el sistema operativo UNIX. La tendencia es que todos los productos funcionen bajo el entorno de Microsoft Windows. Una de las herramientas, Data Surveyor, garantiza compatibilidad con ODMG; consulte el Capítulo 21 en el que revisamos el estándar ODMG orientado a objetos. En general, estos programas realizan procesamiento secuencial en una única máquina. Muchos de estos productos trabajan en modo cliente/servidor. Algunos productos incorporan procesamiento en paralelo en arquitecturas de cálculo en paralelo y funcionan como parte de las herramientas de proceso analítico online (OLAP).

### 28.7.1 Interfaz de usuario

La mayoría de las herramientas se ejecutan bajo un entorno de interfaz gráfica de usuario (GUI). Algunos productos incluyen técnicas de visualización sofisticadas para visualizar los datos y las reglas (por ejemplo, MineSet de SGI), e incluso son capaces de manipular los datos estando en este modo de forma interactiva. Las interfaces en modo texto son poco comunes y aparecen habitualmente en herramientas disponibles en UNIX, como Intelligent Miner de IBM.

### 28.7.2 Interfaz de programación de aplicaciones

Normalmente, la interfaz de programación de aplicaciones (API) es una herramienta opcional. La mayoría de los productos no permiten el uso de sus funciones internas. Sin embargo, algunos de ellos permiten que el programador de aplicaciones reutilice su código. Las interfaces más comunes son las librerías C y las librerías de enlace dinámico (DLLs). Algunas herramientas incluyen lenguajes propietarios de comandos de base de datos.

En la Tabla 28.1 se muestra una lista de las herramientas de minería de datos más representativas. A fecha de hoy, existen casi cien productos comerciales disponibles a nivel mundial. Entre los productos de fuera de los EEUU se encuentran Data Surveyor de los Países Bajos y Polyanalyst de Rusia.

### 28.7.3 Tendencias para el futuro

Las herramientas de minería de datos se encuentran en constante evolución a partir de las ideas procedentes de las últimas investigaciones científicas. Muchas de estas herramientas incorporan los últimos algoritmos tomados de la inteligencia artificial (AI), la estadística y la optimización.

En este momento, el procesamiento rápido se realiza mediante técnicas modernas de bases de datos (como el procesamiento distribuido) en arquitecturas cliente/servidor, en bases de datos en paralelo y en almacenes de datos. En el futuro, la tendencia se dirige hacia el desarrollo de funcionalidades en Internet más perfeccionadas. Adicionalmente, los modelos híbridos se convertirán en algo común y el procesamiento se realizará utilizando todos los recursos disponibles. El procesamiento sacará provecho de los entornos de computación distribuidos y en paralelo. Este movimiento es especialmente importante ya que las bases de datos modernas

contienen cantidades de información muy grandes. No sólo aumenta el número de bases de datos multimedia, sino que además el almacenamiento y la recuperación de imágenes son operaciones lentas. Por otra parte, el coste del almacenamiento secundario está decreciendo; por tanto, resultará factible el almacenamiento masivo de información incluso para las pequeñas compañías. De acuerdo con esto, los programas de minería de datos tendrán que tratar conjuntos de datos cada vez más grandes y cada vez de más compañías.

En un futuro cercano, parece que Microsoft Windows NT y UNIX serán las plataformas estándar, siendo NT la dominante. La mayoría del software de minería de datos utilizará el estándar ODBC para extraer datos de las bases de datos de negocio; se espera que desaparezcan los formatos propietarios de datos de entrada. Existe

**Tabla 28.1.** Algunas herramientas representativas de minería de datos.

| Compañía | Producto | Técnica | Plataforma | Interfaz * |
|---|---|---|---|---|
| Acknosoft | Kate | Árboles de decisión, razonamiento basado en casos | Win NT UNIX | Microsoft Access |
| Angoss | Knowledge Seeker | Árboles de decisión, estadísticas | Win NT | ODBC |
| Business Objects | Business Miner | Redes neuronales, aprendizaje artificial | Win NT | ODBC |
| CrossZ | QueryObject | Análisis estadístico, algoritmo de optimización | Win NT MVS UNIX | ODBC |
| Data Distilleries | Data Surveyor | Muy general, puede convivir con diferentes tipos de minería de datos | UNIX | ODBC Compatible ODMG |
| DBMiner Technology Inc. | DBMiner | Análisis OLAP, asociaciones, clasificación, algoritmos de agrupamiento | Win NT | Microsoft 7.0 OLAP |
| IBM | Intelligent Miner | Clasificación, reglas de asociación, modelos predictivos | UNIX (AIX) | IBM DB2 |
| Megaputer Intelligence | Polyanalyst | Descubrimiento de conocimiento simbólico, programación evolutiva | Win NT OS/2 | ODBC Oracle DB2 |
| NCR | Management Discovery Tool (MDT) | Reglas de asociación | Win NT | ODBC |
| SAS | Enterprise Miner | Árboles de decisión, reglas de asociación, redes neuronales, regresión, agrupamiento | UNIX (Solaris) Win NT Macintosh | ODBC Oracle AS/400 |
| Purple Insight | MineSet | Árboles de decisión, reglas de asociación | UNIX (Irix) | Oracle Sybase Informix |

*ODBC: Conectividad de base de datos abierta

ODMG: *Object Data Management Group*

una clara necesidad de incluir datos no estándar, como imágenes y otros datos multimedia, como datos de entrada a la minería de datos. Sin embargo, el desarrollo de algoritmos para la minería de datos no estándar aún no ha alcanzado el nivel de madurez suficiente para su comercialización.

# 28.8 Resumen

En este capítulo revisamos la minería de datos; una disciplina de gran relevancia en la cual se utiliza la tecnología de bases de datos para obtener conocimiento adicional o patrones a partir de los datos. Mostramos un ejemplo ilustrativo de búsqueda de conocimiento en bases de datos; un tema más amplio que la minería de datos. En lo relativo a la minería de datos, entre las diferentes técnicas, nos fijamos especialmente en los detalles de la minería mediante reglas de asociación, la clasificación y el agrupamiento. Mostramos algoritmos en cada una de estas áreas e ilustramos con ejemplos su modo de funcionamiento.

También se revisaron brevemente otras técnicas, entre las cuales se incluyen las redes neuronales basadas en la inteligencia artificial y los algoritmos genéticos. Se sigue realizando investigación sobre minería de datos y, por tanto, hicimos una descripción de hacia dónde se dirigen estas investigaciones. Resumimos 11 de las casi cien herramientas de minería de datos disponibles hoy en día; se espera que las investigaciones futuras amplíen su número y funcionalidad de forma significativa.

## Preguntas de repaso

**28.1.** ¿Cuáles son las diferentes fases de la búsqueda de conocimiento en las bases de datos? Describa un escenario de aplicación completo en el que se pueda obtener conocimiento a partir de las transacciones de una base de datos existente.

**28.2.** ¿Cuáles son los objetivos o tareas que pretende facilitar la minería de datos?

**28.3.** ¿Cuáles son los cinco tipos de conocimiento obtenidos a partir de la minería de datos?

**28.4.** ¿Qué son las reglas de asociación como tipo de conocimiento? Dé una definición de soporte y confianza y utilícelas para definir una regla de asociación.

**28.5.** ¿Qué es la propiedad de clausura inferior? ¿En qué forma ayuda al desarrollo de un algoritmo eficiente para la búsqueda de reglas de asociación, es decir, en relación con la búsqueda de conjuntos grandes?

**28.6.** ¿Cuál es el factor que motivó el desarrollo del algoritmo de árbol-FP para la minería de reglas de asociación?

**28.7.** Describa con un ejemplo una regla de asociación entre jerarquías.

**28.8.** ¿Qué es una regla de asociación negativa en el contexto de la jerarquía de la Figura 28.3?

**28.9.** ¿Cuáles son las dificultades de obtener reglas de asociación en bases de datos de gran tamaño?

**28.10.** ¿Cuáles son las reglas de clasificación y cómo se relacionan con los árboles de decisión?

**28.11.** ¿Qué es la entropía y cómo se usa en la construcción de árboles de decisión?

**28.12.** ¿En qué se diferencia el agrupamiento de la clasificación?

**28.13.** Describa las redes neuronales y los algoritmos genéticos como técnicas para la minería de datos. ¿Cuáles son las mayores dificultades al usar estas técnicas?

## Ejercicios

**28.14.** Aplique el algoritmo Apriori al siguiente conjunto de datos.

| Id_trans | Artículos_comprados |
|---|---|
| 101 | leche, pan, huevos |
| 102 | leche, zumo |

| | |
|---|---|
| 103 | zumo, mantequilla |
| 104 | leche, pan, huevos |
| 105 | café, huevos |
| 106 | café |
| 107 | café, zumo |
| 108 | leche, pan, galletas, huevos |
| 109 | galletas, mantequilla |
| 110 | leche, pan |

El conjunto de artículos es {leche, pan, galletas, huevos, mantequilla, café, zumo}. Utilice 0,2 como valor mínimo de soporte.

**28.15.** Indique dos reglas con confianza igual a 0,7 o mayor para un conjunto que contenga tres artículos en el Ejercicio 28.14.

**28.16.** Para el algoritmo de particionado, demuestre que cualquier conjunto frecuente en la base de datos debe aparecer como conjunto frecuente local en al menos una partición.

**28.17.** Indique el árbol FP que se obtendría a partir de los datos del Ejercicio 28.14.

**28.18.** Aplique el algoritmo de crecimiento FP al árbol FP del Ejercicio 28.17 e indique los conjuntos frecuentes.

**28.19.** Aplique el algoritmo de clasificación al siguiente conjunto de registros de datos. El atributo de clase es Repetir_cliente.

| RID | Edad | Ciudad | Sexo | Estudios | Repetir_cliente |
|---|---|---|---|---|---|
| 101 | 20...30 | NY | F | Licenciado | SÍ |
| 102 | 20...30 | SF | M | Diplomado | SÍ |
| 103 | 31...40 | NY | F | Licenciado | SÍ |
| 104 | 51...60 | NY | F | Licenciado | NO |
| 105 | 31...40 | LA | M | Bachiller | NO |
| 106 | 41...50 | NY | F | Licenciado | SÍ |
| 107 | 41...50 | NY | F | Diplomado | SÍ |
| 108 | 20...30 | LA | M | Licenciado | SÍ |
| 109 | 20...30 | NY | F | Bachiller | NO |
| 110 | 20...30 | NY | F | Licenciado | SÍ |

**28.20.** Tome el siguiente conjunto de registros de dos dimensiones:

| RID | Dimensión1 | Dimensión2 |
|---|---|---|
| 1 | 8 | 4 |
| 2 | 5 | 4 |
| 3 | 2 | 4 |
| 4 | 2 | 6 |
| 5 | 2 | 8 |
| 6 | 8 | 6 |

Tome también dos esquemas de agrupamiento diferentes: (1) aquel en el que $\text{Agrupamiento}_1$ contiene los registros {1,2,3} y $\text{Agrupamiento}_2$ contiene los registros {4,5,6} y (2) en el que

Agrupamiento $_1$ contiene los registros {1,6} y Agrupamiento$_2$ contiene los registros {2,3,4,5}. ¿Qué esquema es el mejor y por qué?

**28.21.** Utilice el algoritmo de *k*-medias para agrupar los datos del Ejercicio 28.20. Podemos usar un valor de 3 para *K* y suponer que los registros con RIDs 1, 3 y 5 se usan para los centroides (medias) de agrupamiento iniciales.

**28.22.** El algoritmo de *k*-medias utiliza una métrica de parecidos para medir la distancia entre un registro y la media de una agrupación. Si los atributos de los registros no son cuantitativos, sino categóricos por naturaleza, en uno del tipo Nivel_de_ingresos con valores {bajo, medio, alto} o Casado con valores {Sí, No} o Estado de residencia con valores {Alabama, Alaska, ..., Wyoming} la métrica de distancia no será significativa. Defina una métrica de parecidos más adecuada que pueda ser utilizada para el agrupamiento de registros de datos que contengan datos categóricos.

## Bibliografía seleccionada

La documentación sobre minería de datos procede de varios campos entre los que se incluye la estadística, la optimización matemática y la inteligencia artificial. La minería de datos ha empezado a aparecer muy recientemente en la literatura de bases de datos. Por tanto, sólo mencionaremos unos cuantos trabajos en relación con las bases de datos. Chen y otros (1996) hace un buen resumen sobre la minería de datos desde la perspectiva de las bases de datos. Han y Kamber (2001) es un texto excelente, en el que se describe en detalle los distintos algoritmos y técnicas utilizadas en el área de la minería de datos. El trabajo realizado en el centro de investigación de IBM en Almaden ha generado un gran número de conceptos básicos y de algoritmos, así como resultados a partir de varios estudios de rendimiento. Agrawal y otros (1993) informa sobre el primer estudio a gran escala sobre reglas de asociación. El algoritmo Apriori para datos de la cesta de la compra de Agrawal y Srikant (1994) fue mejorado utilizando el particionado de Savasere y otros (1995); Toivonen (1996) propone el muestreo como forma de reducir el trabajo de procesamiento. Cheung y otros (1996) amplía el particionado a los entornos distribuidos; Lin y Dunham (1998) propone técnicas para resolver los problemas de distorsión de datos. Agrawal y otros (1993b) revisa la perspectiva de ejecución de reglas de asociación. Mannila y otros (1994), Park y otros (1995) y Amir y otros (1997) muestran algoritmos adicionales eficientes relacionados con las reglas de asociación. Han y otros (2000) muestra el algoritmo de árbol FP visto en este capítulo. Srikant (1995) propone reglas generalizadas de minería. Savasere y otros (1998) muestra el primer modelo para realizar la minería en asociaciones negativas. Agrawal y otros (1996) describe el sistema Quest de IBM. Sarawagi y otros (1998) describe una implementación donde las reglas de asociación se encuentran integradas con un sistema de gestión de bases de datos relacionales. Piatesky-Shapiro y Frawley (1992) contribuyó con documentos sobre un amplio abanico de temas relacionados con la búsqueda de conocimiento. Zhang y otros (1996) presenta el algoritmo BIRCH para el agrupamiento de bases de datos de tamaño grande. La información sobre aprendizaje mediante árboles de decisión y sobre el algoritmo de clasificación que aparece en este capítulo se puede encontrar en Mitchell (1997).

Adriaans y Zantinge (1996), Fayyad y otros (1997) y Weiss y Indurkhya (1998) son libros dedicados a los distintos aspectos de la minería de datos y su uso en la predicción. La idea de algoritmos genéticos fue propuesta por Holland (1975); una buena revisión sobre los algoritmos genéticos aparece en Srinivas y Patnaik (1974). Las redes neuronales disponen de una amplia documentación; una completa introducción se encuentra disponible en Lippman (1987).

Un libro reciente, Tan y otros (2006), proporciona una completa introducción a la minería de datos y proporciona un conjunto detallado de referencias. También se aconseja a los lectores que consulten las actas de dos importantes conferencias anuales sobre minería de datos: la Knowledge Discovery and Data Mining Conference (KDD), que se celebra desde 1995, y la SIAM International Conference on Data Mining (SDM), que se celebra desde 2001. Puede encontrar enlaces a conferencias anteriores en dblp.uni-trier.de.

CAPÍTULO 29

# Visión general del almacenamiento de datos y OLAP

El aumento de la potencia de procesamiento y la sofisticación de las herramientas y técnicas analíticas ha dado como resultado lo que se ha dado en llamar almacenes de datos (*data warehouses*), los cuales proporcionan el almacenamiento, la funcionalidad y la respuesta adecuada a las consultas que se escapan del ámbito de las bases de datos orientadas a las transacciones. Acompañando a este aumento de la potencia siempre aparece una importante demanda para la mejora del rendimiento del acceso a la información de las bases de datos. Como ya hemos visto a lo largo de este libro, las bases de datos tradicionales buscan el equilibrio entre los requerimientos de acceso a los datos y la necesidad de garantizar la integridad de los mismos. En las organizaciones de hoy en día, los usuarios de los datos suelen encontrarse alejados de las fuentes de los mismos. Muchas personas sólo precisan leer la información, pero conservando una adecuada velocidad de acceso a grandes volúmenes de datos para ser descargados al computador. Con frecuencia, estos datos proceden de múltiples bases de datos. Debido a que muchos de los análisis que se realizan son cíclicos y predecibles, los distribuidores de software y el personal de soporte están diseñando sistemas que soporten estas funciones. En la actualidad, los directivos de nivel medio con poder para tomar decisiones necesitan obtener información a un nivel de detalle tal que les facilite la toma de decisiones. El *almacenamiento de* datos, OLAP (*Online Analytical Processing, Procesamiento analítico en línea*) y la *minería de datos* ofrecen esta funcionalidad. El Capítulo 28 ofreció una introducción a los conceptos que están detrás de esta técnica. En este capítulo mostraremos una panorámica más amplia de las tecnologías de almacenamiento de datos y OLAP.

## 29.1 Introducción, definiciones y terminología

En el Capítulo 1 definimos una base de datos como una colección de datos relacionados, y un sistema de base de datos como la unión de una base de datos y su software de gestión. Un almacén de datos también es una colección de información junto con un sistema de soporte. Sin embargo, existe una clara distinción entre ambos. Las bases de datos tradicionales son transaccionales (relacional, orientada a objetos, red o jerárquica). Los almacenes de datos tienen la característica distintiva de que están proyectados principalmente para las aplicaciones de toma de decisiones. Están optimizados para la recuperación de datos, no para el procesamiento de transacciones rutinarias.

Ya que los almacenes de datos se han desarrollado en numerosas organizaciones para cumplir con sus necesidades particulares, no existe una definición única y estricta del término. La prensa y los libros especializados en la materia han elaborado el concepto de muchas formas, los distribuidores lo han acaparado para

ayudar en la venta de los productos relacionados y los consultores ofrecen una gran variedad de servicios bajo la bandera del almacenamiento de datos (*data warehousing*). Sin embargo, los almacenes de datos se diferencian de las bases de datos tradicionales en su estructura, funcionamiento y objetivo.

W. H. Inmon[1] tipificó un **almacén de datos** como una *colección de datos orientada al sujeto, integrada, no volátil y de tiempo variable para el soporte de las decisiones de los directivos*. Los almacenes de datos proporcionan acceso a los datos para realizar análisis complejos, descubrimiento de conocimiento y toma de decisiones. Soportan peticiones de alto rendimiento sobre los datos y la información de una organización. Existen varios tipos de aplicaciones (OLAP, DSS y minería de datos), los cuales definiremos a continuación.

**OLAP (Procesamiento analítico en línea)** es un término usado para describir el análisis de datos complejos desde el almacén de datos. En el lado de los técnicos expertos en conocimiento, las herramientas OLAP utilizan para los análisis capacidades de computación distribuidas que requieren una potencia de almacenamiento y procesamiento que pueden estar localizadas de forma económica y eficiente en un computador de sobremesa.

Los **DSS (Sistemas de soporte a las decisiones, *Decision-Support Systems*)**, conocido también como **EIS** (Sistemas de información ejecutiva, *Executive Information Systems*) (no debe confundirse con los sistemas de integración empresariales), dan soporte a las personas que toman las decisiones en una organización ofreciéndoles datos de más alto nivel para ayudarles en las decisiones importantes y complejas. La minería de datos (que se trató con detalle en el Capítulo 28) se emplea para el descubrimiento del conocimiento, el proceso de búsqueda de datos para los nuevos conocimientos imprevistos.

Las bases de datos tradicionales soportan **OLTP (Procesamiento de transacciones en línea, *OnLine Transaction Processing*)**, que incluye inserciones, actualizaciones y borrados además de las consultas para la recuperación de información. Los sistemas de relaciones tradicionales están optimizados para procesar consultas que toquen sólo una pequeña parte de la base de datos y transacciones que se ocupen de actualizar o insertar algunas tuplas por relación. Por ello, no están preparadas para gestionar sistemas OLAP, DSS o la minería de datos. Por el contrario, los almacenes de datos están diseñados precisamente para ofrecer una extracción, un procesamiento y una presentación eficientes con vistas a los análisis y la toma de decisiones. En comparación con las bases de datos tradicionales, los almacenes de datos suelen contener enormes cantidades de datos procedentes de diversas fuentes, entre las que pueden incluirse bases de datos de distintos modelos y, en ocasiones, ficheros obtenidos de sistemas y plataformas independientes.

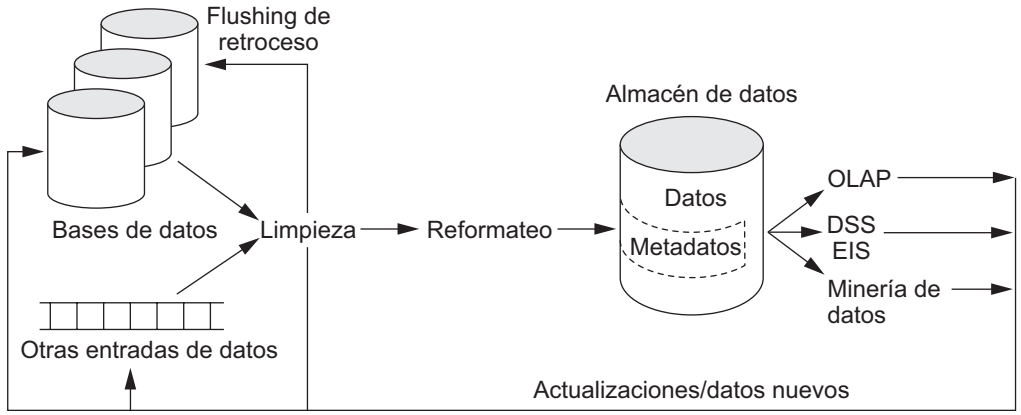
## 29.2 Características de los almacenes de datos

Para hablar de los almacenes de datos y diferenciarlos de las bases de datos tradicionales necesitamos un modelo de datos adecuado. El modelo multidimensional (explicado con detalle en la Sección 29.3) es un buen comienzo para OLAP y las tecnologías de soporte a las decisiones. A diferencia de las bases de datos múltiples, las cuales ofrecen acceso a bases de datos heterogéneas, un almacén de datos es una "agrupación" de datos integrados procedentes de varias fuentes, procesados para su almacenamiento en un modelo multidimensional. A diferencia de la mayoría de bases de datos tradicionales, los almacenes de datos suelen soportar análisis de tendencia y series de tiempo, los cuales requieren de datos de una antigüedad que no suele mantenerse en los sistemas transaccionales.

Comparados con las bases de datos transaccionales, los almacenes de datos no son volátiles. Esto significa que la información que contienen cambia con menos frecuencia y debe ser considerada de tipo "no en tiempo real" con actualizaciones periódicas. En sistemas transaccionales, las transacciones son la unidad y el agente de cambio de la base de datos; por contra, la información de los almacenes de datos es mucho más tosca y se refresca siguiendo una cuidadosa política de refresco, habitualmente de tipo incremental. Las actualiza-

[1] Inmon (1992) acreditó el uso del término almacén por primera vez.

**Figura 29.1.** Ejemplo de transacciones en un modelo de cesta de compra.




ciones del almacén son manipuladas por el componente de adquisición del mismo, el cual proporciona todas las operaciones de preprocesamiento necesarias.

De forma más general podemos definir el almacenamiento de datos como *una colección de tecnologías de soporte a las decisiones, que tiene como objetivo que el trabajador del conocimiento (ejecutivo, director, analista) tome decisiones mejores y más rápidas*.[2] La Figura 29.1 ofrece una panorámica de la estructura conceptual de un almacén de datos. Muestra el proceso completo de almacenamiento de datos, incluyendo la limpieza y el reformateo de los datos antes de su carga en el almacén. Este proceso está manipulado en la actualidad por herramientas conocidas como ETL (Extracción, transformación y carga; *Extraction, Transformation and Loading*). Como respaldo del proceso, OLAP, la minería de datos y el DSS pueden generar nueva información relevante como las reglas; esta información se muestra en la figura volviendo al almacén. La figura muestra también que las fuentes de datos pueden incluir ficheros.

Los almacenes de datos tienen las siguientes características distintivas:[3]

- Vista conceptual multidimensional.
- Dimensionalidad genérica.
- Dimensiones ilimitadas y niveles de agregación.
- Operaciones dimensionales cruzadas sin restricciones.
- Manipulación dinámica de matriz escasa.
- Arquitectura cliente/servidor.
- Soporte multiusuario.
- Accesibilidad.
- Transparencia.
- Manipulación intuitiva de datos.
- Realización de informes consistentes.
- Informes flexibles.

[2] Chaudhuri y Dayal (1997) ofrece un excelente tutorial sobre el tema usando esta definición como punto de arranque.

[3] Codd (1993) acuñó el término OLAP y mencionó estas características. Hemos reordenado la lista original de Codd.

Debido a que engloban grandes volúmenes de información, los almacenes de datos suelen estar en un orden de magnitud (y a veces en dos) mayor que las bases de datos origen. El volumen total de los datos (medido probablemente en terabytes) es un problema que tiene que ver con almacenes de datos a nivel de empresa, almacenes virtuales y mercados de datos:

- Los **almacenes de datos a nivel de empresa** son enormes proyectos que precisan de una enorme inversión de tiempo y recursos.
- Los **almacenes de datos virtuales** ofrecen vistas de las bases de datos operacionales que están materializadas para obtener un acceso más eficiente.
- Los **mercados de datos** suelen estar dirigidos a un subconjunto de una organización, como un departamento, y están mucho más enfocados.

# 29.3 Modelado de datos para los almacenes

Los modelos multidimensionales se benefician de las relaciones inherentes de los datos para rellenar unas matrices multidimensionales llamadas cubos de datos (o hipercubos en el caso de que tengan más de tres dimensiones). Para aquellos datos que se prestan por sí mismos al formateo dimensional, el rendimiento de la consulta en las matrices multidimensionales puede ser mucho mejor que en el modelo de datos relacional. Los periodos fiscales de una empresa, sus productos y sus zonas geográficas son tres ejemplos de dimensiones en un almacén de datos.

Una hoja de cálculo estándar es una matriz de dos dimensiones. Un ejemplo de ello podría ser una hoja de cálculo con las ventas regionales de un producto durante un periodo de tiempo determinado. Los productos podrían ser las filas, mientras que las ventas de cada región comprenderían las columnas (la Figura 29.2 muestra esta organización bidimensional). Si se incorpora una dimensión de tiempo, como los trimestres fiscales de la empresa, se obtendría una matriz tridimensional, la cual estaría representada por un cubo de datos.

La Figura 29.3 muestra un cubo de datos tridimensional que organiza los datos de venta de un producto por periodos fiscales y regiones de venta. Cada celda podría contener datos de un producto, periodo fiscal y región específicos. Incluyendo dimensiones adicionales obtendríamos un hipercubo, aunque más de tres dimensiones no pueden visualizarse fácilmente. Los datos pueden solicitarse directamente en cualquier combinación de dimensiones, lo que deja de lado a las consultas de bases de datos más complejas. Existen herramientas para la visualización de los datos según las dimensiones elegidas por el usuario.

**Figura 29.2.** Un modelo de matriz bidimensional.

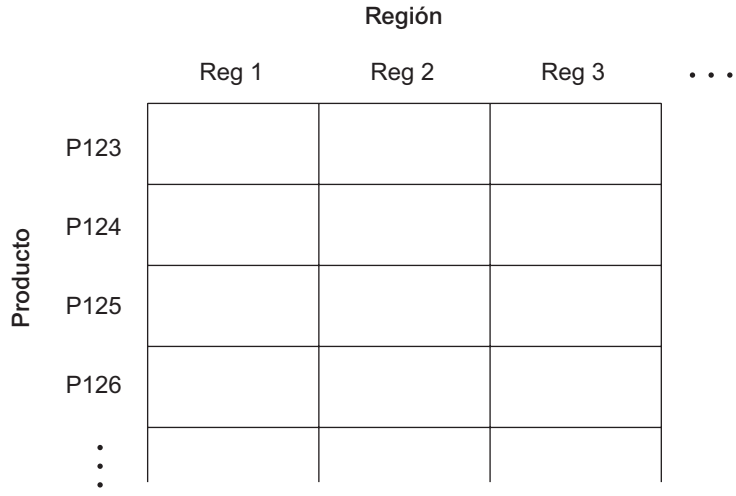
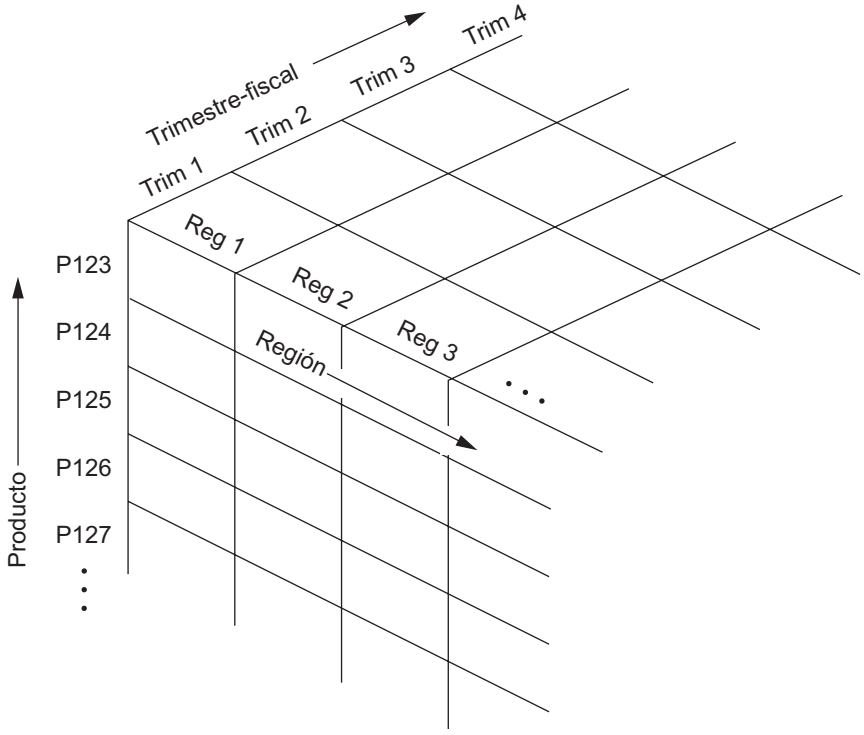


**Figura 29.3.** Un modelo de cubo de datos de tres dimensiones.




El cambio desde una jerarquía (u orientación) dimensional a otra en un cubo de datos se logra fácilmente gracias a una técnica llamada **pivotaje** (o rotación). En esta técnica, el cubo de datos puede ser observado como si rotara para mostrar una orientación diferente de los ejes. Por ejemplo, se puede pivotar el cubo para mostrar las ventas regionales como filas, los totales de ventas por periodos como columnas y los productos de la empresa en la tercera dimensión (véase la Figura 29.4). Por tanto, esta técnica equivale a tener una tabla de ventas regionales independiente por cada producto, donde cada una de ellas muestra las ventas por periodo región por región.

Los modelos multidimensionales se prestan fácilmente a la creación de vistas jerárquicas en algo que se conoce como visualización *roll-up* y *drill-down*. La **visualización *roll-up*** (compactar) mueve hacia arriba la jerarquía, agrupando en unidades más grandes a lo largo de una dimensión (por ejemplo, la suma semanal de datos por periodos o años). La Figura 29.5 muestra una visualización *roll-up* que se desplaza desde productos individuales hasta una basta lista de categorías de productos. En la Figura 29.6, una **visualización *drill-down*** (descomponer) ofrece la operación contraria, proporcionando una vista más fina (quizás descomponiendo por regiones las ventas de un país, las ventas regionales en subregiones y, a su vez, los productos en tipos).

El modelo de almacenamiento multidimensional implica dos tipos de tablas: de dimensión y de hechos. Una **tabla de dimensión** consta de tuplas de atributos de la dimensión. Una **tabla de hechos** puede considerarse como una agrupación de tuplas, una por cada hecho registrado. Este hecho contiene alguna variable, o variables, de medida u observación y las asocia mediante punteros con las tablas de dimensión. La tabla de hechos contiene los datos, y las dimensiones identifican cada tupla de esos datos. La Figura 29.7 contiene un ejemplo de tabla de hechos que puede ser vista desde la perspectiva de varias tablas de dimensión.

Dos esquemas multidimensionales muy comunes son el de estrella y el de copo de nieve (*snowflake*). El **esquema en estrella** consiste en una tabla de hechos con una única tabla por cada dimensión (véase la Figura 29.7). El esquema en **copo de nieve** es una variación del esquema en estrella en el que las tablas dimensiona-

**Figura 29.4.** Versión pivotada del cubo de datos de la Figura 29.3.

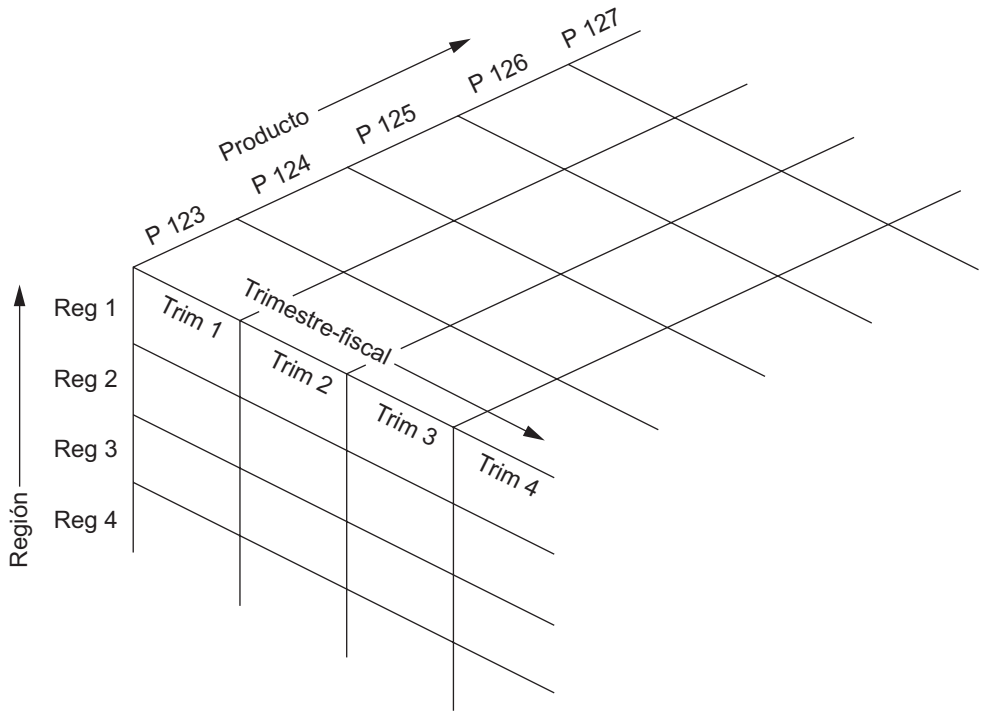
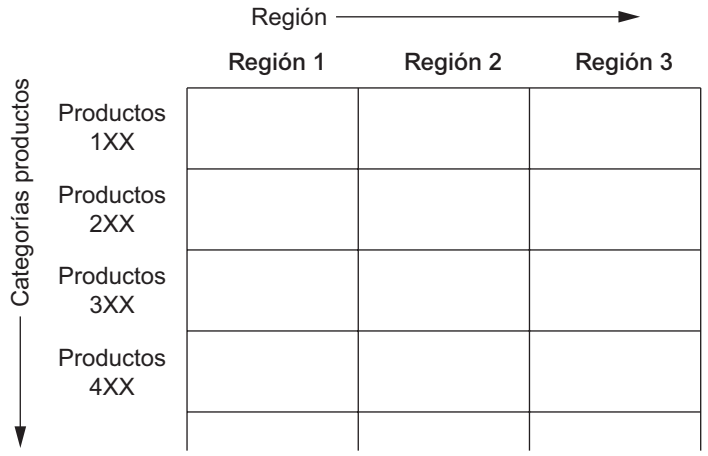


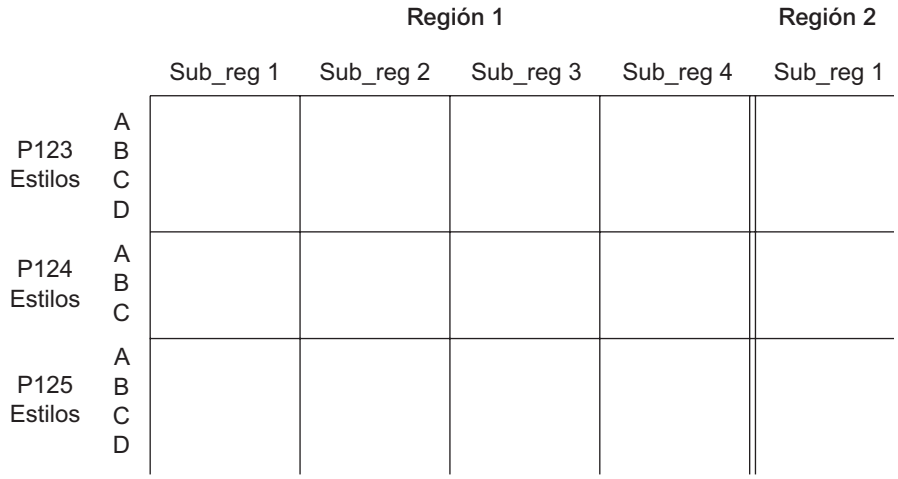

**Figura 29.5.** Operación *roll-up*.




les de éste están organizadas en una jerarquía para normalizarlas (véase la Figura 29.8). Algunas instalaciones son almacenes de datos normalizados hasta la tercera forma normal a fin de que puedan acceder a estos almacenes hasta el mejor nivel de detalle. Una **constelación de hechos** es un conjunto de tablas de hechos que comparten algunas tablas de dimensión. La Figura 29.9 muestra una de estas constelaciones con dos tablas de hechos, resultados de la empresa y proyecciones de la misma. Ambas comparten la tabla de dimensión llamada producto. Las constelaciones de hechos limitan las posibles consultas al almacén.

El almacenamiento en los almacenes de datos también utiliza técnicas de indexación para conseguir un mayor rendimiento en el acceso (consulte el Capítulo 14 para obtener más detalles acerca de la indexación). La **inde-**

**Figura 29.6.** Operación *drill-down*.




**xación *bitmap*** es una técnica que construye un vector de bits por cada valor del dominio (columna) a indexar, y funciona muy bien para aquéllos que tienen una baja cardinalidad. Existe un bit en la posición j-enésima del vector si la fila j-enésima contiene el valor a indexar. Por ejemplo, imagine un inventario de 100.000 coches con una indexación *bitmap* para el tipo de vehículo. Si existen cuatro de estos tipos (económico, compacto, gama media y lujo) encontraremos cuatros vectores de bits, conteniendo cada uno de ellos 100.000 bits (12,5K) hasta completar un tamaño total de índice de 50K. La indexación *bitmap* puede ofrecer unas ventajas importantes en operaciones de entrada/salida y de espacio de almacenamiento en dominios de baja cardinalidad. Con los vectores de bits, una indexación *bitmap* puede ofrecer unas sorprendentes mejoras en el rendimiento de las comparaciones, agregaciones y concatenaciones.

En un esquema en estrella, los datos dimensionales pueden ser indexados a las tuplas de la tabla de hechos mediante una **indexación de concatenación (*join indexing*)**. Este tipo de indexación es muy frecuente a la hora de mantener las relaciones establecidas entre los valores de una clave primaria y una clave externa (*foreign key*), y relacionan los valores de una dimensión de un esquema en estrella con las filas de la tabla de hechos. Por ejemplo, considere una tabla de hechos de ventas que tiene como dimensiones las ciudades y los periodos fiscales. Si existiera una indexación de concatenación en la ciudad, ésta mantendría, por cada ciudad, los ID de tupla de las tuplas que contuvieran esa cuidad. Los índices de concatenación pueden involucrar a varias dimensiones.

**Figura 29.7.** Un esquema en estrella con tablas de hechos y dimensionales.

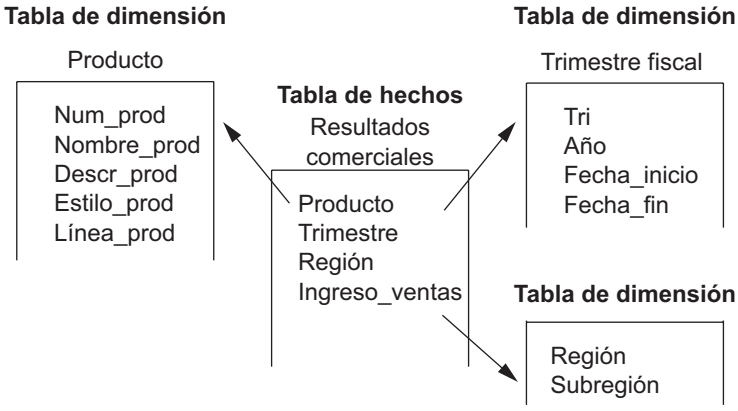


**Figura 29.8.** Un esquema en copo de nieve.

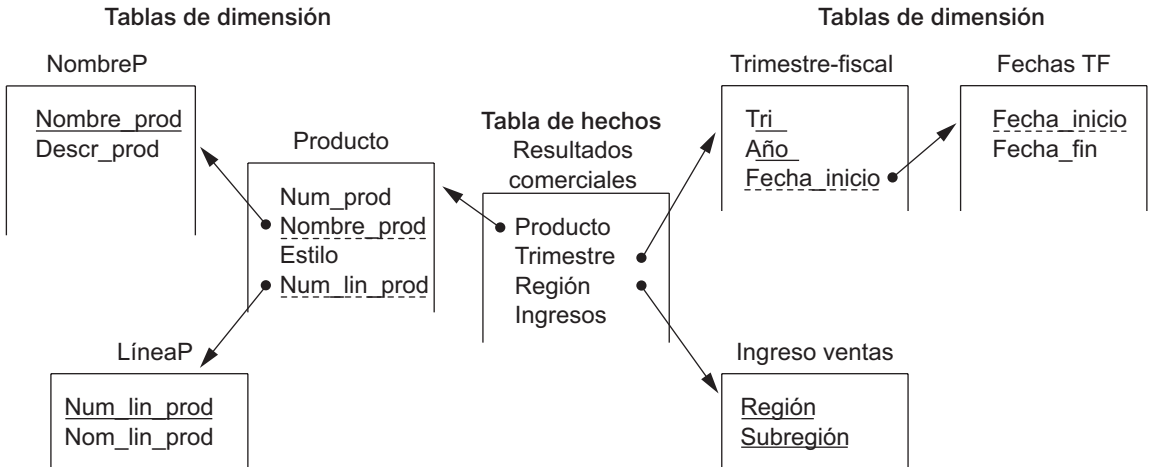
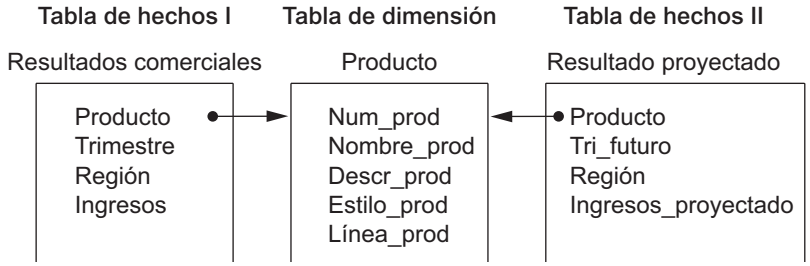



**Figura 29.9.** Una constelación de hechos.




El almacenamiento en almacenes de datos puede facilitar el acceso a datos resumen beneficiándose de la no volatilidad de la información y del grado de previsibilidad de los análisis que serán realizados al utilizarlos. Para ello se utilizan dos métodos: (1) tablas más pequeñas que incluyen los datos resumen, como las ventas trimestrales o los ingresos por línea de producto y (2) la codificación de nivel (por ejemplo, semanal, trimestral o anual) en tablas existentes. Por comparación, la creación y el mantenimiento de unas estructuras como éstas serían algo excesivo en una base de datos volátil y orientada a las transacciones.

# 29.4 Construcción de un almacén de datos

A la hora de preparar un almacén de datos, sus constructores deben disponer de una visión general anticipada del uso que se le dará. Durante la fase de diseño, no existe forma de prever todas las posibles consultas y análisis que se realizarán. Sin embargo, el diseño sí que debe soportar las **consultas *ad-hoc* (temporales)**, esto es, el acceso a los datos con cualquier combinación significativa de los valores de los atributos de las tablas de dimensión o de hechos. Esto obliga a elegir un esquema apropiado que refleje por adelantado este uso.

La adquisición de los datos desde el almacén implica dar los siguientes pasos:

- Los datos deben ser extraídos de fuentes múltiples y heterogéneas como, por ejemplo, bases de datos o cualquier otro lugar en el que exista información relevante.
- Los datos deben estar formateados de forma que sean coherentes dentro del almacén. Los nombres, contenidos y dominios de los datos que provengan de fuentes inconexas deben ser reajustados. Por

ejemplo, las compañías subsidiarias de otras empresas más grandes pueden tener calendarios fiscales en los que sus periodos fiscales finalicen en fechas diferentes, situación ésta que dificulta la agregación de estos datos por periodo. Cada tarjeta de crédito puede comunicar sus transacciones de forma distinta, lo que haría difícil procesar todas las ventas a crédito. Todos estos formatos incoherentes deben ser unificados.

- Los datos deben estar limpios para garantizar su validez. Éste es un proceso complejo y farragoso que está considerado como el mayor consumidor de trabajo del proceso de construcción del almacén. En la entrada de los datos, la limpieza de los mismos debe producirse antes de cargarse en el almacén. No existe ningún concepto acerca de este proceso que sea específico del almacenamiento de datos y que no pueda aplicarse a una base de datos *host*. Sin embargo, y ya que la entrada de información debe examinarse y formatearse de forma coherente, los constructores de almacenes de datos deben tener esta posibilidad para comprobar la validez y la calidad de los mismos. El reconocimiento de los datos erróneos o incompletos es una tarea difícil de automatizar, y la limpieza que requiere una operación de este tipo puede llegar a ser agobiante. Existen algunos aspectos, como la comprobación del dominio, que pueden ser codificados fácilmente dentro de las rutinas de limpieza de datos. Sin embargo, la identificación de otros problemas en los datos puede ser bastante más complicada (por ejemplo, puede que queramos considerar como incorrecta la combinación Ciudad = 'San Francisco' y Estado = 'CT'). Una vez considerados problemas de esta índole, es preciso coordinar los datos procedentes de cada una de las fuentes a la hora de cargarlos en el almacén. A medida que los administradores de los datos de la empresa descubren que la información está siendo limpiada para su introducción en los almacenes querrán actualizar sus propios datos. El proceso de devolver los datos limpios al origen recibe el nombre de ***backflushing*** (véase la Figura 29.1).
- Los datos procedentes de distintas fuentes deben acomodarse al modelo de datos del almacén. Esto supone que la información procedente de bases de datos relacionales, orientadas a objetos o heredadas (red y/o jerárquicas) debe ajustarse a un modelo multidimensional.
- Los datos deben cargarse en el almacén. El volumen total de los datos en los almacenes hace que la carga de los mismos sea una tarea importante. Para ello se precisan herramientas de monitorización y métodos para la recuperación de cargas incompletas o incorrectas. Debido a este volumen de datos, la única posibilidad fiable es la actualización incremental. Es probable que emerja la política de refresco como un compromiso que tenga en cuenta las respuestas a estas preguntas:
  - ¿Cómo debe ser la actualización de los datos?
  - ¿Puede desconectarse el almacén y, en caso afirmativo, durante cuánto tiempo?
  - ¿Cuáles son las interdependencias de los datos?
  - ¿Cuál es la disponibilidad de almacenamiento?
  - ¿Cuáles son los requerimientos de distribución (como la replicación y el particionamiento)?
  - ¿Cuál es el tiempo de carga (incluyendo la limpieza, el formateo, el copiado, la transmisión y la reconstrucción de los índices)?

Como ya hemos comentado, las bases de datos deben tener un cierto equilibrio entre la eficiencia en las transacciones y los requerimientos de las consultas (peticiones temporales del usuario). Pero un almacén de datos está optimizado para las necesidades de acceso del equipo de toma de decisiones. El almacenamiento de la información en un almacén refleja esta especialización y supone los siguientes procesos:

- El almacenamiento de los datos en base al modelo de datos del almacén.
- La creación y el mantenimiento necesario de las estructuras de datos.
- La creación y el mantenimiento adecuado de las rutas de acceso.

- Suministro de información variable en el tiempo a medida que se añadan nuevos datos.
- Soporte para la actualización del almacén de datos.
- Refresco de los datos.
- Depuración de los datos.

Aunque lograr un tiempo de acceso adecuado sea la prioridad inicial a la hora de construir el almacén, el volumen total de información contenido en él suele hacer imposible recargar más tarde el almacén en su totalidad. Entre las posibles alternativas se incluyen el refresco selectivo (parcial) de los datos y versiones de almacenes separadas (¡lo que obliga a disponer del doble de capacidad de almacenamiento!). Cuando se utilizan mecanismos incrementales de refresco de datos, es necesario depurar periódicamente dichos datos; por ejemplo, un almacén que mantenga la información de los doce periodos de negocio anteriores puede que necesite depurar su información cada año.

Los almacenes de datos deben estar también diseñados teniendo en cuenta el entorno en el que residirán. Entre las consideraciones de diseño importantes a tener en cuenta pueden incluirse las siguientes:

- Proyecciones de uso.
- El modelo de datos adecuado.
- Características de las fuentes de datos disponibles.
- Diseño del componente metadato.
- Diseño del componente modular.
- Diseño de la administración y el cambio.
- Consideraciones de arquitectura distribuida y paralela.

Vamos a tratar cada una de estas consideraciones por orden. El diseño de un almacén está dirigido inicialmente por las proyecciones de uso, esto es, las expectativas de quién lo utilizará y cómo lo hará. Por tanto, la elección de un modelo de datos que soporte este uso se convierte en un factor clave. Las proyecciones de uso y las características de las fuentes de datos del almacén deben ser tenidas en cuenta. El diseño modular es una necesidad práctica para permitir que el almacén evolucione junto con la organización y su entorno informativo. Por otro lado, un buen almacén de datos debe estar diseñado de forma que los administradores del mismo puedan planificar y realizar cambios en él a la que vez que proporcionan un soporte óptimo a los usuarios.

Si recuperamos el término metadato del Capítulo 2 recordaremos que estaba definido como la descripción de una base de datos incluyendo su definición de esquema. El **almacén de metadatos** es un componente fundamental de un almacén de datos. Este almacén incluye los metadatos tanto técnicos como de negocio. Los primeros, los metadatos técnicos, cubren los detalles del proceso de adquisición, las estructuras de almacenamiento, las descripciones de los datos, las operaciones y el mantenimiento del almacén, y la funcionalidad de soporte de acceso. Los segundos, los metadatos de negocio, incluyen todas las reglas de negocio relevantes y detalles organizativos que dan soporte al almacén.

La arquitectura del entorno de procesamiento distribuido de la organización es una característica determinante a la hora de diseñar el almacén.

Existen dos tipos de estructuras distribuidas básicas: el almacén distribuido y el almacén federado. En un **almacén distribuido**, todas las cuestiones de las bases de datos distribuidas son relevantes, por ejemplo, la replicación, el particionado, las comunicaciones y la coherencia. Una arquitectura distribuida puede ofrecer una serie de ventajas especialmente importantes al rendimiento del almacén, como son un equilibrado mejorado de la carga, la escalabilidad del rendimiento y una mayor disponibilidad. Un único almacén de metadatos replicado puede residir en cada sede. La idea del **almacén federado** es similar a la de una base de datos federada: una alianza descentralizada de almacenes de datos autónomos, cada uno de ellos con su propio almacén de metadatos. Dada la magnitud inherente de los almacenes de datos, es probable que estas alianzas

consistan en componentes a una escala menor, como los mercados de datos. Las empresas de mayor tamaño pueden preferir federar mercados de datos en lugar de construir almacenes de datos gigantes.

# 29.5 Funcionalidad típica de un almacén de datos

Los almacenes de datos existen para facilitar consultas temporales complejas y que precisan del tratamiento de mucha información. En consecuencia, deben ofrecer un soporte de consulta mucho mayor y más eficiente que el que se demanda por parte de las bases de datos transaccionales. El componente de acceso del almacén de datos soporta la funcionalidad de hoja de cálculo mejorada, el procesamiento eficiente de consultas, consultas estructuradas, consultas temporales, minería de datos y vistas materializadas. En particular, la funcionalidad de hoja de cálculo mejorada incluye soporte para aplicaciones *state-of-theart* y programas OLAP. Esta oferta de funcionalidades programadas incluye las siguientes:

- ***Roll-up*.** Los datos se resumen a través de una generalización ascendente (por ejemplo, de semanal a trimestral o anual).
- ***Drill-down*.** Se dan a conocer los niveles de detalle ascendentes (es el complemento de *roll-up*).
- **Pivotaje.** Se lleva a cabo una fabulación cruzada (conocida también como rotación).
- ***Slice and dice* (aislar y reagrupar).** Se llevan a cabo operaciones de proyección en las dimensiones.
- **Ordenación.** La información se ordena por un valor ordinal.
- **Selección.** Los datos están disponibles por valor o rango.
- **Atributos derivados (computados).** Los atributos son obtenidos mediante operaciones con los valores almacenados y derivados.

Ya que los almacenes de datos están libres de las restricciones del entorno transaccional, se obtiene una importante mejora en el procesamiento de las consultas. Entre las herramientas y técnicas usadas están la transformación de la consulta, la unión e intersección de índices, funciones especiales **ROLAP** (OLAP relacional, *Relational OLAP*) y **MOLAP** (OLAP multidimensional, *Multidimensional OLAP*), extensiones SQL, métodos de concatenación avanzados y escaneo inteligente (como en las consultas múltiples *piggy-backing*).

La mejora en el rendimiento se ha alcanzado también gracias al procesamiento paralelo. Las arquitecturas de servidor paralelo incluyen SMP (Multiprocesador simétrico, *Symmetric MultiProcessor*), cluster, MPP (Procesamiento en paralelo masivo, *Massively Parallel Processing*), y combinaciones de ellas.

Los trabajadores del conocimiento y las personas que toman decisiones emplean herramientas que van desde las consultas paramétricas hasta las consultas temporales y la minería de datos. Por consiguiente, el componente de acceso del almacén debe ofrecer soporte para consultas estructuradas (tanto paramétricas como temporales). A la vez, todo esto confecciona un entorno de consulta administrada. La minería de datos emplea técnicas procedentes del análisis estadístico y la inteligencia artificial. Los primeros pueden ser realizados por hojas de cálculo avanzadas, por programas sofisticados de análisis estadístico o por software de desarrollo propio. Habitualmente, se emplean técnicas como el *lagging* (retardo), el promedio variable y los análisis de regresión. Las técnicas de inteligencia artificial, que pueden incluir algoritmos genéticos y redes neurales, se emplean para la clasificación y permiten descubrir conocimientos a partir de los datos del almacén que, de otra forma, serían impredecibles o difíciles de especificar en las consultas (tratamos con detalle la minería de datos en el Capítulo 28).

# 29.6 Almacenes de datos frente a vistas

Algunas personas consideran a los almacenes de datos como una extensión de las vistas de bases de datos. Como ya se mencionó previamente, las vistas materializadas son una forma de reunir requerimientos para

mejorar el acceso a los datos (consulte el Capítulo 8 para obtener más información acerca de la vistas). Las vistas materializadas se han explorado por su rendimiento mejorado. Las vistas, sin embargo, sólo ofrecen un subconjunto de las funciones y posibilidades de los almacenes de datos. Ambos conceptos son similares en cuanto a que realizan extracciones de la base de datos de tipo sólo lectura y en que están orientados al tema. Sin embargo, se diferencian en varias cosas:

- Los almacenes de datos existen como un sistema de almacenamiento permanente en lugar de materializarse bajo petición.
- Los almacenes de datos no suelen ser relacionales, sino más bien multidimensionales. Las vistas de una base de datos relacional son relacionales.
- Los almacenes de datos pueden indexarse para mejorar su rendimiento. Las vistas no pueden estarlo de forma independiente de las bases de datos subyacentes.
- Los almacenes de datos ofrecen soporte específico de funcionalidad; las vistas no.
- Los almacenes de datos proporcionan grandes cantidades de datos integrados y, con frecuencia, temporales (generalmente más de los contenidos en una base de datos), mientras que las vistas son un extracto de una base de datos.

# 29.7 Problemas y problemas abiertos en los almacenes de datos

## 29.7.1 Dificultades para la implementación de los almacenes de datos

Existen algunos problemas operativos importantes con el almacenamiento de datos: la construcción, la administración y el control de la calidad. La administración del proyecto (el diseño, construcción e implementación del almacén) no debería ser subestimada. La construcción de un almacén corporativo en una empresa de gran tamaño es una ardua tarea que puede llevar años de conceptualización e implementación. Debido a la dificultad y a la cantidad de tiempo necesarias para abordar esta tarea, el ampliamente desarrollado e implementado mercado de datos puede ofrecer una atractiva alternativa, especialmente a aquellas organizaciones con unas necesidades urgentes de soporte para OLAP, DSS y/o minería de datos.

La administración de un almacén de datos es una empresa compleja, proporcional al tamaño y complejidad del mismo. Cualquier organización que pretenda administrar un almacén de datos debe tener muy clara la compleja naturaleza de esta gestión. Aunque diseñado para el acceso de lectura, un almacén de datos no es una estructura más estática que cualquiera de sus fuentes de información. Las bases de datos originales deben evolucionar. El esquema y el componente de adquisición del almacén deben actualizarse para tener en cuenta estas evoluciones.

Un problema significativo en el almacenamiento de datos es la calidad del control de los datos. Tanto la calidad como la coherencia de los datos son la principal preocupación. Aunque durante su adquisición los datos pasan por un proceso de limpieza, la calidad y la coherencia siguen siendo un quebradero de cabeza para el administrador de la base de datos. La mezcla de datos procedentes de fuentes dispares y heterogéneas es un gran problema dadas las diferencias existentes en los nombres, definiciones de dominios, identificación de números, etc. Cada vez que una base de datos se modifica, el administrador del almacén debe considerar las posibles interacciones con otros elementos del mismo.

Las proyecciones de uso deben estimarse a la baja previamente a la construcción del almacén y revisarse continuamente para reflejar los requisitos en cada momento. A medida que los patrones de utilización se aclaran y cambian a lo largo del tiempo, el almacenamiento y las rutas de acceso deben afinarse para mantener opti-

mizado el uso que la organización hace del almacén de datos. Esta actividad debe mantenerse a lo largo de la vida del mismo para que siempre esté por delante de la demanda. El almacén debe estar diseñado también para acomodar la agregación y el desgaste de las fuentes de datos sin que se tenga que realizar un mayor rediseño. Los orígenes de los datos evolucionan, y el almacén debe ser capaz de acomodarse a estos cambios. Adecuar los datos disponibles al modelo de datos del almacén será un continuo desafío, una tarea que es más un arte que una ciencia. Debido al vertiginoso cambio que se produce en la tecnología, tanto los requisitos como las posibilidades del almacén variarán considerablemente a lo largo del tiempo. Por otro lado, la propia tecnología de almacenamiento de datos continuará evolucionando durante algún tiempo, por lo que las estructuras de componentes y las funcionalidades serán actualizadas continuamente. Este cambio es una buena razón para un diseño modular pleno de los componentes.

La administración de un almacén de datos requerirá de una mayor pericia que la necesaria para tratar con una base de datos tradicional. Por ello será preciso contar con un grupo de expertos altamente cualificado con áreas de conocimiento solapadas, en lugar de disponer de una sola persona. Al igual que ocurre con la administración de una base de datos, la de un almacén de datos es una tarea técnica sólo en parte; una gran parte de la responsabilidad supone trabajar de forma eficiente con todos los miembros de la organización interesados en el almacén. Sin embargo, los problemas con los que se enfrenta a veces el administrador de una base de datos son mucho más importantes en el caso de un almacén, por lo que el espectro de sus responsabilidades es también mucho mayor.

El diseño de la función de administración y de la selección del equipo que se encargará de ello son tareas cruciales. La gestión del almacén de datos en organizaciones más grandes supondrá con seguridad una tarea más importante. En la actualidad ya pueden encontrarse en el mercado herramientas comerciales para las funciones de administración. La gestión efectiva del almacén de datos será una función de equipo, lo que requerirá un amplio conjunto de habilidades técnicas, coordinación precisa y liderazgo efectivo. Al igual que hemos preparado la posible evolución del almacén, debemos reconocer también que el equipo técnico deberá evolucionar, por fuerza, con él.

### 29.7.2 Problemas abiertos en el almacenamiento de datos

Existe mucha palabrería típica del marketing alrededor del término “almacén de datos”; las expectativas exageradas puede que decaigan, pero el concepto de colecciones de datos integradas que permitan realizar análisis sofisticados y toma de decisiones indudablemente se verá reforzado.

El almacenamiento de datos es un área activa de investigación que, muy probablemente se vea incrementada en un futuro cercano a medida que los almacenes y los mercados de datos proliferen. Los problemas antiguos sufrirán un nuevo enfoque; por ejemplo, la limpieza de los datos, la indexación, el particionado y las vistas recibirán una renovada atención.

La investigación académica de las tecnologías de almacenamiento de datos se centrarán en aquellos aspectos de la automatización del almacén que, en la actualidad, precisan de la intervención humana, como son la adquisición de los datos, el control de su calidad, la selección y construcción de las rutas de acceso y las estructuras adecuadas, el mantenimiento automático, la funcionalidad y la optimización del rendimiento. La aplicación de funcionalidades de bases de datos activas (consulte la Sección 23.1) en el almacén será un área que recibirá también una considerable atención. La incorporación de reglas de dominio y de negocio apropiadas a la creación y a los procesos de mantenimiento del almacén podrían hacerle más inteligente, relevante y auto-gobernable.

En la actualidad ya pueden encontrarse en el mercado programas de distintos fabricantes para almacenar datos, los cuales se centran principalmente en la administración del almacén y en las aplicaciones OLAP/DSS. Otras tareas del proceso de almacenamiento, como el diseño y la adquisición de los datos (especialmente, su limpieza), están siendo desarrolladas fundamentalmente por los equipos internos de directivos y consultores de TI.

# 29.8 Resumen

En este capítulo hemos examinado el campo conocido como almacenamiento de datos. Esta área puede verse como un proceso que precisa de la realización de una serie de tareas previas. Por el contrario, la minería de datos (consulte el Capítulo 28) puede considerarse como una actividad que dibuja el conocimiento contenido en un almacén de datos. Hemos introducido una serie de conceptos clave relacionados con el almacenamiento de datos y hemos comentado la funcionalidad especial asociada con una vista de datos multidimensional. También hemos tratado las distintas formas en las que los almacenes de datos suministran información a las personas que toman las decisiones a un nivel de detalle correcto, basándose en una organización y perspectiva apropiadas.

## Preguntas de repaso

**29.1.** ¿Qué es un almacén de datos? ¿En qué difiere de una base de datos?

**29.2.** Defina los términos: OLAP, ROLAP, MOLAP y DSS.

**29.3.** Describa las características de un almacén de datos. Clasifíquelas en base a la funcionalidad de un almacén y las ventajas derivadas de él que obtienen los usuarios.

**29.4.** ¿Qué es el modelo de datos multidimensional? ¿Cómo se utiliza en el almacenamiento de datos?

**29.5.** Defina los siguientes términos: *esquema en estrella*, *esquema en copo de nieve*, *constelación de hechos*, *mercados de datos*.

**29.6.** ¿Qué tipos de índices se construyen para un almacén? Ilustre los usos de cada uno de ellos con un ejemplo.

**29.7.** Describa los pasos necesarios para la construcción de un almacén.

**29.8.** ¿Qué consideraciones juegan un papel principal en el diseño de un almacén?

## Bibliografía seleccionada

El almacenamiento de datos se ha convertido en un tema popular que ha aparecido en muchas publicaciones en los últimos años. Inmon (1992) tiene el mérito de haber difundido el término. Codd (1993) popularizó el término OLAP y definió un conjunto de características para que los almacenes de datos lo soportasen. Kimball (1996) es conocido por su contribución al desarrollo del campo del almacenamiento de datos. Mattison (1996) es uno de los diversos libros sobre almacenamiento de datos que ofrece un análisis global de las técnicas disponibles para los almacenes de datos y las estrategias que deben seguir las compañías para implementarlos. Ponniah ofrece una panorámica práctica muy correcta del proceso de construcción de un almacén de datos, desde la recopilación de los requisitos hasta el mantenimiento de la implantación. Bischoff y Alexander (1997) es una recopilación de consejos de los expertos. Chaudhuri y Dayal (1997) ofrece un excelente tutorial sobre el tema, mientras que Widom (1995) se centra en un buen número de problemas de investigación.

CAPÍTULO 30

# Tecnologías y aplicaciones emergentes de bases de datos

A lo largo de este libro hemos revisado diversos temas en relación con el modelado, el diseño y las funciones de las bases de datos, así como con la estructura interna y los temas de rendimiento en relación con los sistemas de gestión de bases de datos. En el Capítulo 27 tratamos las bases de datos en Internet que proporcionan acceso universal a los datos y vimos el uso de XML que facilitará el desarrollo de aplicaciones en las que se encuentren implicadas bases de datos de distinta naturaleza y sobre distintas plataformas DBMS. En los dos capítulos anteriores, tratamos las variantes en las tecnologías de gestión de bases de datos, como la minería de datos y los almacenes de datos, que proporcionan bases de datos de gran tamaño y herramientas para el soporte a la toma de decisiones. En este capítulo dirigimos nuestra atención a dos tipos de desarrollos en continua evolución en el área de las bases de datos: las tecnologías emergentes de bases de datos y sus principales campos de aplicación. No pretendemos hacerlo de forma exhaustiva; sólo nos centraremos en algunas tecnologías relevantes y en los avances en las aplicaciones. El primer punto tiene que ver con la creación de nuevas funcionalidades en los DBMSs para poder trabajar con diferentes tipos de nuevas aplicaciones, entre las que se incluyen las bases de datos móviles que permiten a los usuarios el acceso universal y flexible a los datos desde cualquier lugar y desde cualquier equipo, y las bases de datos multimedia que proporcionan la posibilidad de almacenamiento y procesamiento de información multimedia. En las Secciones 30.1 y 30.2 se presentarán y revisarán brevemente los temas y los modelos relacionados con la resolución de los problemas específicos que surgen en las tecnologías de bases de datos móviles y multimedia.

Posteriormente, veremos dos dominios de aplicación que se han basado tradicionalmente en el procesamiento manual de los sistemas de archivos, también llamados soluciones con sistemas a medida. En la Sección 30.3 se revisan los sistemas de información geográfica, que gestionan datos geográficos únicamente o datos espaciales combinados con datos no espaciales, como los datos del censo. En la Sección 30.4 se revisan las bases de datos biológicas y sus aplicaciones, en particular las que contienen datos genéticos sobre diferentes organismos, incluidos los datos del genoma humano. Aparte de esto, se caracterizan por su naturaleza *estática* (aquella situación en la que el usuario final sólo puede obtener datos de la base de datos); la actualización de nueva información está restringida a los expertos del dominio que revisan y analizan los nuevos datos introducidos.

# 30.1 Bases de datos móviles[1]

Los recientes avances en la tecnología móvil e inalámbrica han conducido a la **computación móvil**, una nueva dimensión en la comunicación y el procesamiento de los datos. Los dispositivos computacionales móviles junto con las comunicaciones inalámbricas, permiten a los clientes el acceso a los datos desde prácticamente cualquier lugar y en cualquier momento. Esta funcionalidad resulta especialmente útil a las empresas dispersas geográficamente. Entre los ejemplos típicos podríamos incluir las agendas electrónicas, los servicios de información de noticias y de valores de bolsa, y la gestión comercial automatizada. Sin embargo, existen varios problemas en el hardware y en el software que deben ser resueltos antes de que se puedan utilizar en su totalidad las posibilidades de la computación móvil.

Algunos de los problemas del software, entre los cuales podemos citar la gestión de datos, la gestión de transacciones y la recuperación de bases de datos, tienen su origen en los sistemas de bases de datos distribuidas. En la computación móvil, sin embargo, estos problemas resultan más complicados, principalmente debido a que la conectividad en las comunicaciones inalámbricas es a veces limitada e intermitente, a que la duración de la energía (batería) es limitada en las unidades móviles y a la topología cambiante de la red. Además, la computación móvil introduce nuevos retos y posibilidades de arquitectura.

## 30.1.1 Arquitectura de la computación móvil

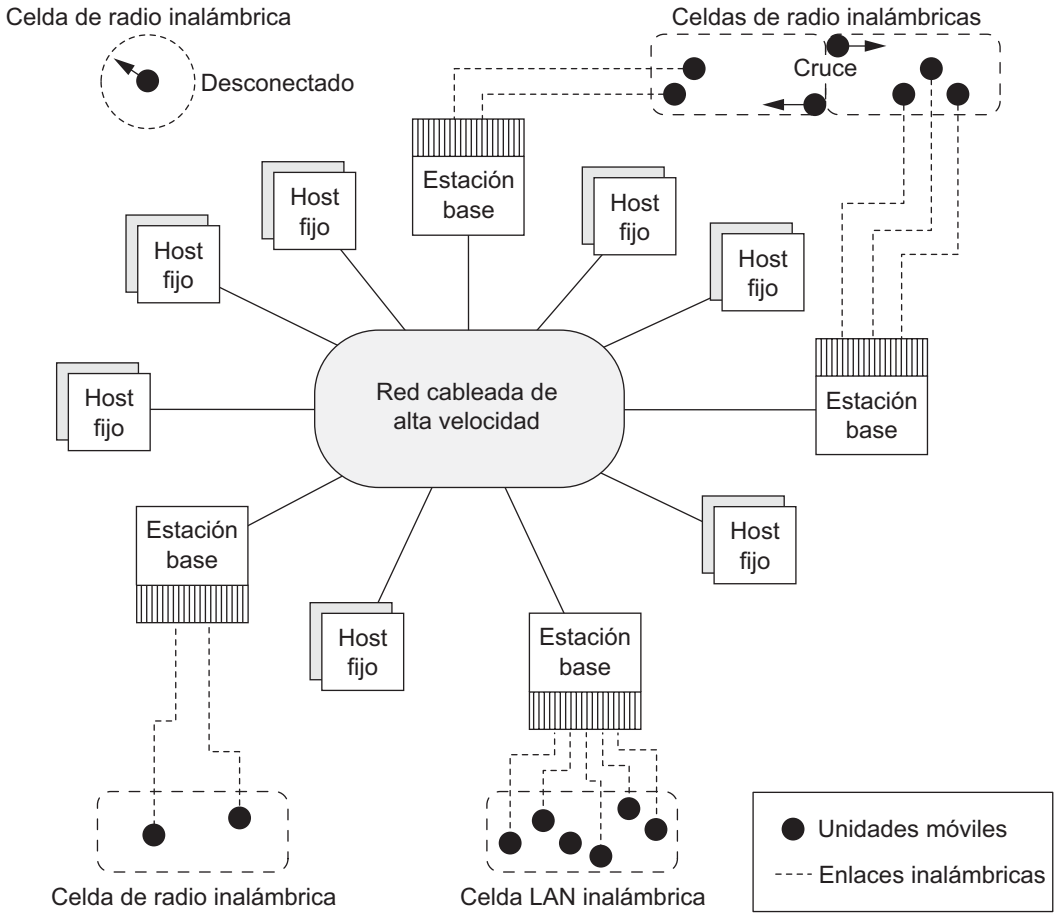
**Plataforma móvil basada en infraestructura.** La arquitectura general de una plataforma móvil se muestra en la Figura 30.1. Se trata de una arquitectura distribuida en la que varios computadores, a los que se les denomina generalmente ***Host* fijo (FS)** y **Estación base (BS)**, se interconectan a través de una red de cable de alta velocidad. Los *hosts* fijos son, por lo general, computadores dedicados que no están equipados normalmente para la gestión de unidades móviles pero que pueden ser configurados para ello. Las estaciones base son pasarelas entre las **Unidades móviles (MU)** y la red fija. Están equipadas con interfaces inalámbricas y ofrecen servicios de acceso a la red a las unidades móviles clientes.

El medio inalámbrico sobre el que se comunican las unidades móviles y las estaciones base dispone de un ancho de banda significativamente inferior al de la red fija. La generación actual de tecnología inalámbrica tiene una tasa de transmisión de datos que varía desde decenas a cientos de kilobits por segundo (telefonía celular 2G) hasta decenas de megabits por segundo (Ethernet inalámbrica, conocida popularmente como WiFi). La Ethernet (por cable) actual, en comparación, proporciona tasas de transmisión en torno a los cientos de megabits por segundo.

Aparte de las tasas de transmisión, existen también otras características que diferencian las opciones de conectividad inalámbrica. Entre algunas de estas características podemos incluir el rango de frecuencias, la interferencia, la ubicación de un acceso y la posibilidad de conmutación de paquetes. Algunas opciones de acceso inalámbrico permiten la itinerancia continua dentro de una zona geográfica (por ejemplo, las redes celulares), mientras que las redes WiFi se ubican en torno a una estación base. Algunas redes, como WiFi y Bluetooth, utilizan zonas libres de licencia dentro del espectro de frecuencias, lo cual puede producir interferencias con otros aparatos, como los teléfonos inalámbricos. Por último, las redes inalámbricas modernas pueden realizar la transferencia de datos en unidades llamadas paquetes, que es lo que se utiliza por lo general en las redes por cable para mantener el ancho de banda. Las aplicaciones inalámbricas deben tener en cuenta estas características a la hora de elegir una opción de comunicaciones. Por ejemplo, los objetos materiales bloquean la señal de infrarrojos. Aunque resulte molesto para algunas aplicaciones, este bloqueo permite la comunicación inalámbrica segura dentro de una habitación cerrada. Las unidades móviles se pueden desplazar con libertad dentro de un **dominio de movilidad geográfica**, un área circunscrita por la cobertura de la red inalámbrica. Para gestionar la movilidad de las unidades, todo el dominio de movilidad geográfica se divide en uno o más dominios de menor tamaño denominados **celdas**, cada una de las cuales está controlada por, al menos, una

[1] Se agradece la contribución de Waigan Yee y Wanxia Xie a esta sección.

**Figura 30.1.** Una arquitectura general de plataforma móvil basada en infraestructura.




Adaptado de Dunham y Helal (1995).

estación base. La disciplina de movilidad requiere que el movimiento de las unidades móviles no esté restringido dentro de las celdas que pertenecen a un dominio de movilidad geográfica, a la vez que se mantiene la **contigüidad de acceso** a la información; es decir, el movimiento (especialmente el intercelular), no debería afectar negativamente al proceso de obtención de información.

**Plataforma móvil no basada en infraestructura.** La arquitectura de comunicaciones que acabamos de describir está diseñada para dar al cliente la impresión de que se encuentra conectado a una red fija, emulando la arquitectura cliente/servidor tradicional. Además de lo anterior, las comunicaciones inalámbricas posibilitan otros tipos de arquitecturas. Una de ellas es una plataforma móvil sin infraestructura, también llamada red móvil ad hoc (MANET) y que se muestra en la Figura 30.2. En una MANET, las unidades móviles que pertenecen a ella no necesitan comunicarse a través de una red fija, sino que forman una propia entre ellas utilizando tecnologías de bajo coste como Bluetooth. En una MANET, las unidades móviles se encargan de encaminar sus propios datos, actuando en realidad como estaciones base además de como clientes. Además, deben ser lo suficientemente robustas como para gestionar los cambios en la topología de la red, como la incorporación o abandono de la red por parte de otras unidades móviles. Según esto, las aplicaciones MANET añaden un nivel adicional de complejidad a los temas relacionados con la computación móvil y la gestión de datos en entornos móviles.

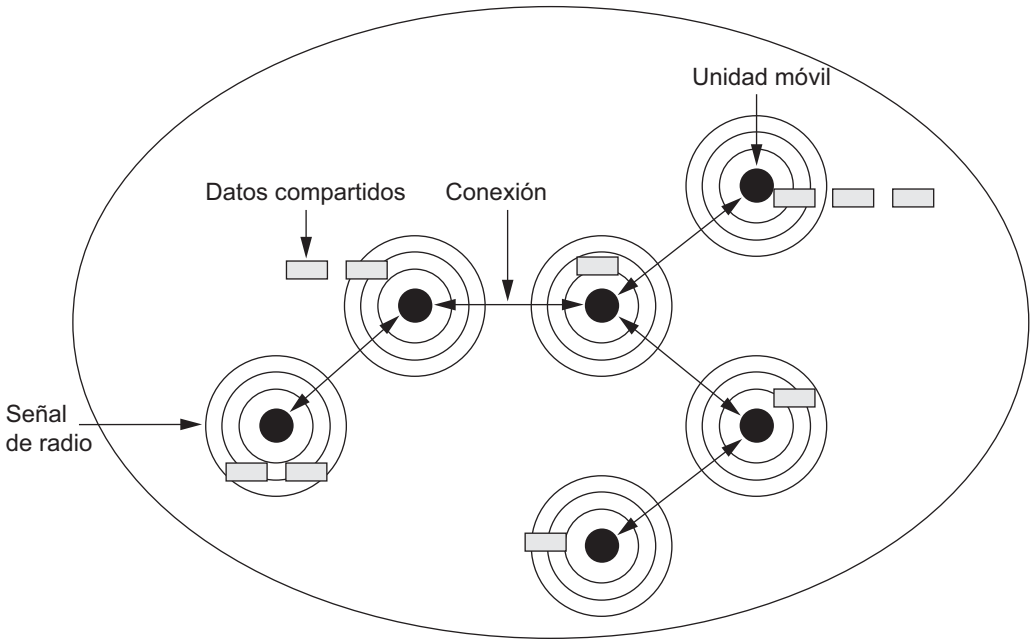
Las aplicaciones MANET se pueden englobar dentro del paradigma de comunicación entre iguales (P2P), en el sentido de que una unidad móvil es, a la vez, cliente y servidor. La arquitectura de comunicación entre iguales es un tipo de red en la que cada uno de los nodos tiene las mismas responsabilidades. Esto contrasta con la arquitectura cliente/servidor en la que los servidores se dedican a servir a los clientes. Existen dos tipos de redes de comunicación entre iguales: **redes P2P híbridas** y **redes P2P descentralizadas**. En una red P2P híbrida, existe un servidor de directorio centralizado que puede proporcionar la información a todos los nodos. Normalmente, los nodos publican información sobre sus recursos en el directorio del servidor y consultan al servidor la información sobre los recursos. Napster es un ejemplo típico de red P2P híbrida. En una red P2P descentralizada no existen servidores centrales. En la primera generación de redes P2P descentralizadas como, por ejemplo, Gnutella, las consultas y el enrutamiento se realizan mediante difusión, lo cual provoca una inundación de mensajes en la red. En la segunda generación de redes P2P descentralizadas (es decir, redes P2P estructuradas, como Chord y CAN), las consultas y el enrutamiento se basan en tablas *hash* distribuidas. Las redes P2P estructuradas garantizan que se accede a los demás nodos en un número determinado de saltos. Las aplicaciones MANET que se revisan más adelante se basan en redes P2P descentralizadas.

Las principales características de las aplicaciones MANET son las siguientes:

- **Desconexiones frecuentes**. No es posible suponer que todos los dispositivos móviles se encuentren siempre conectados en el desarrollo de aplicaciones cooperativas. Es necesario tratar con las desconexiones frecuentes, a veces intencionadas, de las unidades móviles.
- **Particionado frecuente de la red.** Las desconexiones frecuentes y el cambio de posición de los dispositivos móviles pueden provocar frecuentes particionados de la red. El particionado de la red es un serio problema a tener en cuenta durante el desarrollo de aplicaciones MANET.
- **Control centralizado dificultoso.** A veces, un servidor central en una MANET no resulta útil o no es posible. Por ejemplo, un servidor central en el caso de particionado de la red inutilizará los dispositivos en todas las particiones excepto en una.
- **Heterogeneidad de los nodos.** Obviamente, existe heterogeneidad entre los nodos en el hardware y el software, incluidos los sistemas operativos, lenguajes y bases de datos disponibles en cada plataforma. Aparte de esto, los nodos son heterogéneos en cuanto a capacidades de computación como CPU y almacenamiento, ancho de banda y capacidad de batería. Es esencial realizar un balanceo de cargas en función de sus capacidades.

El procesamiento transaccional y el control de la consistencia de los datos se hacen más difíciles ya que existe la posibilidad del particionado de la red y no existe un control centralizado en esta arquitectura. La búsqueda de recursos y el enrutamiento de los datos por parte de las unidades móviles hacen que la computación en las MANET sea más complicada. Todo esto requiere el desarrollo de nuevos modelos, técnicas y algoritmos que nos permitan construir y utilizar las aplicaciones MANET. Como ejemplo de aplicaciones MANET se encuentran las aplicaciones de respuesta a emergencias ante desastres naturales o no naturales, el reparto de información en los campos de batalla, los sistemas de transporte por superficie, los juegos multiusuario, las pizarras compartidas y los calendarios distribuidos. Los trágicos sucesos del 11 de septiembre de 2001 demostraron claramente la necesidad de aplicaciones de este tipo. A pesar del hecho de que la ciudad de Nueva York disponía de un plan de emergencias, su capacidad de gestión del ataque fue limitada. Su desastrosa base de coordinación, ubicada bajo las Torres Gemelas fue destruida, así como la centralita telefónica de Verizon cercana. La pérdida de estos edificios inutilizó la capacidad de las autoridades de coordinar al personal y a los recursos de forma efectiva, con trágicas consecuencias. Se espera que este tipo de redes y las aplicaciones relacionadas tengan gran importancia dentro de unos pocos años. En la actualidad, las MANET son un área de investigación activa en el terreno académico y en la industria. Estas investigaciones se encuentran aún en sus primeras etapas; por tanto, lo que veremos a continuación se orientará hacia la arquitectura básica de computación móvil descrita anteriormente.

**Figura 30.2.** La arquitectura de MANET: una red móvil ad hoc.[2]




## 30.1.2 Características de los entornos móviles

Entre algunas de las características de la computación en entornos móviles podemos incluir la alta latencia de comunicaciones, la conectividad inalámbrica intermitente, la duración limitada de las baterías y, por supuesto, el cambio en la ubicación de los clientes. La latencia está motivada por los procesos inherentes al medio inalámbrico, como la codificación de los datos para su transmisión inalámbrica, y la sintonización y el filtrado de las señales inalámbricas en el receptor. La duración de las baterías está directamente relacionada con su tamaño, e indirectamente relacionada con las capacidades del dispositivo móvil y su rendimiento global. La conectividad intermitente puede ser intencionada o no intencionada. Las desconexiones no intencionadas suceden en áreas no cubiertas por la señal inalámbrica, por ejemplo en ascensores o en túneles. Las desconexiones intencionadas se producen por deseo del usuario, por ejemplo durante el despegue de un avión o cuando se apaga un dispositivo móvil. Por último, lo normal es que los clientes cambien su ubicación; de eso se trata. Todas estas características tienen un impacto en el tratamiento de los datos y las aplicaciones móviles robustas deberían tenerlas en cuenta durante su diseño.

Para compensar las altas latencias y la conectividad no fiable, los clientes almacenan réplicas de los datos importantes o accedidos con frecuencia para trabajar desconectados cuando sea necesario. Aparte de incrementar la disponibilidad de los datos y el tiempo de respuesta, este almacenamiento puede reducir también el consumo de energía del cliente eliminando la necesidad de hacer transmisiones inalámbricas que consumen energía en cada acceso a los datos.

Por otra parte, el servidor podría no contactar con el cliente. Es posible que un cliente esté inaccesible porque se encuentre en **reposo** (un estado de ahorro de energía en el que se sitúan muchos subsistemas) o porque se encuentre fuera del rango de alcance se una estación base. En cualquier caso, ni el cliente ni el servidor pueden contactar con el otro, y es necesario realizar modificaciones en la arquitectura para solucionar esos casos.

[2] Esta arquitectura se basa en la propuesta IETF de 1999 con comentarios de Carson y Macker (1999).

Para el caso de los componentes incomunicados se agregan **intermediarios** a la arquitectura. En el caso de un cliente (y, por simetría, para un servidor), el intermediario o *proxy* puede almacenar las actualizaciones destinadas al servidor. Cuando la conexión vuelva a estar disponible, el intermediario dirigirá de forma automática estas actualizaciones almacenadas hacia su destino final.

Según se indicó anteriormente, la computación móvil plantea retos a los servidores y a los clientes. La latencia existente en la comunicación inalámbrica hace que la escalabilidad sea un problema. Ya que la latencia en las comunicaciones inalámbricas incrementa el tiempo para servir las peticiones de cada cliente, el servidor podrá tratar con pocos clientes. Una forma en la que los servidores alivian este problema es mediante la **difusión** de los datos siempre que sea posible. La difusión se aprovecha de una característica natural de las comunicaciones por radio, y es escalable ya que una única difusión de un elemento de datos puede resolver todas las peticiones pendientes de ese dato. Por ejemplo, en lugar de enviar información meteorológica a todos los clientes de una celda, un servidor podría limitarse a difundirla periódicamente. La difusión también reduce la carga del servidor, ya que los clientes no necesitan mantener las conexiones activas con él.

La movilidad de los clientes también plantea muchos retos en cuanto a la gestión de los datos. En primer lugar, los servidores deben seguir la pista a la ubicación de los clientes para encaminar los mensajes hacia ellos de forma eficiente. En segundo lugar, los datos de los clientes deberían ser almacenados en la ubicación de la red que minimice el tráfico necesario para accederlos. Mantener los datos en una ubicación fija incrementa la latencia de acceso si el cliente se aleja de allí. Por último, según se mencionó anteriormente, el hecho de desplazarse de una celda a otra debe resultar transparente para el cliente. El servidor debe ser capaz de dirigir el envío de los datos desde una estación base a la otra de forma elegante, sin que el cliente se entere.

La movilidad del cliente también propicia nuevas aplicaciones *basadas en la ubicación*. Por ejemplo, pensemos en una aplicación de agenda electrónica que le diga al usuario la situación del restaurante más próximo. Queda claro que *el más próximo* es en relación a la posición actual del cliente y que su desplazamiento podría invalidar los datos si estos han sido almacenados. Si se produce un desplazamiento, el cliente deberá invalidar partes de sus datos almacenados y solicitar la actualización de los datos a la base de datos.

### 30.1.3 Temas sobre gestión de datos

Desde el punto de vista de la gestión de los datos, la computación móvil puede ser considerada como una variante de la computación distribuida. Las bases de datos móviles pueden ser distribuidas bajo dos escenarios posibles:

1. Toda la base de datos se encuentra distribuida principalmente entre los componentes cableados, quizá con replicación total o parcial. Una estación base o un *host* fijo gestiona su propia base de datos mediante una funcionalidad de tipo DBMS, con funcionalidad adicional para la localización de unidades móviles y con funciones adicionales de consulta y tratamiento de transacciones que cumplan los requisitos de los entornos móviles.
2. La base de datos está distribuida entre los componentes cableados e inalámbricos. La responsabilidad de la gestión de los datos se reparte entre las estaciones base o entre los *hosts* fijos y las unidades móviles.

De acuerdo con lo anterior, los temas de gestión de datos distribuidos que vimos en el Capítulo 24 pueden ser aplicables también a las bases de datos móviles con las siguientes consideraciones y variaciones:

- **Distribución de datos y replicación.** Los datos se encuentran distribuidos de manera no uniforme entre las estaciones base y las unidades móviles. Las restricciones de consistencia dan lugar al problema de la gestión del almacenamiento intermedio. Los almacenes intermedios intentan proporcionar los datos que son accedidos o actualizados con más frecuencia a las unidades móviles que procesan sus propias transacciones y pueden quedar desconectados durante largos periodos de tiempo.

- **Modelos de transacción.** La tolerancia a fallos y la ejecución correcta de las transacciones se agravan en un entorno móvil. Una transacción móvil se ejecuta de forma secuencial a lo largo de varias estaciones base y, quizá, sobre diferentes conjuntos de datos dependiendo del movimiento de la unidad móvil. Falta una coordinación centralizada de las transacciones, en particular en el escenario (2) anterior. Además, lo más probable es que una transacción móvil tenga una larga duración debido a la desconexión que se produce en las unidades móviles. Según esto, es posible que las clásicas propiedades ACID de las transacciones (consulte el Capítulo 19) tengan que ser modificadas y que haya que definir nuevos modelos transaccionales.
- **Procesamiento de consultas.** Es importante conocer dónde se encuentran ubicados los datos, ya que afecta al análisis de coste/beneficio en el procesamiento de las consultas. La optimización de consultas se vuelve más complicada debido a la movilidad y al rápido cambio de los recursos en las unidades móviles. La respuesta a las consultas tiene que ser devuelta a unidades móviles que aun encontrándose en tránsito o atravesando los límites entre celdas necesitan recibir resultados correctos y completos.
- **Recuperación y tolerancia a fallos.** El entorno de base de datos móvil debe enfrentarse a fallos en las unidades, en los soportes de datos, en las transacciones y en las comunicaciones. Los fallos en las unidades se deben frecuentemente a la limitación en la capacidad de las baterías. El apagado voluntario de una unidad móvil *no* debería ser tratado como fallo. Los fallos en transacciones son más frecuentes cuando una unidad móvil atraviesa de una celda a otra. El fallo en una unidad móvil provoca un particionado de la red y afecta a los algoritmos de encaminamiento.
- **Diseño de bases de datos móviles.** El problema de resolución de nombres globales en el tratamiento de las consultas se empeora debido a la movilidad y a las caídas frecuentes. El diseño de bases de datos móviles debe tener en cuenta diversos aspectos de la gestión de metadatos; por ejemplo, la actualización constante de la información de ubicación.
- **Servicio en función de la ubicación.** A medida que los clientes se desplazan, la información almacenada que depende de la ubicación puede quedarse obsoleta. En estos casos son importantes las técnicas de desahucio. Además, la actualización de consultas dependientes de la ubicación y la posterior aplicación de estas consultas (espaciales) para refrescar la caché se convierte también en un problema.
- **División del trabajo.** Determinadas características de los entornos móviles obligan a cambiar la división del trabajo en el procesamiento de las consultas. En algunos casos, el cliente debe trabajar de forma independiente del servidor. Sin embargo, ¿cuáles son las consecuencias de permitir un acceso completamente independiente a datos replicados? Aún está por determinar la relación entre las posibles funcionalidades a nivel de cliente y sus consecuencias.
- **Seguridad.** Los datos móviles son menos seguros que los que permanecen en ubicaciones fijas. Las técnicas adecuadas de gestión y autorización de acceso a los datos críticos resultan más importantes en estos entornos. Los datos son también más volátiles y es necesario desarrollar técnicas para compensar su posible pérdida.

### 30.1.4 Aplicación. Bases de datos sincronizadas intermitentemente

El escenario de computación móvil se está haciendo cada vez más popular a medida que las personas se llevan el trabajo lejos de sus oficinas y hogares y llevan a cabo una gran cantidad de actividades y funciones: todo tipo de ventas, particularmente en productos farmacéuticos, artículos de consumo y el sector industrial; defensa de la ley; consultoría y planificación financiera y de seguros; bienes inmuebles o actividades de gestión de la propiedad; servicios de mensajería y transportes; etc. En estas aplicaciones un servidor, o un grupo de ellos, administran la base de datos central y los clientes portan portátiles o *palmtops* con el software DBMS residente para llevar a cabo la mayor parte de la actividad de transacción *local*.

Los clientes conectan a través de una red o de una conexión por marcación telefónica (o, posiblemente, a través de Internet) con el servidor, normalmente para una sesión corta; por ejemplo, 30 a 60 minutos. Envían actualizaciones al servidor y éste, a su vez, entra en ellos en su base de datos central, que debe mantener los datos actualizados y preparar copias apropiadas para todos los clientes del sistema. De este modo, siempre que los clientes conectan (a través de un proceso conocido como sincronización de un cliente con un servidor), reciben un lote de actualizaciones para que las instalen en sus bases de datos locales. La principal característica de este escenario es que los clientes están en su mayor parte desconectados; el servidor no tiene por qué alcanzar necesariamente al cliente. Este entorno tiene unos problemas parecidos a los de las bases de datos distribuidas y cliente/servidor, y otros propios de las bases de datos móviles, pero presenta varios problemas de investigación adicionales que hay que investigar. A este entorno nos referimos como **Entorno de base de datos intermitentemente sincronizada** (ISDBE, *Intermittently Synchronized Database Environment*), y a las bases de datos correspondientes como Bases de datos intermitentemente sincronizadas (ISDBS, *Intermittently Synchronized Databases*).

En su conjunto, las siguientes características de las ISDBs las diferencian de las bases de datos móviles que hemos explicado anteriormente:

1. Un cliente conecta con el servidor cuando quiere recibir actualizaciones del mismo, enviarle actualizaciones, o procesar transacciones que necesitan datos no locales. Esta comunicación puede ser de tipo unidifusión (comunicación de uno a uno entre el servidor y el cliente) o de tipo multidifusión (un emisor o un servidor puede comunicar periódicamente con un conjunto de receptores o actualizar un grupo de clientes).
2. Un servidor no puede conectar con un cliente a voluntad.
3. Los problemas de las conexiones cliente inalámbricas frente a las cableadas y la conservación de la energía son generalmente inmateriales.
4. Un cliente es libre de gestionar sus propios datos y transacciones mientras está desconectado. También puede efectuar su propia recuperación en cierta medida.
5. Un cliente tiene varias formas de conectar con un servidor, y en caso de muchos servidores, puede elegir el servidor en particular con el que quiere conectar en base a la proximidad, los nodos de comunicación disponibles, los recursos disponibles, etcétera.

Debido a estas diferencias, hay una necesidad de tener en cuenta ciertos problemas relacionados con las ISDBs que son diferentes de los que normalmente surgen con los sistemas de bases de datos móviles. Nos referimos al diseño de base de datos del servidor para las bases de datos de servidor, la gestión de la consistencia y la sincronización entre el procesamiento de transacciones y actualizaciones de las bases de datos cliente y servidor, el uso eficaz del ancho de banda del servidor, y la consecución de la escalabilidad en los entornos ISDB.

# 30.2 Bases de datos multimedia

En los próximos años se espera que los sistemas de información multimedia dominen nuestra actividad diaria. Nuestras casas estarán cableadas para permitir grandes anchos de banda que permitan interactuar con aplicaciones multimedia interactivas. Nuestras televisiones de alta definición o nuestras estaciones de trabajo tendrán acceso a un gran número de bases de datos, incluyendo librerías digitales, que distribuirán inmensas cantidades de contenido multimedia de diferentes fuentes.

## 30.2.1 La naturaleza de los datos y las aplicaciones multimedia

En la Sección 24.3 tratamos los temas de modelado avanzado relacionado con la información multimedia. En el contexto de los objetos relacionales DBMS (ORDBMS) examinamos también el procesamiento de múlti-

ples tipos de datos en el Capítulo 22. Los DBMSs se han ido incorporando constantemente a los tipos de datos que soportan. En la actualidad, la mayoría de sistemas admiten los siguientes tipos de datos multimedia:

- **Texto.** Puede ser formateado o sin formatear. Para facilitar el análisis gramatical de documentos estructurados se emplean estándares como SGML y sus variantes (como HTML).
- **Gráficos.** Incluyen dibujos e ilustraciones que están codificados usando algún estándar descriptivo (por ejemplo, CGM, PICT, postscript).
- **Imágenes.** Incluyen dibujos, fotografías, etc. Están codificadas en los formatos estándar como Bitmap, JPEG y MPEG. La compresión se consigue en los formatos JPEG y MPEG. Estas imágenes no están subdivididas en componentes. Por tanto, preguntar por ellas mediante el contenido (por ejemplo, localizar todas las imágenes que contengan círculos) no es algo trivial.
- **Animaciones.** Secuencias temporales de imágenes o datos gráficos.
- **Vídeo.** Un conjunto de datos fotográficos secuenciados temporalmente que se representan según una velocidad especificada (por ejemplo, 30 fotogramas por segundo).
- **Audio estructurado.** Una secuencia de componentes de audio que contienen nota, tono, duración, etc.
- **Audio.** Muestra de datos tomados a partir de auriculares y grabados en una cadena de bits en formato digitalizado. Las grabaciones analógicas suelen transformarse a digitales antes de almacenarse.
- **Datos multimedia compuestos o mixtos.** Una combinación de distintos tipos de datos multimedia, como audio y vídeo, que pueden mezclarse físicamente para obtener un nuevo formato de almacenamiento, o lógicamente mientras se mantienen los tipos y formatos originales. Los datos compuestos contienen también información de control adicional que describe cómo debe mostrarse dicho dato.

**Naturaleza de las aplicaciones multimedia.** Los datos multimedia pueden ser almacenados, entregados y utilizados de muy diferentes formas. Las aplicaciones pueden categorizarse de la siguiente forma en función de sus características de administración de los datos:

- **Aplicaciones de almacén.** Existen muchos datos multimedia y metadatos que se almacenan sólo con propósitos de recuperarlos. Para ello, es preciso que un DBMS mantenga un almacén central de esos datos, donde estén organizados en niveles de almacenamiento jerárquicos (discos locales, discos terciarios y cintas, discos ópticos, etc.). Como ejemplos podemos citar los almacenes de imágenes de satélite, los diseños y dibujos de ingeniería, fotografías espaciales e imágenes radiológicas escaneadas.
- **Aplicaciones de presentación.** Un gran número de aplicaciones implican la entrega de datos multimedia sujeta a restricciones temporales. Tanto el audio como el vídeo son ejemplos de ello; en estas aplicaciones, el visionado óptimo o las condiciones de audición requieren que el DBMS entregue la información sujeta a ciertas velocidades que ofrezcan cierta *calidad de servicio* por encima de cierto umbral. Los datos se consumen a medida que se envían, al contrario que ocurre con las aplicaciones de almacén donde pueden procesarse más adelante (por ejemplo, el correo electrónico multimedia). El simple visionado de un vídeo multimedia, por ejemplo, precisa de un sistema que simule la funcionalidad VCR. Otras presentaciones multimedia complejas e interactivas implican una cierta instrumentación para controlar el orden de recuperación de los componentes en serie o en paralelo. Los entornos Interactivos deben soportar ciertas habilidades como el análisis de la edición en tiempo real o la anotación de datos de vídeo y audio.
- **Trabajo cooperativo usando información multimedia.** Ésta es una nueva categoría de aplicaciones en la que los ingenieros podrían ejecutar una tarea de diseño compleja mediante la fusión de dibujos, la adaptación de materias a restricciones de diseño, la generación de nueva documentación, el cambio de notificaciones, etc. Las redes de salud inteligentes así como la telemedicina provocarán que los doctores colaboren entre sí, analizando los datos y la información multimedia de un paciente en tiempo real a medida que se generan.

Todas estas áreas presentan los mayores desafíos para el diseño de sistemas de bases de datos multimedia.

### 30.2.2 Cuestiones relativas a la administración de los datos

Las aplicaciones multimedia tratan con cientos de imágenes, documentos, segmentos de audio y vídeo y texto libre que dependen de una forma crucial del modelado apropiado de la estructura y del contenido de los datos para después diseñar los esquemas de base de datos adecuados para almacenar y recuperar información multimedia. Estos sistemas son complejos y abarcan un gran número de temas, entre los que se pueden citar los siguientes:

- **Modelado.** Esta área tiene el potencial de aplicar al problema las técnicas de recuperación de información frente a las de base de datos. Existen problemas en relación a objetos complejos (consulte el Capítulo 20) conformados por un amplio rango de tipos de datos: numérico, texto, gráfico (imagen generada por computador), imagen gráfica animada, *stream* de audio y secuencia de vídeo. Los documentos constituyen un área especializada y son merecedores de una consideración especial.
- **Diseño.** Los diseños conceptual, lógico y físico de las bases de datos multimedia no han sido totalmente solucionados, por lo que sigue siendo un área activa de investigación. El proceso de diseño puede estar basado en la metodología general descrita en el Capítulo 12, aunque los problemas de rendimiento y de ajuste fino en cada nivel son mucho más complejos.
- **Almacenamiento.** El almacenamiento de datos multimedia en dispositivos estándares como discos presenta problemas de representación, compresión, asignación a jerarquías de dispositivo, archivado y *buffering* durante la operación de entrada/salida. Un modo que tienen los distribuidores de productos multimedia de afrontar este tema es adhiriéndose a estándares como JPEG o MPEG. En los DBMSs, los "BLOB" (Objetos binarios grandes) permiten almacenar y recuperar *bitmaps* sin tipo. Para tratar con la sincronización y la compresión/descompresión se precisará de software estandarizado, y estará asociado con los problemas de los índices, que aún se encuentran en fase de investigación.
- **Consultas y recuperación.** La forma en que una *base de datos* recupera información está basada en los lenguajes de consulta y en estructuras de índice internas. La forma de *recuperar la información* confía estrictamente en palabras clave o en términos de índice predefinidos. Para el caso de las imágenes, el vídeo y el audio se abre un enorme abanico de problemas, entre los que se pueden citar la formulación eficiente de la consulta, su ejecución y su optimización. Las técnicas estándar de optimización comentadas en el Capítulo 16 deben modificarse para trabajar con tipos de datos multimedia.
- **Rendimiento.** Para las aplicaciones multimedia que sólo trabajan con documentos y texto, las restricciones de rendimiento están subjetivamente determinadas por el usuario. En aplicaciones que implican el uso de *playback* de vídeo o la sincronización audio-vídeo, las limitaciones físicas son las que predominan. Por ejemplo, el vídeo debe reproducirse a una velocidad fija de 30 fotogramas por segundo. Las técnicas de optimización de consultas pueden computar tiempos de respuesta esperados antes de evaluar la consulta. El uso del procesamiento paralelo de datos puede aliviar ciertos problemas, aunque es una técnica que se encuentra en fase de experimentación.

Todo esto ha originado una gran variedad de problemas de investigación. Veamos algunos de los más representativos.

### 30.2.3 Problemas de investigación abierta

**Perspectiva de la recuperación de información en la consulta de bases de datos multimedia.** El modelado del contenido de datos no ha sido un tema en los modelos y sistemas de bases de datos porque los datos cuentan con una estructura rígida y es posible inferir una de sus instancias a partir del esquema. Por el contrario, la IR (Recuperación de la información, *Information Retrieval*) es un tema importante a la hora

de modelar el contenido de los documentos de texto (mediante el uso de palabras clave, índices relativos a la frase, redes semánticas, frecuencias de palabras, codificación Soundex, etc.) para los que la estructura dentro del objeto multimedia suele estar abandonada. Modelando el contenido, el sistema puede determinar si un documento es relevante para una consulta examinando sus descriptores de contenido. Consideremos, por ejemplo, un informe de reclamación de un accidente de una compañía de seguros como objeto multimedia: dicho informe contiene imágenes del accidente, formularios de seguros estructurados, grabaciones de audio de las partes implicadas, el informe del perito de la compañía, etc. ¿Qué modelo de datos debería usarse para representar información multimedia como ésta? ¿Cómo deberían formularse las consultas contra estos datos? La ejecución eficiente de las mismas se convierte en un tema complejo, y la heterogeneidad de la semántica y la complejidad representativa de la información multimedia dan origen a nuevos problemas.

**Requerimientos del modelado y la recuperación de datos multimedia/hipermedia.** Para capturar toda la potencia expresiva del modelado de datos multimedia, el sistema debe contar con una construcción general que permita al usuario especificar enlaces entre dos nodos arbitrarios. Los **enlaces hipermedia**, o hiperenlaces, tienen numerosas características diferentes:

- Los enlaces pueden especificarse con o sin información asociada, y pueden tener grandes descripciones asociadas a ellos.
- Los enlaces pueden empezar desde un punto específico del nodo o desde todo él.
- Los enlaces pueden ser direccionales o sin dirección cuando pueden atravesarse en cualquier sentido.

La capacidad de enlace del modelo de datos debe tener en cuenta todas estas variaciones. Cuando se precisa de una recuperación de datos multimedia basada en el contenido, el mecanismo de consulta debe tener acceso a los enlaces y a la información asociada a los mismos. El sistema debe proporcionar facilidades para definir vistas sobre todos los enlaces, ya sean privados o públicos. La información contextual valiosa puede obtenerse a partir de la información estructural. Los enlaces hipermedia generados automáticamente no revelan nada acerca de los dos nodos, y a diferencia de lo que ocurre con los desarrollados manualmente, podrían tener significados diferentes. Las posibilidades de creación y uso de este tipo de enlaces, así como el desarrollo y utilización de lenguajes de consulta navegables para utilizar estos enlaces, son rasgos importantes de cualquier sistema para permitir un uso efectivo de la información multimedia. Esta área es significativa para las bases de datos web entrelazadas.

La World Wide Web presenta una oportunidad de acceder a una inmensa cantidad de información mediante un *array* de bases de datos no estructuradas y estructuradas que están interconectadas. El increíble éxito y crecimiento de la Web ha convertido el problema de la localización, acceso y mantenimiento de toda esta información en todo un desafío. Durante los últimos años, varios proyectos han intentado definir armazones y lenguajes que nos permitieran definir el contenido semántico de la Web de forma que esté bien definido y sea procesable por la máquina. El esfuerzo es conocido colectivamente con el término **Semantic Web**. RDF (Infraestructura de descripción de recursos, *Resource Description Framework*), XHTML (Lenguaje de marcado de hipertexto extensible, *eXtensible HyperText Markup Language*), DAML (Lenguaje de marcado de agente DARPA, *DARPA Agent Markup Language*) y OIL (Capa de inferencia ontológica, *Ontology Inference Layer*) son algunos de sus principales componentes.[3] Una explicación exhaustiva queda fuera del objetivo de este debate.

**Indexación de imágenes.** Existen dos métodos de indexación de imágenes: identificar los objetos automáticamente mediante técnicas de procesamiento de imágenes, y asignar términos y frases índice mediante indexación manual. Existe un importante problema con la escalabilidad cuando se utilizan técnicas de procesamiento de imágenes. El estado actual de la técnica sólo permite indexar patrones simples en las imágenes. La complejidad incrementa con el número de características reconocibles. Otro importante problema se

[3] Consulte Fensel (2000) si desea una visión general de estos términos.

refiere a la complejidad de la consulta. Las reglas y los mecanismos de inferencia pueden usarse para derivar hechos de nivel más alto a partir de características simples de las imágenes. De forma similar, puede usarse la abstracción para capturar conceptos que no podrían definirse mediante conjuntos de pares <atributo, valor>. Esto permite generar consultas de alto nivel del tipo "localizar los hoteles que tienen vestíbulos abiertos y permiten una máxima luz solar en la zona del escritorio" en una aplicación arquitectónica.

La recuperación de información en la indexación de imágenes está basada en uno de estos tres esquemas:

1. **Sistemas clasificatorios.** Clasifica las imágenes jerárquicamente en categorías predeterminadas. En este planteamiento, el indexador y el usuario deben tener un buen conocimiento de estas categorías. Los detalles más finos de una imagen compleja y las relaciones existentes entre los objetos de la misma no pueden capturarse.
2. **Sistemas basados en palabras clave.** Utiliza un vocabulario de indexación similar al usado en la indexación de documentos de texto. Los hechos simples representados en la imagen (como la *región*) y los derivados como resultado de una interpretación de alto nivel de los humanos (como hielo permanente, nevada reciente y hielo polar) pueden ser capturados.
3. **Sistemas entidad-atributo-relación.** Se identifican todos los objetos de la imagen y las relaciones existentes entre los objetos y sus atributos.

En el caso de documentos de texto, un indexador puede elegir la palabras clave del almacén de palabras disponibles en el documento. Esto no es posible en el caso de datos visuales y de vídeo.

**Problemas en la recuperación de texto.** La recuperación de texto siempre ha sido la clave en las aplicaciones de negocio y sistemas de librería, y aunque se ha avanzado mucho en algunos de los problemas que veremos a continuación, aún quedan muchas mejoras por realizar, especialmente en las siguientes áreas:

- **Indexación de frase.** Es posible lograr mejoras sustanciales si se asignan descriptores de frase a los documentos y se utilizan en las consultas, ya que estos descriptores son un buen indicador del contenido del documento y de la información necesaria.
- **Uso de diccionarios de sinónimos (tesauros).** Una de las razones de la poca capacidad de recuperación de los sistemas actuales es la diferencia entre el vocabulario de los usuarios y el utilizado a la hora de indexar los documentos. Una de las soluciones es utilizar un tesauro para ampliar la consulta del usuario con términos relacionados. El problema entonces se plantea a la hora de localizar un tesauro para el dominio que nos interesa. Otro recurso en este contexto son las **ontologías**. Una ontología necesariamente comporta, o conlleva, alguna clase de visión del mundo en relación al dominio dado. La visión del mundo es con frecuencia concebida como un conjunto de conceptos (por ejemplo, entidades, atributos, procesos), sus definiciones y sus interrelaciones que describen el mundo objetivo. Una ontología puede construirse de dos formas: dependiente del dominio y genérica. El objetivo de las ontologías genéricas es construir una infraestructura general para todas (o casi todas) las categorías encontradas por la existencia humana. Ya se han construido algunas ontologías como la genética (consulte la Sección 29.4) y la de componentes electrónicos.[4]
- **Resolución de la ambigüedad.** Una de las razones de la baja precisión (la proporción del número de elementos relevantes recuperados con relación al número total de los mismos) de los sistemas de recuperación de información de texto es que las palabras tienen significados múltiples y contextuales. Una de las formas de resolver la ambigüedad es utilizar un diccionario *online* o una ontología; otra es comparar los contextos en los que se encuentran dos palabras.

En las tres primeras décadas del desarrollo de los DBMSs (entre 1965 y 1995) el foco principal ha estado en la administración de la enorme cantidad de datos de negocio e industriales. En las próximas décadas, la infor-

[4] En Uschold y Gruninger (1996) puede encontrar un buen debate sobre las ontologías.

mación textual no numérica dominará el contenido de las bases de datos. El problema de la recuperación de texto se está poniendo cada vez más de relieve en el entorno de los documentos HTML y XML. La Web contiene actualmente miles de millones de estas páginas. Los motores de búsqueda localizan los documentos relevantes en función de una lista de palabras, lo que supone una forma libre de lenguaje de consulta natural. La obtención del resultado correcto que cumple los requerimientos de precisión (% de documentos recuperados que son relevantes) y retentiva (% del total de documentos relevantes que son recuperados), los cuales son la métrica estándar en la recuperación de información, sigue siendo todo un desafío. Como consecuencia de ello, los DBMSs verán ampliadas sus funciones con la incorporación de nuevas funcionalidades como la comparación, la conceptualización, el entendimiento, la indexación y el resumen de documentos. Los sistemas de información multimedia prometen unificar disciplinas que históricamente han estado separadas: la recuperación de información y la administración de una base de datos.

### 30.2.4 Aplicaciones de bases de datos multimedia

Las aplicaciones de bases de datos multimedia a gran escala deben abarcar un gran número de disciplinas y mejorar el potencial ya existente. Algunas de las aplicaciones importantes implicadas serán:

- **Administración de documentos y registros.** Un gran número de industrias y negocios mantienen registros muy detallados y una amplia variedad de documentos. Los datos pueden incluir diseños de ingeniería y datos de fabricación, registros médicos de pacientes, publicaciones y reclamaciones a compañías de seguros, por citar algunos.
- **Diseminación del conocimiento.** El modo multimedia, un medio eficaz de diseminación del conocimiento, tendrá un fantástico crecimiento en libros electrónicos, catálogos, manuales, enciclopedias y almacenes de información acerca de muchos temas.
- **Educación y aprendizaje.** El aprendizaje asistido por computador y los materiales destinados a diferentes audiencias (desde estudiantes de guardería a profesionales) pueden diseñarse a partir de fuentes multimedia. Se espera que las librerías digitales tengan una importancia clave en el modo en que los futuros estudiantes, investigadores y personas de todo tipo accedan a los inmensos almacenes de material educativo.
- **Marketing, publicidad, ventas, entretenimiento y viajes.** Virtualmente, no existe límite en la forma de utilizar la información multimedia en estas aplicaciones (desde efectivas presentaciones de ventas a circuitos virtuales por ciudades y galerías). La industria cinematográfica ya ha mostrado la potencia de los efectos especiales en la creación de animaciones y de animales o alienígenas diseñados sintéticamente. El uso de objetos prediseñados almacenados en bases de datos multimedia expandirán el rango de estas aplicaciones.
- **Control en tiempo real y monitorización.** Asociada a las tecnologías de bases de datos activas (consulte el Capítulo 24), la presentación multimedia de información puede suponer una forma muy efectiva de monitorizar y controlar tareas complejas como las operaciones de fabricación, las plantas de energía nuclear, los pacientes en unidades de cuidados intensivos y los sistemas de transporte.

**Sistemas comerciales para la administración de Información multimedia.** No existe ningún DBMS diseñado en exclusiva para controlar datos multimedia, por lo que no hay nada que tenga todas las funcionalidades necesarias para dar un soporte completo a todas estas aplicaciones. Sin embargo, algunos DBMSs soportan tipos de datos multimedia; podemos citar Informix Dynamic Server, UDB (Base de datos universal DB2, *DB2 Universal Database*) de IBM, Oracle 9 y 10, CA-JASMINE, SYBASE y ODB II. Todos ellos soportan objetos, lo que resulta esencial pata modelar los distintos tipos de elementos multimedia complejos. Uno de los principales problemas de estos sistemas es que "palas de datos, los cartuchos y las extensiones" para manipular datos multimedia están diseñados de una manera muy específica. La funcionalidad se

entrega sin tener mucho en cuenta la escalabilidad y el rendimiento. Existen productos que operan en solitario o en combinación con los sistemas de otros fabricantes para permitir la recuperación de los datos de una imagen por su contenido. Entre ellos podemos citar Virage, Excalibur y QBIC de IBM. Las operaciones multimedia deben ser estandardizadas. MPEG-7 y otros estándares están diseñados para resolver algunos de estos problemas.

Muchos sistemas de bases de datos han sido construidos y "prototipados" con muy diversos grados de soporte de las funciones definidas por el usuario. Como ejemplo notable, el sistema QBIC (Consulta por contenido de imagen, *Query By Image Content*) de IBM Almaden permite realizar consultas usando atributos como los colores, las texturas, las formas y la posición de los objetos. En una base de datos llena de imágenes, los métodos identifican objetos en imágenes, segmentos de vídeo en pequeñas secuencias llamadas *shots* y características de procesamiento relacionadas con los atributos anteriores. Algunos sistemas de base de datos sensor (por ejemplo, COUGAR de Cornell) mezclan datos almacenados representados como relaciones y datos sensor representados como series de tiempo. Cada consulta sensor de larga ejecución define una vista persistente que se mantiene durante un periodo determinado. El sistema de administración de la base de datos de vídeo VDBMS está diseñado para soportar un amplio abanico de funciones de vídeo, así como un tipo de dato abstracto bien definido. Los descriptores semánticos y visuales (incluyendo características visuales de bajo nivel como el histograma de color, la textura y la orientación límite) que representan e indexan contenido de vídeo para su búsqueda, se extraen durante el procesamiento del mismo y se almacenan en tablas separadas en la base de datos para que la búsqueda pueda hacerse por contenido.

## 30.3 GIS (Sistemas de información geográfica, *Geographic Information Systems*)[5]

Los GIS[6] suelen estar definidos como una *integración sistemática de hardware y software para la captura, el almacenamiento, la visualización, la actualización y el análisis de datos espaciales*. Durante la década de los 60 y 70, los GIS no eran más que una simple herramienta software integrada en un computador que permitía resolver problemas espaciales con relativa facilidad. Con el tiempo, los GIS expandieron su significado y pasaron a ser considerados como una Ciencia de Información Geográfica en lugar de como una herramienta o un sistema. Esta noción la introdujo Goodchild (1992): *la manipulación de la información espacial mediante la tecnología GIS presenta un abanico de desafíos intelectuales y científicos mucho mayores de los que implica la frase 'manipulacion de datos espaciales'; en efecto, una ciencia de información geográfica.* Goodchild mostró las diferentes cuestiones intelectuales y científicas que aparecían durante el proceso de resolución de un problema en GIS: cuestiones que implican la determinación de la exactitud de un mapa, diseños de modelos de datos geográficos dependientes del tiempo y la elaboración de mejores herramientas y métodos de conversión.

GIS puede verse también como un área interdisciplinaria que incorpora muchos campos de estudio diferentes, como la Geodesia (proyecciones, vigilancia, cartografía, etc.), la Sensación Remota, la Fotogravimetría, la Ciencia Medioambiental, la Planificación de ciudades, la Ciencia Cognitiva y otras. Como resultado, GIS se basa en los avances realizados en campos como la informática, las bases de datos, la estadística y la inteligencia artificial. Los diferentes problemas y preguntas que emanan de la integración de múltiples disciplinas hacen de él algo más que una simple herramienta.

Comentamos algunas características de los datos espaciales en la Sección 24.3.1. Pero en ésta nuestro interés irá encaminado a describir los desafíos planteados por GIS como uno de los dominios de aplicación emergentes para la tecnología de bases de datos.

---

[5] Agradecemos la importante contribución de Liora Sahar a esta sección.

[6] Utilizaremos la abreviatura GIS para referirnos a sistema o sistemas, en función del contexto.

La manipulación de información geográfica plantea una cuestión básica: *¿Por qué es especial lo espacial?*

La información geográfica/espacial es especial porque describe los objetos que tienen una posición distinta (atributos y relaciones entre sí dentro de un espacio definido). La primera ley de la declaración geográfica dice que *todo está relacionado con todo, pero las cosas próximas lo están más que las lejanas*. Esta ley obliga a los sistemas GIS a permitir análisis basados en la topología entre objetos. La topología requiere de la definición de adyacencia, proximidad, conectividad y otras relaciones entre características.

La información geográfica también precisa de un modo de integración entre diferentes fuentes de datos a diferentes niveles de precisión. Ya que el sistema trata con aspectos de nuestra vida diaria, debe actualizarse regularmente para mantenerlo actualizado y fiable. Mucha de la información para uso práctico almacenada en las bases de datos GIS requiere una forma especial de recuperación y manipulación. Los sistemas GIS y las aplicaciones tratan con información que puede ser visionada como un dato con significado y contexto específicos en lugar de como un simple dato.

## 30.3.1 Componentes de los sistemas GIS

Los sistemas GIS pueden considerarse como una integración de tres componentes: hardware y software, datos y personas.

- **Hardware y software.** El hardware está relacionado con los dispositivos utilizados por los usuarios finales, como los dispositivos gráficos o los *plotters* y los escáneres. La manipulación y el almacenamiento de los datos se consigue utilizando varios procesadores. Con el desarrollo de Internet y las aplicaciones web, los servidores web se han convertido en parte de muchas arquitecturas, por lo que los GIS siguen la arquitectura de tres niveles (consulte la Figura 2.7). La parte software está relacionada con los procesos usados para definir, almacenar y manipular los datos, por lo que es muy similar a un DBMS. Para conseguir formas eficientes de almacenamiento, recuperación y manipulación de los datos se han propuesto muchos modelos.

- **Datos.** Los datos geográficos pueden dividirse en dos grupos principales: vectores y rasterizaciones.

  Las capas vectoriales en GIS se refieren a objetos discretos representados por puntos, líneas y polígonos. Las líneas están formadas por la conexión de uno o más puntos, mientras que los polígonos son conjuntos cerrados de líneas. Las capas representan geometrías que comparten un conjunto de atributos. Los objetos que están dentro de una capa tienen definida una topología mutua entre ellos (este tema lo trataremos más adelante). Los procesos para convertir un vector en un *raster* y viceversa forman parte de muchas implementaciones GIS.

  El *raster* es una rejilla continua de celdas en dos dimensiones o el equivalente a *voxels* (celdas cúbicas) en tres dimensiones. Los conjuntos de datos *raster* están divididos conceptualmente en categóricos y continuos. En un *raster* categórico, cada celda está enlazada a una categoría en una tabla separada. Como ejemplos podemos citar los tipos de suelo, los de vegetación y la idoneidad de la tierra. Las imágenes rasterizadas continuas suelen describir fenómenos continuos en el espacio como los DEM (Modelos de elevación digital, *Digital Elevation Models*), donde cada píxel es un valor de elevación. A diferencia de los categóricos, los *raster* continuos no tienen asociada una tabla de tipo atributo/categoría.

  Existe un gran rango de fuentes de datos para los datos de tipo vector y *raster*. Entre los primeros podemos citar mapas digitalizados, rasgos geográficos extraídos del examen de mapas, etc. Entre las fuentes *raster* típicas tenemos las imágenes aéreas (escaneadas o generadas digitalmente), las imágenes de satélites (como el Landsat, Spot e Ikonos) y las imágenes escaneadas de mapas.

- **Personas.** Las personas están implicadas en todas las fases del desarrollo de un sistema GIS y en la recopilación de los datos. Entre ellos se incluyen los cartógrafos y estudiosos que crean los mapas y

estudian el terreno, y los usuarios del sistema que recopilan los datos, los cargan en él, lo manipulan y analizan los resultados.

### 30.3.2 Características de los datos en GIS

Existen características particulares de los datos geográficos que hacen que su modelado sea más complicado que en las aplicaciones convencionales. El contexto geográfico, las relaciones topológicas y otras relaciones espaciales son fundamentalmente importantes para definir las reglas de integridad espacial. Es preciso considerar varios aspectos de los objetos geográficos (Friis-Christensen y Jensen 2001). Los resumiremos a continuación:

- **Localización.** La localización espacial de rasgos geográficos está definida por coordenadas en un sistema de referencia específico. Esos rasgos están representados por puntos, líneas o polígonos, y su geometría hace referencia a las representación tridimensional en el espacio.
- **Temporalidad.** El modelo de base de datos debe considerar tanto la existencia como el cambio a lo largo del tiempo de los rasgos. Esto es especialmente crucial cuando se habla de datos dinámicos, como parcelas de tierra, ya que tenemos que representar datos válidos y actualizados.
- **Rasgos espaciales complejos.** Los rasgos abarcan varias representaciones espaciales que incluyen puntos, líneas, polígonos y *rasters*. La representación compleja permite asociar, por ejemplo, un objeto tridimensional con diferentes polígonos de sus facetas.
- **Valores temáticos.** Las diferentes propiedades y cualidades de un objeto pueden ser representadas mediante sus atributos.
- **Objetos ambiguos.** La ambigüedad tiene que ver con la incertidumbre a la hora de ubicar y clasificar temáticamente un objeto. La localización del objeto está representada por coordenadas y está asociada con un margen de error. El aspecto temático se representa relacionando un objeto a una clase con un cierto grado o porcentaje de certeza. Las provincias o las corporaciones municipales siempre están en movimiento desplazando límites, carreteras y construyendo edificaciones, por lo que nunca se puede garantizar al cien por cien que estas bases de datos sean fiables en términos de rasgos topológicos.
- **Entidad frente a datos basados en campos.** El mundo puede representarse como un conjunto de entidades discretas como bosques, ríos, carreteras y edificios. Todo este conjunto recibe el nombre de *aproximación basada en entidades*. El *planteamiento basado en campos* representa el mundo como una función continua con atributos que varía en el espacio. Los fenómenos naturales como la distribución de la contaminación del aire y del terreno puede representarse mejor usando este planteamiento.
- **Generalización.** La generalización está relacionada con el nivel de escala y de detalle asociados al objeto. Los objetos pueden incorporarse con escalas más grandes o más pequeñas, mientras que el proceso contrario está muy limitado. Por ejemplo, la capa de provincias puede añadirse a la de países, pero no al contrario sin contar con datos externos. Pueden usarse imágenes de una resolución de 2 metros por píxel para generar otra de 10 metros, mientras que realizar la operación al contrario no añade ningún detalle a la imagen y sólo es una división sintética del original.
- **Roles.** Un objeto perteneciente a un modelo de datos puede asumir diferentes roles en función del *universo de discurso*. Por tanto, el rol es *dependiente de la aplicación*.
- **ID de objeto.** Los objetos deben estar identificados de forma inequívoca dentro del modelo de datos. Mas aún, para el intercambio de datos entre organizaciones deben existir IDs de objeto universales.
- **Calidad de los datos.** La calidad de los datos se refiere a la credibilidad y la fiabilidad de los mismos, o dicho de un modo más general, *lo bueno que es el dato*. La *calidad cuantitativa* está relacionada con componentes medibles como la precisión espacial (el error en la posición del objeto). La *calidad cua-*

*litativa* afecta a componentes no medibles, los cuales están relacionados con el conjunto completo de datos y no con objetos específicos.

**Restricciones en GIS.** Las restricciones son un aspecto muy importante de los objetos geográficos en GIS. Las típicas restricciones de integridad de clave, de dominio, referencial y semánticas generales (consulte las Secciones 5.2.4 y 5.2.5) no pueden capturar las características distintivas de la información geográfica. Es posible relacionar una provincia con una comunidad autónoma usando una clave externa, pero, ¿cómo podemos imponer la localización de la provincia dentro de los límites de la comunidad? Las restricciones que tratan específicamente con información espacial pueden clasificarse en topológicas, semánticas y definidas por el usuario (Cockcroft, 1997):

- **Restricciones de integridad topológicas.** La topología se encarga del comportamiento de los rasgos geográficos y de las relaciones espaciales existentes entre ellos. Las restricciones de integridad topológicas se utilizan durante los procesos de inserción y actualización para ayudar a reducir el margen de error y mejorar la calidad de los datos. Las relaciones espaciales están definidas por entidades dentro del modelo de datos y son específicas de la aplicación. Cuando, por ejemplo, se subdivide una parcela, podemos obligar a que las nuevas sub-parcelas se encuentren dentro de los límites de la original.
- **Restricciones de integridad semánticas.** La semántica trata con el "significado" del rasgo espacial. Las restricciones de integridad semántica definen si el estado de la base de datos es válido de acuerdo a las reglas semánticas aplicadas a los rasgos. Por ejemplo, el límite de una parcela no puede cruzar el contorno de un edificio y dos edificaciones no pueden compartir un límite. Algunas de las restricciones semánticas aplicadas a las carreteras, como las calles de una única dirección, pueden ser complicadas de incorporar.
- **Restricciones de integridad definidas por el usuario.** Las restricciones de integridad definidas por el usuario son las *reglas de negocio* de los DMBS no espaciales. Un ejemplo de este tipo de reglas podría ser que una casa debe estar localizada a una cierta distancia de una boca de incendios.
- **Restricciones temporales.** Este tipo de restricciones están caracterizadas por su carácter puntual y duradero en el tiempo. La temporalidad puntual define el suceso del evento con un tiempo específico, como un accidente, un terremoto, una fiesta, etc. La durabilidad se refiere a la descripción de los cambios que ocurren a lo largo del tiempo (Brodeur y otros, 2000). Por ejemplo, un puente se construyó, se mantuvo y se demolió. Por consiguiente, además del cambio en la localización/geometría, el dato debe tener adjunto un componente de tiempo.

Otra restricción a tener en cuenta es la **restricción de generalización**. Esta restricción hace referencia a los distintos niveles de escala y abstracción existentes dentro en un sistema. Cuando se actualiza una determinada capa de datos dentro de un sistema, debe satisfacer distintas restricciones. ¿Cómo deben modificarse estas restricciones, o permanecer inalteradas, cuando se crean otros niveles de escala? Por ejemplo, la geometría de un edificio aparecerá diferente (o desaparecerá por completo) en una escala grande (1:1.000) frente a otra más pequeña (1:100K). Las restricciones que eran válidas en un nivel pueden ser irrelevantes en otro. Este problema ha sido abordado en diferentes proyectos de investigación, pero aún continúa sin solución, especialmente cuando hablamos de la creación automática de los niveles de datos y de la agregación.

### 30.3.3 Modelos de datos conceptuales para GIS

Esta sección describe brevemente los modelos conceptuales más comunes para el almacenamiento de datos espaciales en GIS. Cada uno de estos modelos está implementado actualmente en los Sistemas Geográficos, y los más conocidos son los siguientes:

- **Modelo de datos *raster*.** El dato *raster* es un array de celdas, donde cada una de ellas representa un atributo. Los metadatos suelen estar almacenados en un fichero de cabecera que suele incluir las coor-

denadas geográficas de la celda superior izquierda de la rejilla, el tamaño de la celda y el número de filas y columnas. Este modelo es muy utilizado en aplicaciones analíticas como el modelado, el álgebra de mapas y las técnicas más avanzadas de extracción y agrupamiento de rasgos geográficos.

- **Modelo de datos vectorial.** El dato vectorial es la representación discreta de un objeto, principalmente como puntos, líneas y polígonos. Los rasgos geográficos pueden tener establecidas relaciones entre sí (topología) e incorporar un comportamiento. El análisis espacial se lleva a cabo usando un conjunto bien conocido de herramientas para analizar el dato espacial y obtener información de los diferentes niveles de toma de decisión.
- **Modelo de red.** La red es esencialmente un tipo especial de modelo de rasgo topológico que permite el modelado de flujo y el transporte de recursos y servicios. La relación de la topología de red define cómo se conectan las líneas entre sí en los puntos de intersección (nodos). Estas reglas se almacenan en tablas de conectividad. Muchas aplicaciones de la vida diaria utilizan el modelo de red como la optimización de la ruta de un autobús escolar, la planificación de rutas de emergencia, el diseño de sistemas de flujo y la creación de modelos de transporte.
- **Modelo de datos TIN.** Las TIN (Redes irregulares trianguladas, *Triangulated Irregular Networks*) se utilizan para crear y representar superficies. En la estructura TIN, esa superficie aparece como triángulos no solapados construidos a partir de puntos espaciados irregularmente. La densidad de los puntos viene determinada por los cambios de relieve de la superficie modelada. La estructura de datos TIN manipula la información relativa a los nodos que contiene cada triángulo y sus vecinos. Existen varias ventajas importantes en el modelo TIN. En un modelo *raster* DTM, cada píxel tiene un valor de elevación. Algunos de estos valores pueden ser las medidas originales, mientras que otros son el resultado de una interpolación. En un modelo TIN, desde que no existe limitación a la hora de representar el valor de un píxel o localización específicos, se mantienen los puntos medidos originalmente. La densidad de los puntos no es fija, sino que cambia de acuerdo al relieve de la superficie. Por tanto, el modelo puede mantener aquellos puntos que mejor lo representan. Un menor número de puntos a controlar redunda en un almacenamiento, análisis y administración más eficaces de los datos. El modelo TIN es también el más sencillo para calcular elevaciones, inclinaciones y la línea de observación entre puntos.

### 30.3.4 Mejoras DBMS para GIS

Hasta mediados de la década de los 90, los sistemas de información geográfica estaban basados principalmente en modelos de datos propietarios. Estos modelos estaban basados en ficheros y optimizados para conseguir un acceso rápido y eficiente. Esta propiedad provocaba un evidente problema a la hora de compartir datos entre distintas organizaciones. No existían estándares para la transferencia, lo que limitaba a los fabricantes en sus opciones para compartir. Los modelos GIS evolucionaron hacia estructuras geo-relacionales y el atributo fue almacenado en una base de datos relacional. Los rasgos espaciales se hospedaron en un sistema de ficheros y enlazados a los atributos. Aunque este modelo se benefició de los RDBMSs, seguía limitado en su escalabilidad y no podía beneficiarse de todas las posibilidades de los sistemas relacionales comerciales como la copia de seguridad y la recuperación, la replicación o la seguridad, ya que una gran parte de la información (los datos espaciales) seguían almacenados en un sistema de ficheros propietario.

En los 90, se diseñaron las bases de datos basadas en objetos para paliar las debilidades de los RDBMSs. Un único RDBMS no era capaz de tratar con tipos de datos abundantes, como los geográficos, y presentaban problemas cuando se enfrentaban a datos GIS. La nueva tecnología permitió el almacenamiento persistente de objetos (así como el transitorio) y los métodos relacionados. Los sistemas OODBMSs permitieron al usuario definir entidades naturales y complejas y ofrecieron los conceptos y opciones resultantes del método OO (consulte el Capítulo 20). Los OODBMSs permitieron la definición y el almacenamiento de información estructurada y no estructura, como imágenes de satélite, mediante los BLOBs. Pero a pesar de todas estas ventajas, los OODBMSs no tuvieron éxito en el mercado.

El siguiente paso supuso el diseño de los ORDBMSs (Sistemas de administración de bases de datos de objetos relacionales, *Object Relational Database Management Systems*) para dar soporte a las aplicaciones GIS. Estos sistemas eran RDBMSs adaptados para tratar con los objetos de almacenamiento de rasgos específicos de localización, su comportamiento y sus atributos (habitualmente definidos como objetos simples) dentro del mismo sistema. Para dar sustento a los sistemas geográficos, estos ORDBMSs tuvieron que crear un lenguaje que soportase los tipos de datos específicos y un analizador (*parser*) de consultas que pudiera interpretar SQL. La gran cantidad de información almacenada en el GIS requiere el uso de optimizadores de consultas ampliados, la indexación y una estructura de almacenamiento eficientemente diseñada que prestase especial atención a cualquier servicio de replicación necesario. El, a veces, largo tiempo de procesamiento de las consultas requería de servicios de transacción especiales que tratasen con el gran número de registros geográficos.

Durante los últimos cinco años, han aparecido varias extensiones de bases de datos de diferentes fabricantes que amplían sus ORDBMSs.[7] Informix lanzó Spatial y Geodetic DataBlade (que implementa los índices de árbol R), IBM hizo público su DB2 Spatial Extender y Oracle sus opciones OracleSpatial y OracleLocator. Estas extensiones permiten que el usuario almacene, administre y recupere objetos geográficos. Pueden usarse junto con el software GIS para conseguir funciones más complejas de análisis, edición y mapeado espaciales. El ArcSDE de ESRI es un software GIS que puede usarse para acceder a grandes bases de datos geográficas multiusuario almacenadas en RDBMSs. Permite una mejora en el rendimiento de la administración de los datos, amplía el rango de tipos de datos de la base de datos y activa la portabilidad de esquemas entre RDBMSs. ArcSDE soporta DB2 de IBM, Informix, Oracle y Microsoft SQL. Más recientemente, algunos fabricantes han lanzado soluciones para el servidor de aplicación que permiten enrutamiento, mapeo y capacidades de análisis más avanzadas. MapViewer de Oracle es un ejemplo que soporta el mapeo de la información almacenada en OracleSpatial u OracleLocator. ESRI e IBM colaboraron para introducir una solución espacial que incluye el DB2 Universal Database™ Spatial Extender para permitir al usuario almacenar y manipular datos espaciales y tradicionales.

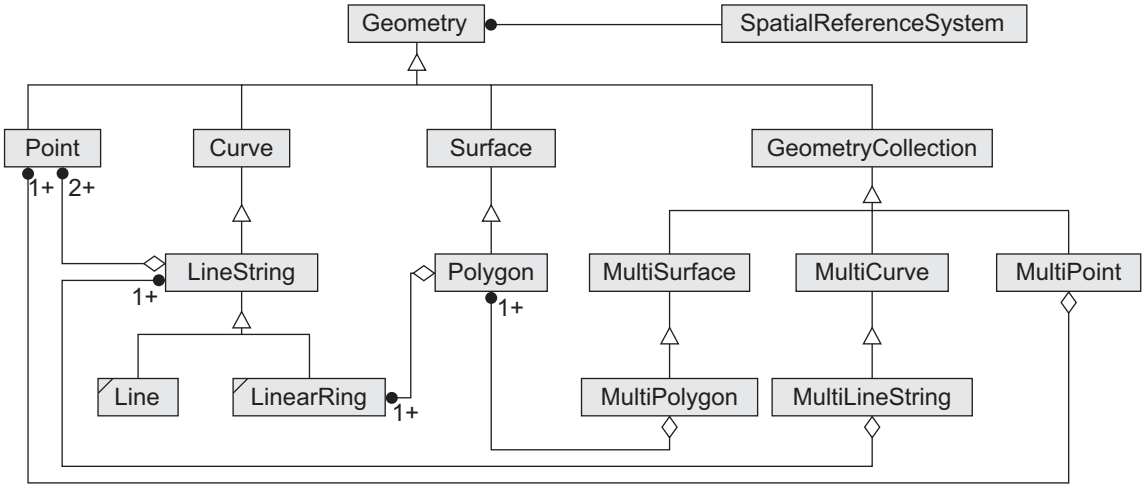
### 30.3.5 Estándares y operaciones GIS

Uno de los primeros pasos fundamentales del diseño de una base de datos es comprender los requisitos del sistema. El diseño se basa en los bloques de construcción del sistema, es decir, las entidades, los métodos y las restricciones que éste podría soportar. Las necesidades de estándares y políticas dentro de la comunidad GIS que definan los tipos de datos y los métodos han sido dirigidas por varias organizaciones. Algunas de ellas, como OGC™ (Consorcio geoespacial abierto, *Open Geospatial Consortium*), FGDC (Comité federal de datos geográficos de Estados Unidos, *U.S. Federal Geographic Data Committee*) e ISO (Organización internacional para la estandarización, *International Organization for Standardization*) empezaron a definir estándares para permitir la comparición de datos espaciales. En 1999, el OGC definió los tipos de datos centrales (véase la Figura 30.3) y los métodos que deben soportar los sistemas GIS. El esquema propuesto estaba diseñado para su acceso a través de SQL. En 2003, el OGC publicó el GML (Lenguaje de marcado geográfico, *Geography Markup Language*) que es un XML codificado especialmente para ajustarse a la información geográfica. Permite el transporte, modelado y almacenamiento de información espacial, incluyendo la geometría y las propiedades de los rasgos geográficos. Los objetos definidos en GML soportan la descripción de la geometría de un rasgo, la topología, el sistema de referencia, etc.

En el modelo de Jerarquía de Clase Geométrica (*Geometry Class Hierarchy*) propuesto por el OGC, la geometría es el tipo raíz. Cuenta con un sistema espacial referenciado (sistema de coordenadas, proyección) y cuatro subtipos: Point, Curve, Surface y GeometryCollection. GeometryCollection es una colección de posibles tipos de geometrías diferentes. MultiPoint, MultiLineString y MultiPolygon son subtipos de GeometryCollection que se utilizan para manipular la agregación de las distintas geometrías. Curve es un objeto geométrico unidimensional, una secuencia de puntos, con un subtipo LineString. LineString especi-

[7] Varios de los productos mencionados aquí son marcas registradas; hemos enumerado las conocidas al final de esta sección.

**Figura 30.3.** Jerarquía de clase geométrica.




Por cortesía de Open GIS Consortium, 1999.

fica la forma de interpolación entre los puntos. La notación de la Figura 30.3 también indica la restricción de cardinalidad mínima (por ejemplo, +1, +2) de algunas de las relaciones establecidas entre las características.

Los estándares definen varios métodos para la verificación de las relaciones espaciales entre los objetos geométricos:

- **Igualdad (*equal*).** ¿Es la geometría *espacialmente igual* a otra geometría?
- **Disjunto (*disjoint*).** ¿Comparten las geometrías un punto?
- **Intersección (*intersect*).** ¿Se cortan las geometrías?
- **Contacto (*touch*).** ¿Se tocan espacialmente las geometrías (intersectan en sus bordes)?
- **Cruce (*cross*).** ¿Están las geometrías cruzadas espacialmente? ¿Se solapan (pueden ser de distintas dimensiones como líneas y polígonos)?
- **Dentro (*within*).** ¿Está la geometría *espacialmente dentro* de otra?
- **Contiene (*contain*).** ¿Está la geometría completamente contenida dentro de otra?
- **Superposición (*overlap*).** ¿Se solapan las geometrías (deben estar en la misma dimensión)?
- **Relación (*relate*).** ¿Están relacionadas las geometrías espacialmente? (Testado por la comprobación de las intersecciones entre el interior, el borde y el exterior de las dos geometrías).

El análisis espacial de las geometrías en el sistema se consigue definiendo los siguientes métodos:

- **Distancia (*distance*).** Devuelve la distancia más corta entre dos puntos cualesquiera de las dos geometrías.
- **Búfer (*buffer*).** Devuelve una geometría que representa a todos los puntos cuya distancia desde la geometría dada es menor o igual a la distancia.
- **ConvexHull.** Devuelve una geometría que representa la parte convexa de la misma. La parte convexa es el polígono más pequeño que engloba un conjunto de puntos dados y que contiene todas las posibles líneas formadas por ese conjunto de puntos.
- **Intersección (*intersection*).** Devuelve una geometría que representa el conjunto de puntos de intersección de esa geometría con otra.

- **Unión (*union*).** Devuelve una geometría que representa el conjunto de puntos de unión de esa geometría con otra.
- **Diferencia (*difference*).** Devuelve una geometría que representa el conjunto de puntos de diferencia de esa geometría con otra.
- **SymDifference.** Devuelve una geometría que representa el conjunto de puntos de diferencia simétrica de esa geometría con otra (los puntos que están en una de las geometrías pero no en las dos).

Los estándares definen aserciones para las diferentes geometrías. Por ejemplo, un polígono está definido como una superficie plana descrita por un límite exterior y que puede tener varios límites interiores. Cada uno de estos límites interiores define un hueco en el polígono. Para esta geometría se deben cumplir las siguientes aseveraciones:

- Los polígonos están cerrados topológicamente.
- El borde de un polígono es un conjunto de anillos lineales que define sus límites exterior e interior.
- Ninguno de los anillos del borde se cruzan. Pueden cortarse en un punto pero sólo pueden ser tangentes.

Una característica está definida como un objeto con localización espacial y atributos geométricos. Se almacenan en las tablas como filas y cada atributo geométrico es una clave externa que hace referencia a una tabla o vista de geometría. Las relaciones entre los rasgos están definidas como referencias de clave externa entre las tablas.

Una restricción importante que recibe una especial atención en los estándares es el sistema de referencia espacial, también conocido como Datum, que mantiene el tamaño, la forma y el origen de un elipsoide que representa a la Tierra. Cada rasgo se almacena en una fila de la base de datos y debe contar con una columna asociada en la que se indique su sistema de referencia espacial. El Sistema de Referencia Espacial identifica el sistema de coordenadas de todas las geometrías almacenadas en la columna. Define la localización específica del rasgo en una superficie tridimensional (longitud/latitud) o en un sistema de proyección específico (la proyección es una transformación matemática de la forma tridimensional de la Tierra a un mapa plano y contiene distorsiones inherentes). La tabla de Sistemas de Referencia Espacial almacena información de cada uno de los Sistemas de Referencia individuales de la base de datos. El siguiente ejemplo muestra la creación de una tabla de estados:

```
CREATE TABLE ESTADOS
 ( NombreEstado   VARCHAR(50)   NOT NULL,
   FormaEstado    POLYGON       NOT NULL,
   País           VARCHAR(50),
 PRIMARY KEY (NombreEstado),
 FOREIGN KEY (País) REFERENCES PAÍSES (NombrePaís),
 );
```

Esta sentencia define el nombre del estado, su geometría (polígono) y el país, además de indicar que la clave primaria es el nombre del estado (no puede ser nulo) y una clave externa contra la tabla de países.

La siguiente es una sentencia que recupera los estados que tengan un área mayor que 50.000:

```
SELECT    NombreEstado
FROM      ESTADOS
WHERE     (AREA(FormaEstado) >50000);
```

Area es un método definido en los estándares OGC (OGC 1999) que devuelve el área de una superficie en las unidades del sistema de coordenadas. La sentencia siguiente recuperará todos los estados que comparten frontera con Texas. El método Touches devuelve 1 cuando las geometrías se tocan espacialmente.

```
SELECT    S1.NombreEstado
FROM          ESTADOS S1, ESTADOS S2
WHERE     ( (TOUCHES( S1.FormaEstado, S2.FormaEstado) == 1)
          AND
          (S2.NombreEstado = 'Texas') )
```

En Estados Unidos se ha adoptado un SDTS (Estándar de transferencia de datos espaciales, *Spatial Data Transfer Standard*) como la Publicación 173 FIPS (Estándar de procesamiento de información federal, *Federal Information Processing Standard*). SDTS es un estándar de intercambio independiente orientado a la transferencia de datos espaciales. Soporta el cambio de rasgos GIS vectoriales, sus atributos, topología y metadatos así como conjuntos de datos *raster*. Compañías como ESRI (ArcINFO), LEICA (Erdas-IMAGINE), Intergraph y MapInfo han incorporado el soporte para los estándares SDTS en sus productos.

Aunque se ha invertido mucho esfuerzo en el desarrollo de estándares para el almacenamiento, la manipulación y la transferencia de datos geográficos, la comunidad GIS aún no ha adoptado completamente sus estándares. Algunas organizaciones todavía mantienen sus viejos sistemas, que a veces están basados en ficheros, y encuentran complicado adoptar las regulaciones dirigidas a compartir los datos.

### 30.3.6 Aplicaciones y software GIS

Desde que GIS trata con el mundo que nos rodea, existen una gran cantidad de aplicaciones basadas en él. GIS empezó en los centros de investigación de las universidades y ha sido utilizado tradicionalmente por empresas y disciplinas específicas como los militares y el gobierno. GIS se expandió a otros campos en la década pasada, como el mercado inmobiliario, y en la actualidad se utiliza en casi todos los aspectos de nuestra vida cotidiana. Se usa en aplicaciones para la toma de decisiones de pequeñas y grandes compañías, en el sector de la salud, la ciencia medioambiental, la planificación de ciudades, la navegación, el transporte, los servicios de emergencia, el cumplimiento de la ley, el control del censo, etc. Las personas normales y corrientes interactúan con la tecnología GIS cada vez que recuperan direcciones de los sitios de Internet o utilizan los sistemas de navegación GPS de sus vehículos.

Debido a las distintas necesidades de los usuarios finales, existe una gran variedad de productos y formas de interactuar con GIS. Los profesionales necesitan software como ArcINFO de ESRI, que facilita la edición de datos y herramientas de análisis avanzados. Una sencilla confección de mapas, informes y análisis puede conseguirse a través de un GIS de escritorio como ArcGIS de ESRI. Otros productos, como ArcPad de ESRI, proporcionan mapeado, recopilación de datos GIS e integración GPS en los dispositivos móviles ligeros utilizados por los usuarios finales. Como ya hemos mencionado, los GIS de Internet permiten mostrar, consultar y confeccionar mapas mediante sistemas como ESRI ArcIMS, Intergraph GeoMedia Web Map y MapInfo MapXtreme.

**Ejemplo de software GIS: ArcINFO y Geodatabase.** ArcINFO y Geodatabase son productos de ESRI (Environmental Systems Research Institute, Inc.). ArcINFO fue diseñado originalmente en la década de los 70 para minicomputadores, evolucionó hacia un entorno UNIX y actualmente está disponible en varias plataformas. Los datos de ArcINFO se almacenan en espacios de trabajo. Cada uno de ellos cuenta con varios campos de aplicación relacionados, capas de datos y un directorio INFO. El formato de campo de aplicación (*coverage*) es un formato propiedad de ESRI para el almacenamiento de datos de tipo vectorial. Los objetos básicos implementados en ArcINFO son puntos, líneas y polígonos. El directorio INFO es el primer RDBMS totalmente robusto utilizado en GIS. INFO almacena y manipula los datos de atributos de las diferentes geometrías que están almacenadas en un sistema basado en ficheros.

El formato *coverage* permite que el usuario construya la topología entre los rasgos geográficos. Las estructuras de datos topológicas contienen información integrada acerca de las relaciones espaciales. La definición de topología está heredada de las nociones básicas que declaran que un arco es una curva entre dos nodos que

tiene puntos intermedios (vértices) y que un polígono es un conjunto de arcos. Estas relaciones se almacenan en tablas de atributos: PAT (PointAttributeTable), AAT (ArcAttributeTable) y PAT (PolygonAttributeTable) (PAT es la misma para polígono y punto porque un polígono está identificado en el sistema por una etiqueta punto). Es posible almacenar otro tipo de información de atributos en tablas diferentes y concatenarlas con las primeras. Para permitir operaciones de análisis rápido, puede crearse la topología de una capa usando comandos como Clean y Build.

ArcINFO consta de varios módulos que permiten la manipulación de los datos. ARC es el módulo principal. Con ARC, el usuario puede limpiar (Clean) y construir (Build) una topología de capas *coverage*. Clean comprueba la violación de las reglas topológicas como los arcos colgados, huecos entre nodos cercanos, etc. ARC permite el análisis espacial (revestimiento de capas vectoriales, intersección, unión, división, etc.). También ofrece funciones para la importación y la exportación de datos y varias utilidades como la definición de proyecciones. El módulo AML (*Arc Macro Language*) permite al usuario escribir *scripts* que incorporen comandos de otros módulos de arcINFO. ArcEdit permite la digitalización y la edición de datos, mientras que con ArcPlot podemos crear mapas complejos. El módulo GRID se utiliza para el análisis *raster*. Este análisis está limitado y no permite el procesamiento avanzado de una imagen (por ejemplo, detectar los bordes). También existe el módulo Network, empleado para el enrutamiento (localizar la ruta más corta...), la geocodificación y la ubicación, y el módulo Image Integration. El DBMS de arcINFO reside en el directorio INFO y las consultas pueden realizarse mediante el módulo Tables. Las funciones de consulta principales son Select/Reselect/Aselect y Nselect. Select se utiliza para declarar la tabla sobre la que el usuario realizará la consulta. Reselect es el equivalente del comando select de SQL. Aselect se emplea para añadir filas a una selección previa usando una sentencia de consulta diferente, y Nselect se usa para revertir la selección.

Geodatabase es el modelo de datos definido por ESRI (para los productos y aplicaciones arcGIS) para el almacenamiento, la recuperación y la manipulación de datos geográficos. Geodatabase mejora el entrono de trabajo al permitir que el usuario administre los datos en un modelo de base de datos relacional, implemente reglas de negocio y construya topologías y relaciones entre los tipos de datos. Este modelo permite el almacenamiento de objetos complejos en una base de datos relacional. A diferencia de ArcINFO, donde la geometría se guarda en un sistema basado en ficheros, en Geodatabase se hace en una tabla de una base de datos. Geodatabase soporta el modelo de datos vectorial orientado a objetos y organiza los datos geográficos en una jerarquía de objetos: clase objeto (no espacial), clase rasgo geográfico (espacial) y conjuntos de datos de rasgos (clases de rasgo con la misma referencia espacial). Los rasgos geográficos en las *geodatabases* tienen una forma geométrica y relaciones con otros rasgos. La forma del rasgo se almacena en una columna llamada geometry y puede ser representada como puntos (o multipuntos), polilíneas y polígonos. Los atributos y los comportamientos pueden adjuntarse al rasgo como subtipos y reglas de topología e integridad. Por ejemplo, la capa carretera puede especializarse en autopistas, carreteras principales y carreteras secundarias, y una autopista debe contener un número mínimo de carriles. Por otra parte, la topología puede definirse entre rasgos y es posible forzar las reglas de topología. Por ejemplo, no puede coincidir el contorno de un edificio con una carretera o los polígonos de los estados no pueden solaparse.

Geodatabase permite diseñar una base de datos personal que soporte muchos lectores y un único editor usando Microsoft® Office Access como base de datos. Las bases de datos multiusuario incorporan un sistema como Oracle con el software ArcSDE para la gestión del esquema y la edición.

### 30.3.7 Trabajo futuro en GIS

GIS se ha desarrollado rápidamente sobre todo durante los últimos diez años. El auge de las tecnologías de bases de datos y el creciente interés que muestran las nuevas disciplinas por GIS han dado lugar a nuevas preguntas y problemas. Las nuevas aplicaciones continuarán presentando nuevos desafíos como los siguientes:

- **Fuentes de datos.** Con mucha frecuencia aparecen nuevos métodos de recopilación de datos en la comunidad GIS. Con prontitud y eficacia se incorporan al sistema nuevos satélites, cámaras aéreas y

GPS. Un ejemplo reciente es el satélite Nube de puntos LIDAR que implica una enorme cantidad de datos a procesar. La fusión de nuevas tecnologías con distintas precisiones es un constante desafío al que se enfrenta la comunidad. La ingente cantidad de información introducida por algunas de estas nuevas fuentes es un importante obstáculo para las bases de datos. La diversidad en las fuentes de datos redunda en el desafío de la integración.

- **Modelos de datos.** Los dos modelos de datos más utilizados son los vectoriales y los *raster*. Un importante problema práctico al que se enfrentan los sistemas GIS actuales es que tratan sobre todo con un modelo. La asociación de los datos *raster* y vectoriales está limitada y la incorporación completa de ambos modelos sigue siendo un desafío.
- **Estándares.** La comunidad GIS necesita una implementación global o, al menos, estándares nacionales. Las recomendaciones ISO, OGC y FDGC deben ser dirigidas, consideradas y usadas tanto por los sectores públicos como por los privados. Esto permitiría un mejor intercambio de información entre las distintas organizaciones de todo el mundo. El International Standardization Body (ISO TC 211) y el European Standards Body (CEN TC278) están actualmente debatiendo varios temas, entre ellos, la conversión entre los datos vectoriales y *raster* para mejorar el rendimiento de las consultas.
- **Nuevas arquitecturas.** Las aplicaciones GIS necesitarán una nueva arquitectura cliente/servidor que se beneficie de los avances producidos en los RDBMS y los OODBMSs y en la tecnología de bases de datos relacionales extendidas. Los sistemas anteriores separaban los datos espaciales de los que no lo eran y dejaban el control de estos últimos a un DBMS. Dicho proceso exige un modelado y una integración adecuados a medida que evolucionan los dos tipos. Geodatabase parece que ha solventado este problema gracias a arcINFO. Sin embargo, se necesitan herramientas específicas para transferir datos, controlar cambios y gestionar el flujo de trabajo.
- **Estrategia del ciclo de vida del objeto y su versionado.** Debido a las constantes evoluciones de los datos geográficos, GIS debe mantener una cartografía elaborada y datos del área: un problema de administración que debe aliviarse mediante la actualización incremental unida a los esquemas de autorización de actualizaciones para los diferentes niveles de usuarios. Bajo la estrategia del ciclo de vida del objeto, que incluye las actividades de creación, destrucción y modificación del mismo, así como el auspicio de nuevas versiones en objetos permanentes, puede definirse un completo conjunto de métodos para controlar esas actividades.
- **GIS móvil.** Los dispositivos móviles con información geográfica son cada vez más comunes, y la recepción de actualizaciones *online* junto con las consultas a bases de datos mientras se está de viaje serán su área de aplicación principal. Con la llegada de los teléfonos celulares con capacidad GPS, las aplicaciones basadas en la localización son inminentes. El GIS móvil será un tema extremadamente importante en las principales inversiones en investigación.
- **GIS temporal.** El panorama geográfico que nos rodea cambia constantemente. El hombre realiza cambios constantemente, y muchos de estos cambios deben ser registrados. Un GIS que no se mantenga en continua actualización tenderá a desaparecer. Todas las modificaciones humanas, como carreteras, edificios, presas y canales fluviales, deben actualizarse con una intervención mínima.
- **Modelado de varios aspectos de GIS.** En el futuro, los sistemas cubrirán un amplio rango de funciones (desde herramientas para el análisis de mercados, hasta utilidades para la navegación de automóviles) que necesitarán datos y funcionalidades orientadas a los límites. En el otro extremo, las aplicaciones medioambientales, la hidrología y la agricultura necesitarán modelos de datos orientados al terreno. No está claro que todas estas funcionalidades puedan estar soportadas por un GIS de propósito general. Las necesidades especializadas de los GIS obligarán a que los DBMSs de propósito general deban mejorar incluyendo tipos de datos y funciones adicionales antes de que las aplicaciones GIS generales puedan estar soportadas.

- **Notación común.** Las herramientas y los sistemas propuestos en GIS deben ser también intuitivos y contar con una notación fácil de usar. Ya que el ISO/TC 211 y el OGC adaptaron UML al modelado de clases espaciales, es preciso considerar una extensión UML que trate con objetos geográficos.
- **Generalización.** Los sistemas GIS almacenan datos en diferentes niveles de escala y precisión. Cuando un usuario recupera información de la base de datos, se solicita con un cierto grado de escala y precisión que puede diferir de los que están guardados originalmente. Por ello, es preciso tratar con problemas como la actualización de una capa específica y la generación automática del resto de niveles.
- **DBMSs especializados para GIS.** Las tecnologías de bases de datos evolucionan y la comunidad GIS debe incorporar las ventajas que los DBMSs tienen que ofrecer, como la optimización y el procesamiento de consultas complejas, la seguridad, el *backup*, la recuperación y, en algunos casos, los mecanismos de recuperación ante fallos. Aunque algunos fabricantes GIS intentan seguir los avances del entorno de las bases de datos, los usuarios están acostumbrados a ciertos productos y pueden ser renuentes a aprender y trabajar con un nuevo entorno. Problemas como los diferentes niveles de precisión que deben mantenerse en la base de datos y ser incorporados al proceso de análisis en los diferentes modelos de datos, necesitan aún mucho estudio. La ausencia de administradores de bases de datos que entiendan las necesidades especiales de GIS puede ser un obstáculo para muchas organizaciones públicas y privadas que implementan estos sistemas usando un DBMS estándar.

# 30.4 Control de los datos del genoma[8]

## 30.4.1 Biología y genética

La biología engloba una enorme variedad de información. Las ciencias medioambientales nos ofrecen una visión del modo en que las especies viven e interactúan en un mundo repleto de fenómenos naturales. La biología y la ecología estudian especies particulares. La anatomía se centra en la estructura completa de un organismo, documentando los aspectos físicos de cuerpos individuales. La medicina tradicional y la fisiología desmenuzan el organismo en sistemas y tejidos y se afanan en recopilar información acerca del trabajo de esos sistemas y del organismo como un todo. La histología y la biología celular profundizan en los tejidos a nivel celular y ofrecen nociones acerca de la estructura interna y la función de la célula. Esta riqueza de información que se ha generado, clasificado y almacenado durante siglos se ha convertido recientemente en una aplicación primordial de la tecnología de bases de datos.

La **genética** ha emergido como un campo ideal para la aplicación de la tecnología de la información. En un sentido amplio, puede considerarse como la construcción de modelos de datos basados en la información sobre los genes (considerados como las unidades fundamentales de la herencia) y la búsqueda de relaciones entre esa información. La genética puede dividirse en tres ramas: mendeliana, molecular y poblacional. La genética mendeliana es el estudio de la transmisión de los rasgos entre las distintas generaciones. La genética molecular se encarga de analizar la estructura química y la función de los genes a nivel molecular. Por último, la genética poblacional observa el modo en que la información genética varía entre los distintos grupos de organismos.

La genética molecular ofrece un aspecto mucho más detallado de la información genética al permitir que los investigadores examinen la composición, la estructura y el funcionamiento de los genes. Sus orígenes pueden establecerse en dos descubrimientos importantes. El primero ocurrió en 1869 cuando Friedrich Miescher descubrió el núcleo y su componente primario, el ADN (ácido desoxiribonucleico). En investigaciones posteriores se determinó que el ADN y otro compuesto relacionado, el ARN (ácido ribonucleico), formaban parte de los nucleótidos (azúcar, fosfato y base que, combinados, formaban el ácido nucleico) enlazados en polímeros

[8] Se agradece la contribución de Ying Liu, Nalini Polavarapu y Saurav Sahay a esta sección.

largos mediante el azúcar y el fosfato. El segundo descubrimiento fue la demostración de Oswald Avery en 1944 de que el ADN era la sustancia molecular encargada de transportar la información genética. A partir de entonces, los genes fueron considerados como cadenas de ácidos nucleicos alineados en cromosomas y que ofrecían tres funciones principales: (1) duplicar la información genética entre las generaciones, (2) proporcionar cianotipos para la creación de polipéptidos y (3) acumular los cambios que permiten que permiten la evolución. Watson y Crick descubrieron la estructura de doble hélice del ADN en 1953, lo cual mostró una nueva dirección en la investigación de la genética molecular.[9] El descubrimiento del ADN y de su estructura es considerado como el trabajo biológico más importante de los últimos 100 años, y el nuevo campo que se abrió puede ser la frontera científica de los próximos 100 años. En 1962, Watson, Crack y Wilkins ganaron el Premio Nóbel de fisiología/medicina por este gran hallazgo.[10]

## 30.4.2 Características de los datos biológicos

Los datos biológicos exhiben muchas características especiales que hacen que su control suponga un gran problema. Por ejemplo, las nuevas técnicas experimentales permiten la obtención de ingentes cantidades de datos biológicos a partir de un único experimento. El desarrollo de los *microarrays* de cDNA de alta densidad, u oligonucleótidos, en la década de los 90 del pasado siglo ha hecho surgir un nuevo nivel de sensibilidad y especificidad en la monitorización a gran escala de la expresión del gen. Un experimento con un *microarray* ADN (*gene-chip*) puede obtener datos de cientos de genes simultáneamente, lo que dificulta enormemente al investigador la interpretación de esos datos y la formación de conclusiones acerca de las funciones de los mismos. Estos *microarrays* son matrices bidimensionales en las que cada elemento registra el nivel de expresión de un gen particular de un experimento concreto. El objetivo es localizar los genes que han cambiado de forma más significativa entre el dato de ejemplo y el de control en la forma de sobreexpresión y subexpresión.

Empezaremos resumiendo las características relacionadas con la información biológica, y nos centraremos en un campo multidisciplinar llamado **bioinformática**, la cual ya se imparte en algunas universidades. La bioinformática se dirige al control de la información genética centrándose especialmente en el ADN y en el análisis de la secuencia de las proteínas. Para aparejar todos los tipos de información biológica, es preciso avanzar en su modelado, almacenamiento, recuperación y administración. Además, las aplicaciones de la bioinformática se extienden al estudio de las mutaciones y las enfermedades relacionadas, los efectos de las drogas en los genes, las rutas bioquímicas y las proteínas, las investigaciones antropológicas de los patrones de migración de las tribus y los tratamientos terapéuticos, por citar algunas.

**Característica 1.** Los *datos biológicos son altamente complejos en comparación con la mayoría de otros dominios o aplicaciones*. Definiciones como ésta implican que la información debe ser capaz de representar una subestructura compleja de datos, así como sus relaciones, y garantizar que no se pierde información durante el modelado de los datos biológicos. La estructura de los mismos ofrece con frecuencia un contexto adicional de interpretación de la información. Los sistemas de información biológica deben ser capaces de representar cualquier nivel de complejidad en cualquier esquema de datos, relación o subestructura (no sólo datos de tablas, binarios o jerárquicos). Como ejemplo, MITOMAP es una base de datos que documenta el genoma mitocondrial humano.[11]

El genoma mitocondrial es una pequeña pieza circular de ADN que encierra información de aproximadamente 16.569 bases de nucleótidos; 52 ARN mensajeros codificados de genes de emplazamiento, ARN ribosomal

[9] Véase Nature, 171:737 1953.

[10] *http://www.pbs.org/wgbh/aso/databank/entries/do53dn.html.*

[11] Los detalles acerca de MITOMAP y su compleja información pueden consultarse en Kogelnik y otros (1997, 1998), Brandon y otros (2005) y en *http://www.mitomap.org.*

y ARN de transferencia; alrededor de 1.500 variantes de población conocidas; más de 60 asociaciones de enfermedades conocidas; y un conjunto limitado de conocimiento sobre las interacciones moleculares complejas de la energía bioquímica que produce caminos de fosforilación oxidativa. Incluye enlaces a cerca de 3.000 referencias bibliográficas. Como se podría esperar, su administración ha encontrado grandes problemas; nos hemos visto incapaces de usar los RDBMSs o ODBMSs tradicionales para capturar todos los aspectos de los datos.

**Característica 2.** *La cantidad y el rango de variación de los datos son grandes*. Por tanto, los sistemas biológicos deben ser flexibles a la hora de manipular los tipos de datos y los valores. Con un rango tan amplio de posibles valores, la asignación de restricciones a los tipos de datos debe estar limitada, ya que se podrían descartar valores inesperados (por ejemplo, valores *outlier*) que son muy comunes en el dominio biológico. La exclusión de este tipo de valores provoca una pérdida de información. Además, las frecuentes excepciones producidas en las estructuras de los datos biológicos pueden precisar la elección de tipos de datos que estén disponibles para un fragmento de información concreto.

**Característica 3.** *Los esquemas de las bases de datos biológicas cambian rápidamente*. Por tanto, para mejorar el flujo de información entre las generaciones o versiones de bases de datos, es preciso soportar la evolución del esquema y la migración de los objetos de datos. La capacidad de ampliar el esquema, algo muy frecuente cuando se trabaja en un escenario biológico, no está soportada por la mayoría de sistemas de bases de datos relacionales u orientados a objetos. En breve, sistemas como GenBank publicarán la base de datos completa con nuevos esquemas una o dos veces al año, en lugar de hacer las modificaciones de forma incremental cada vez que sea necesario. Una base de datos evolucionista debe ofrecer un mecanismo oportuno y ordenado de seguir los cambios que se producen en entidades individuales de las bases de datos biológicas a lo largo del tiempo. Este tipo de seguimiento es importante para que los investigadores sean capaces de acceder y reproducir resultados previos.

**Característica 4.** *Las representaciones del mismo dato realizadas por distintos biólogos podrían ser diferentes (incluso usando el mismo sistema)*. Por tanto, es preciso soportar mecanismos para *alinear* diferentes esquemas, o versiones, biológicos. Dada la complejidad de este tipo de información, existen muchas formas de modelar una entidad concreta, lo que hace que los resultados reflejen con frecuencia el foco de interés particular del investigador. Aun cuando dos personas puedan producir diferentes modelos de datos cuando se les pide interpretar la misma entidad, estos modelos tendrán numerosos puntos en común. En situaciones como ésta, a los científicos les podría resultar interesante ejecutar consultas a través de estos puntos comunes. El enlace de estos datos a través de una red de esquemas podría resolver el problema.

**Característica 5.** *La mayoría de los usuarios de los datos biológicos no necesitan acceso de escritura a la base de datos; un acceso de sólo lectura sería suficiente*. El acceso de escritura está limitado a los usuarios con privilegios llamados guardianes (*curators*). Por ejemplo, la base de datos creada como parte del proyecto MITOMAP tiene un promedio de más de 15.000 usuarios de Internet por mes. Al mes se producen menos de veinte solicitudes a MITOMAP de tipo no guardián. En otras palabras, el número de usuarios que requieren acceso de escritura es pequeño. Los usuarios generan un gran abanico de patrones de acceso de lectura a la base de datos, pero no son los mismos que los empleados en las bases de datos tradicionales. Las búsquedas temporales realizadas por los usuarios, demandan la indexación de combinaciones, con frecuencia inesperadas, de las clases de instancia de datos.

**Característica 6.** *La mayoría de biólogos no tienen constancia de la estructura interna de la base de datos ni del diseño del esquema*. Las interfaces de las bases de datos biológicas muestran información a los usuarios de forma que sea aplicable al problema que intentan resolver y que refleje la estructura de datos subyacente de una forma fácilmente comprensible. Los usuarios "biológicos" suelen saber qué tipo de dato precisan, pero carecen del conocimiento técnico acerca de la estructura del mismo o de cómo lo representa el DBMS. Confían en que los técnicos les proporcionen las vistas necesarias contra la base de datos. Los esque-

mas relacionales fallan a la hora de ofrecer pistas, o una información intuitiva, al usuario relacionada con el significado de su esquema. Las interfaces web, en particular, ofrecen con frecuencia pantallas de búsqueda preestablecidas, las cuales pueden limitar el acceso a la base de datos. Sin embargo, si estas interfaces estuvieran generadas directamente a partir de la estructura de la base de datos, podrían ofrecer un mayor rango de acceso, aunque podrían no garantizar una usabilidad máxima.

**Característica 7.** *El contexto de los datos añade significado sobre su uso en aplicaciones biológicas*. Por tanto, el contexto debe ser mantenido y llevado al usuario cuando sea necesario. Adicionalmente, debería ser posible integrar tantos contextos como fuera posible con vistas a maximizar la interpretación de los datos biológicos. Los valores aislados son los que menos se utilizan en estos entornos. Por ejemplo, la secuencia de una hebra de ADN no es particularmente útil si no se cuenta con información adicional acerca de su organización, función, etc. Un nucleótido sencillo de una hebra de ADN, por ejemplo, visto en un contexto en el que se generan hebras de ADN no deseadas, podría interpretarse como un elemento causante de la anemia de células falciformes.

**Característica 8.** *La definición y representación de consultas complejas es extremadamente importante para los biólogos*. De esta forma, los sistemas biológicos deben soportan la ejecución de consultas complejas. Sin conocimiento alguno de la estructura de los datos (consulte la Característica 6), los usuarios normales no pueden construir una consulta compleja mediante los conjuntos de datos de su propiedad. Por tanto, y para que sean verdaderamente útiles, los sistemas deben ofrecer alguna herramienta para la construcción de estas consultas. Como ya hemos comentado anteriormente, muchos sistemas ofrecen plantillas de consulta predefinidas.

**Característica 9.** *Los usuarios de la información biológica necesitan acceder con frecuencia a los valores "antiguos" de los datos, particularmente cuando quieren verificar los resultados anteriores*. De este modo, es preciso que la base de datos ofrezca soporte a las modificaciones de los valores de los datos mediante algún sistema de archivos. En el ámbito biológico, es importante poder acceder al valor más reciente así como al anterior. Los investigadores quieren consultar el dato más actualizado, pero también deben ser capaces de reconstruir el trabajo previo y evaluar de nuevo la información actual y la anterior. Así, los valores que van a actualizarse en una base de datos biológica no pueden desecharse alegremente. Todas estas características apuntan claramente hacia el hecho de que los DBMSs actuales no satisfacen por completo las necesidades y la complejidad inherentes a los datos biológicos. Por tanto, se hace necesaria una nueva dirección en los sistemas de administración de bases de datos.[12]

### 30.4.3 El proyecto del genoma humano y las bases de datos biológicas existentes

El término genoma está definido como la información genética total que puede obtenerse acerca de una entidad. El genoma humano, por ejemplo, suele hacer referencia al conjunto completo de genes necesarios para crear un ser humano (más de 25.000 diseminados por más de 23 pares de cromosomas y una cantidad estimada de 3 a 4 mil millones de nucleótidos). El objetivo del HGP (Proyecto del genoma humano, *Human Genome Project*) ha sido conseguir la secuencia completa (el orden de las bases) de estos nucleótidos. En junio del año 2000 se anunció un primer borrador de toda la secuencia del genoma humano, estudio que apareció publicado en el año 2001.[13]

El genoma está completo con un 99,9% de fiabilidad. De forma aislada, la secuencia del ADN humano no resulta particularmente útil. Sin embargo, esta secuencia puede combinarse con otros datos y usarse como una

[12] Si desea más información, consulte Kogelnik y otros (1997, 1998), y Brandon y otros (2005).

[13] Si desea más información, consulte Lander, E.S. y otros (2001).

potente herramienta que ayude en temas relacionados con la genética, bioquímica, medicina, antropología y agricultura. En las bases de datos existentes sobre el genoma, el interés ha estado puesto en la *curating* (o recopilación con algún examen inicial y control de calidad) y en la clasificación de la información referente a los datos de la secuencia del genoma. Además del genoma humano, se han investigado el de numerosos organismos como *E. coli*, *Drosophila* y *C. elegans*. Vamos a comentar a continuación algunas de las bases de datos y sistemas existentes que soportan o se han desarrollado a partir del HGP.

**GenBank.** En la actualidad, la mejor base de datos sobre la secuencia del ADN es GenBank, mantenida por el NCBI (Centro nacional de información biotecnológica, *National Center for Biotechnology Information*) de la NLM (Librería nacional de medicina, *National Library of Medicine*). Se estableció en 1978 como un almacén central para los datos de la secuencia del ADN. Desde entonces, ha ampliado algo su ámbito para incluir datos de etiqueta de secuencia, datos de las secuencias de las proteínas, estructuras de las proteínas en tres dimensiones, taxonomía y enlaces a la MEDLINE (literatura biomédica). En agosto de 2005, la base de datos superaba los 100 gigabases de secuencias de datos representativas tanto de genes individuales como de genomas completos o parciales de más de 165.000 organismos. A través de la colaboración internacional con el EMBL (Laboratorio de biología molecular europeo, *European Molecular Biology Laboratory*) en el Reino Unido y el DDBJ (Banco de datos de ADN de Japón, *DNA Data Bank of Japan*), los datos se intercambian diariamente entre estas tres organizaciones. La copia espejo de las secuencias de datos de estos tres sitios ofrece a los científicos de todo el mundo un acceso más rápido a la información.

Aunque es una base de datos compleja y de gran amplitud, se centra en las secuencias humanas y de otros organismos y en los enlaces a la literatura que versa sobre el tema. Se han añadido recientemente otras fuentes de datos limitadas (por ejemplo, la estructura tridimensional y el OMIM, comentados más adelante), reformando el OMIM y las bases de datos PDB existentes, y rediseñando la estructura del sistema GenBank para adaptarlo a estos nuevos conjuntos de datos.

El sistema se mantiene como una combinación de ficheros planos, bases de datos relacionales y ficheros que contienen **ASN.1 (Notación de sintaxis abstracta 1***, **Abstract Syntax Notation One***), una sintaxis para la definición de estructuras de datos desarrollada por la industria de las telecomunicaciones. Cada entrada GenBank está asignada por el NCBI a un identificador único. Las actualizaciones tienen un nuevo identificador, lo que permite mantener intacta la entidad original con vistas a mantener un histórico. Las referencias más antiguas a una entidad no indican inadvertidamente un nuevo y posible valor inadecuado. Los conceptos más recientes también reciben un segundo conjunto de identificadores únicos (UID), los cuales marcan la forma más actualizada del mismo a la vez que se permite el acceso a las versiones más viejas mediante su identificador original.

El usuario normal de la base de datos no puede acceder directamente a la estructura de los datos mediante una consulta o cualquier otra función, aunque es posible obtener una instantánea de la misma mediante la exportación a distintos formatos, incluyendo ASN.1. El mecanismo de consulta se realiza con la aplicación Entrez (o su versión web), que permite la búsqueda por palabra clave, secuencia y UID GenBank mediante una interfaz estática.

**GDB (Base de datos del genoma, *Genome DataBase*).** Creada en 1989, la GDB es un catálogo de datos sobre el mapeo de los genes humanos, un proceso que asocia una porción de información con una localización particular en el genoma humano. El grado de precisión de esta localización en el mapa depende del origen de los datos, aunque generalmente no se encuentra al nivel de bases nucleótidas individuales. Los datos GDB incluyen información que describe principalmente el mapa (distancia y limites de confianza) y los datos de prueba PCR (Reacción de la cadena polimerasa, *Polymerase Chain Reaction*) (condiciones experimentales, PCR elementales y reactivos usados). Recientemente se han realizado esfuerzos para añadir datos relativos a las mutaciones enlazadas a localizaciones genéticas, líneas de células usadas en los experimentos, librerías de pruebas ADN y algún polimorfismo limitado y datos sobre población.

El sistema GDB está construido alrededor de SYBASE, un DBMS relacional comercial, y sus datos están modelados usando las técnicas estándar de entidad-relación (consulte los Capítulos 3 y 4). Los implementa-

dores de GDB encontraron ciertas dificultades en el uso de este modelo a la hora de capturar algo más que un simple mapa y datos experimentales. Para mejorar la integridad de los datos y simplificar la programación de la aplicación, GDB distribuye un conjunto de herramientas de acceso a la base de datos. Sin embargo, la mayoría de los usuarios utilizan una interfaz web para localizar los diez administradores de datos entrelazados. Cada administrador controla los enlaces (relaciones) de una de las diez tablas que se encuentran dentro del sistema GDB. Al igual que ocurre con GenBank, los usuarios sólo obtienen una visión de alto nivel de los datos cuando realizan una búsqueda, pero no tienen constancia alguna de la estructura de las tablas GDB. Los métodos de búsqueda son más útiles cuando los usuarios sólo quieren encontrar un índice dentro de un mapa o datos de investigación. La búsqueda exploratoria en la base de datos con fines temporales no está impulsada mediante las interfaces presentes. La integración de las estructuras de las bases de datos de GDB y OMIM nunca fue establecida completamente.

**OMIM** (**herencia mendeliana en el hombre).** La OMIM es un compendio electrónico de información de las bases genéticas de las enfermedades humanas. Comenzó en 1966 como una copia realizada por Victor McCusick que contaba con 1.500 entradas. El GDB lo convirtió en un texto electrónico completo entre 1987 y 1989. En 1991, su administración se transfirió de la universidad Johns Hopkins al NCBI, y toda la base de datos se convirtió a su formato GenBank. En la actualidad cuenta con más de 16.000 entradas.

OMIM abarca cinco áreas de enfermedades basadas en órganos y sistemas. Cualquier propiedad morfológica, bioquímica, de comportamiento, etc. que esté bajo estudio se dice que es el **fenotipo** de un individuo o célula. Mendel se percató de que los genes pueden existir en numerosas formas diferentes conocidas como **alelos**. Un **genotipo** se refiere a la composición alélica actual de una persona.

La estructura de las entradas de fenotipo y genotipo contienen datos textuales muy poco estructurados como descripciones generales, nomenclaturas, modos de herencia, variaciones, estructura del gen, mapeos y numerosas categorías inferiores. Las entradas de texto fueron convertidas a un formato estructurado ASN.1 cuando la OMIM fue transferida al NCBI. Esto mejoró notablemente la posibilidad de enlazar los datos OMIM con otras bases de datos y proporcionó también una estructura rigurosa para los datos. Sin embargo, la forma básica de la base de datos sigue siendo difícil de modificar.

**EcoCyc** (Enciclopedia de los genes y el metabolismo de la *Escherichia coli*, *Encyclopedia of Escherichia coli Genes and Metabolism*) es un reciente experimento que combina información acerca del genoma y el metabolismo de la *E. coli* K-12. La base de datos fue creada en 1996 gracias a la colaboración entre el Stanford Research Institute y el Marine Biological Laboratory. Cataloga y describe los genes conocidos de la *E. coli*, las encimas codificadas por esos genes y las reacciones bioquímicas catalizadas por cada encima y su organización en las rutas metabólicas. EcoCyc amplía la secuencia y los dominios de función de la información genómica. En mayo de 2005, la información contenida en EcoCyc había sido deducida a partir de 10.747 publicaciones revisadas.

Lo primero que se utilizó para implementar el sistema fue un modelo de datos orientado a objetos en el que la información estaba almacenada en Ocelot, un sistema de representación del conocimiento. Los datos EcoCyc fueron ordenados en una jerarquía de clases objeto que se basaba en las observaciones de que las propiedades de una reacción son independientes de la encima que la cataliza, y que una encima tiene un número de propiedades que son *distintas de forma lógica* de sus reacciones.

EcoCyc proporciona dos métodos diferentes de consulta: (1) directa (mediante consultas predefinidas) e (2) indirecta (a través de navegación hipertexto). Las consultas directas se realizan usando menús y diálogos que permiten iniciar un gran, aunque finito, conjunto de consultas. Está soportada la no navegación de las actuales estructuras de datos. Además, no hay documentados mecanismos para la evolución del esquema.

La Tabla 30.1 resume los rasgos de las principales bases de datos relacionadas con el genoma, así como HGMDB y ACEDB. Existen algunas bases de datos sobre proteínas que contienen información acerca de sus estructuras. Entre las más importantes podemos citar la SWISS-PROT de la Universidad de Ginebra, la PDB (Banco de datos de proteínas, *Protein Data Bank*) del Laboratorio Nacional Brookhaven y la PIR (Recursos

**Tabla 30.1.** Resumen de las principales bases de datos relacionadas con el genoma.

| Nombre de la base de datos | Contenido principal | Tecnología inicial | Tecnología actual | Áreas problemáticas de la base de datos | Tipos de datos primarios |
|---|---|---|---|---|---|
| Genbank | Secuencia de ADN/ARN, proteínas | Ficheros de texto | Fichero plano/ASN.1 | Visualización del esquema, evolución del esquema, enlace con otras bases de datos | Texto, numérico, algunos tipos complejos |
| OMIM | Fenotipos y genotipos de enfermedades, etc. | *Index cards*/ ficheros de texto | Fichero plano/ASN.1 | Desestructurada, entradas de texto libre, enlaces con otras bases de datos | Texto |
| GDB | Datos de enlace del mapa genético | Fichero plano | Relacional | Expansión/ evolución del esquema, objetos complejos, enlaces con otras bases de datos | Texto, numérico |
| ACEDB | Datos de enlace del mapa genético, secuencia de datos (no humano) | OO | OO | Expansión/ evolución del esquema, enlaces con otras bases de datos | Texto, numérico |
| HGMDB | Secuencia y variantes de la secuencia | Fichero plano específico de la aplicación | Fichero plano específico de la aplicación | Expansión/ evolución del esquema, enlaces con otras bases de datos | Texto |
| EcoCyc | Reacciones bioquímicas y caminos | OO | OO | Bloqueado en jerarquía de clase, evolución del esquema | Tipos complejos, texto, numérico |

de identificación de proteínas, *Protein Identification Resource*) de la *National Biomedical Research Foundation*.

Durante los últimos diez años, el interés por las aplicaciones de bases de datos que tratan la biología y la medicina ha aumentado sensiblemente. GenBank, GDB y OMIM se han creado como almacenes centrales para ciertos tipos de datos biológicos, pero, aunque extremadamente útiles, aún no cubren el espectro completo de

los datos del proyecto del genoma humano. Sin embargo, en todo el mundo se están haciendo importantes esfuerzos destinados al diseño de nuevas herramientas y técnicas que aliviarán del problema de la administración de los datos a los biólogos y los investigadores médicos.

**Ontología del gen.** Explicamos el concepto de ontología en la Sección 30.2.3 en el contexto del modelado de la información multimedia. El Consorcio GO (Ontología del gen, *Gene Ontology*) fue creado en 1998 como una colaboración entre los tres modelos de bases de datos de organismos: la FlyBase, la MGI (Informática del genoma del ratón, *Mouse Genome Informatics*) y la SGD (Base de datos del genoma del Saccharomyces, *Saccharomyces or yeast Genome Database*). Su objetivo es producir un vocabulario estructurado, definido de forma precisa, común y controlado para describir los roles de los genes en cualquier organismo. Con la terminación de la secuencia del genoma de muchas especies, se ha observado que una gran fracción de los genes de los distintos organismos muestran similitudes en sus papeles biológicos, lo que ha llevado a los biólogos a afirmar que es muy probable que exista un universo limitado de genes y proteínas que se conservan en la mayoría, o en todas, las células vivas. En el otro extremo, los datos sobre el genoma están creciendo exponencialmente y no existe un método uniforme de interpretar y conceptuar los elementos biológicos compartidos. La GO hace posible la anotación de los productos de genes usando un vocabulario común basado en sus atributos biológicos compartidos y en la interoperabilidad entre las bases de datos genómicas.

El Consorcio GO ha desarrollado tres principios organizativos (la función molecular, los procesos biológicos y el componente celular) para describir los atributos de los genes, los productos de genes o los grupos de productos de genes. La función molecular está definida como la actividad bioquímica de un producto de gen. Los procesos biológicos hacen referencia a un objetivo biológico al cual contribuyen un gen o un producto de gen. Por último, el componente celular es el lugar de la célula en el que está activo un producto de gen. Cada ontología engloba un vocabulario de términos y relaciones bien definido. Los primeros están organizados en forma de DAGs (Gráficos acíclicos dirigidos, *Directed Acyclic Graphs)* en los que un nodo término puede tener varios padres y varios hijos. Un término hijo puede ser una *instancia de (es un/a)* o una *parte de* sus padres. En la última versión de la GO se tenían contabilizados más de 20.000 términos y más de 25.000 relaciones entre ellos. La anotación de un producto de gen funciona independientemente de cada una de las bases de datos colaboradoras. En la base de datos se incluye un subconjunto de estas anotaciones, que en la actualidad contiene más de 1.386.000 productos de genes y 5.244.000 asociaciones entre ellos y los términos GO.

La ontología del gen fue implementada usando MySQL, un DBMS relacional de código abierto. Mensualmente puede conseguirse una versión en los formatos SQL y XML. Para el acceso a la base de datos y el desarrollo de aplicaciones existe un conjunto de librerías escritas en C, Java, Perl, XML, etc. Los navegadores web y las GO independientes pueden obtenerse del Consorcio GO. Otro importante esfuerzo hacia la construcción de ontologías en los dominios de la biomedicina y la salud ha sido el UMLS (Sistema de lenguaje médico unificado, *Unified Medical Language System*) de la *National Library of Medicine*. UMLS está dividido en tres componentes: el Metatesauro, o base de datos de vocabulario; la Red semántica, que clasifica y enlaza los conceptos del vocabulario y el Léxico especialista, que contiene información léxica acerca de estos conceptos. El proyecto *Open Biomedical Ontologies* de la Universidad de Stanford es otro intento para integrar y ofrecer acceso a vocabularios controlados para su uso compartido entre distintos dominios médicos y biológicos. Estas fuentes de conocimiento se utilizarán para el marcado semántico de los sistemas de información biomédicos.

***Gene Expression Omnibus* (GEO).** Es un almacén público que almacena datos de expresiones de genes de alto rendimiento enviados por la comunidad científica. Está mantenida por el NCBI (Centro nacional para la información biotecnológica, *National Center for Biotechnology Information*). GEO archiva datos de experimentos basados en *microarrays* que miden los niveles relativos de mARN, ADN genómico y moléculas de proteínas. También contiene datos de tecnologías que no están basadas en los arrays, como SAGE (Análisis en serie de la expresión génica, *Serial Analysis of Gene Expression*) y la tecnología proteómica de la espectrometría de masas. En enero de 2006 podían estudiarse más de 69.000 ejemplos de los aproximadamente mil millones de medidas de expresiones de genes individuales, de más de 100 organismos.

## Bibliografía seleccionada

**Bibliografía seleccionada para las bases de datos móviles.** Existe un interés creciente por la computación móvil, y los estudios acerca de las bases de datos móviles han sufrido un importante crecimiento durante los últimos diez años. De entre todos los libros escritos acerca de este tema, Dhawan (1997) es una excelente fuente. Las redes inalámbricas y su futuro se tratan en Holtzman y Goodman (1993). Jing y otros (1999) es un exhaustivo estudio sobre los problemas relacionados con el acceso a la información en entornos móviles. Imielinski y Badrinath (1994) es un buen punto de partida para los temas relacionados con las bases de datos móviles, mientras que Imielinski y Badrinath (1992) aborda el tema de la ubicación de los datos y los metadatos en una arquitectura móvil. Dunham y Helal (1995) comenta los problemas del procesamiento de las consultas, la distribución de los datos y el control de las transacciones en las bases de datos móviles. Barbara (1999) es un reciente estudio de investigación de la administración de los datos móviles relacionada con la diseminación de los datos, la consistencia y la consulta dependiente de la localización. Pitoura y Samaras (1998) describe todos los aspectos de los problemas y las soluciones de las bases de datos móviles. Bertino y otros (1998) plantea métodos para la tolerancia a los fallos y la recuperación ante fallos en las bases de datos móviles. Pitoura y otros (1995) propone un modelo *cluster* para tratar con las desconexiones en el procesamiento de una transacción. Preguiça y otros (2003) investiga las transacciones basadas en reservas para las bases de datos móviles. Milojicic y otros (2002) presenta un tutorial acerca de la computación entre iguales. Corson and Macker (1999) es una nota respuesta al informe IETF (1999) que discute los problemas de rendimiento del protocolo de redes *ad-hoc* móviles. Stoica y otros (2001) y Ratnasamy y otros (2001) tratan las redes de igual-a-igual estructuradas basadas en tablas *hash* distribuidas. Acharya y otros (1995) considera los programas de difusión que minimizan la latencia media de una consulta y exploran el impacto de este tipo de programas en las estrategias óptimas de *caching* de cliente. Prasad y otros (2004) habla de una plataforma *middleware* para el desarrollo de aplicaciones MANET. Xie y otros (2003) y Xie (2005) proponen un nuevo modelo de transacción y nuevas técnicas que soporten el procesamiento de transacciones para las aplicaciones MANET. La transmisión (o *pushing*) de los datos como un medio de escalabilidad para la diseminación de información a los clientes la trata Yee y otros (2002). Chintalapati y otros (1997) proporciona un algoritmo de administración de localizaciones adaptable. Jensen y otros (2001) comenta los problemas de la gestión de los datos cuando forman parte de servicios basados en la localización. Wolfson (2001) describe una forma nueva de modelar eficazmente la movilidad de un objeto mediante la descripción de la posición usando trayectorias en lugar de puntos. Para obtener una visión inicial acerca de los problemas de la escalabilidad ISDB y obtener un acercamiento a la agregación de datos y la agrupación de clientes, consulte Mahajan y otros (1998). Los algoritmos de agregación específicos para el agrupamiento de datos en el servidor en las aplicaciones ISDB pueden encontrarse en Yee y otros (2001). Gray y otros (1993) aborda los conflictos de actualización y las técnicas de resolución en la sincronización ISDB. Breibart y otros (1999) explica en detalle los algoritmos de sincronización diferida para los datos replicados.

**Bibliografía seleccionada para las bases de datos multimedia.** La administración de las bases de datos multimedia se ha convertido en una importante área de investigación en la que existen varios proyectos industriales en marcha. Grosky (1994, 1997) ofrece dos excelentes tutoriales sobre el tema. Pazandak y Srivastava (1995) muestra una evaluación de los sistemas de bases de datos relacionados con los requisitos de las bases de datos multimedia. Grosky y otros (1997) contiene varios artículos interesantes entre los que se incluye un estudio sobre la indexación y la recuperación basado en contenido de Jagadish (1997). Faloutsos y otros (1994) también habla de un sistema de consulta de imágenes por contenido. Li y otros (1998) introduce el modelado de imagen de forma que ésta se ve como un objeto complejo, estructurado y jerárquico con propiedades semánticas y visuales. Nwosu y otros (1996) y Subramanian y Jajodia (1996) son libros sobre el tema. Flickner y otros (1995) describen el sistema QBIC desarrollado por IBM. El sistema COUGAR basado en sensor aparece en Bonnet y otros (2001). Aref y otros (2004) describe el sistema de base de datos de vídeo VDBMS. Lassila (1998) aborda la necesidad de los metadatos para el acceso a la información multimedia en la Web; el esfuerzo de la semántica web se resume en Fensel (2000). Khan (2000) ofreció una disertación acer-

ca de la recuperación de información basada en la ontología. Uschold y Gruninger (1996) es un buen punto de partida para estudiar las ontologías. Corcho y otros (2003) compara los lenguajes de ontología y comenta los métodos de construcción. El análisis del contenido multimedia, la indexación y el filtrado aparece en Dimitrova (1999). Un estudio acerca de la recuperación multimedia basada en el contenido puede verse en Yoshitaka y Ichikawa (1999). Las siguientes son otras fuentes adicionales de información:

- CA-JASMINE (ODBMS multimedia): *http://www.cai.com/products/jasmine.htm*
- Tecnologías Excalibur: *http://www.excalib.com*
- Virage, Inc. (recuperación de imágenes basada en el contenido): *http://www.virage.com*
- QBIC (Consulta por contenido de la imagen, *Query by Image Content*) de IBM: *http://www.ibm.com*

**Bibliografía seleccionada para GIS.** Existen numerosos libros dedicados a GIS. Adam y Gangopadhyay (1997) y Laurini y Thompson (1992) se centran en las bases de datos GIS y en los problemas de administración de la información. Kemp (1993) ofrece una visión general de los temas GIS y las fuentes de datos. Huxhold (1991) muestra una introducción al Urban GIS. Demers (2002) es otro libro acerca de los fundamentos de GIS. Maguire y otros (1991) ofrece una buena colección de artículos relacionados con GIS. Antenucci (1998) presenta un debate sobre las tecnologías GIS. Longley y otros (2001) argumenta acerca del papel que juega GIS como ciencia junto con los diferentes aspectos de GIS como campo interdisciplinario. El libro revisa las aplicaciones y las herramientas GIS para la resolución de problemas y la toma de decisiones. Miller (2004) relata la relación espacial y los análisis espaciales de la primera ley de la geografía introducida por Tobler. Goodchild (1992) defiende GIS como una Ciencia de información geográfica (*Geographic Information Science*) en lugar de como una herramienta o un sistema. Goodchild (1992a) ofrece una buena perspectiva de los temas relativos al modelado de datos geográficos. Shekhar y Chawla (2002) muestra los problemas y los planteamientos de la Administración de datos espaciales (*Spatial Data Management*) que forman el corazón de GIS. Friis-Christensen y Jensen (2001) muestra los diferentes aspectos y requisitos de modelado en un GIS y evalúa las notas de los modelados actuales. Borges, Laender y Davis (1999) presenta las restricciones de integridad espacial para el modelado en GIS y propone el uso de una extensión para el modelo OMT. Cockcroft (1997) propuso un conjunto de restricciones para los datos GIS. Gordillo y Balaguer (1998) habla de los problemas de las topologías y los datos de campo en el modelado GIS y propone soluciones para mejorar el diseño. Brodeur, Bédard y Proulx (2000) presenta la importancia de la documentación del almacén de bases de datos GIS en alianza con los estándares internacionales, a la vez que enfatiza el esquema conceptual y el diccionario de datos. Indulska y Orlowska (2002) presenta los problemas de los distintos niveles de agregación de las bases de datos GIS. Worboys y Duckham (2004) explora la perspectiva informática de GIS centrándose principalmente en los modelos que representan los datos espaciales. También presta mucha atención al papel del contexto en la interpretación de datos GIS. Maune (2001) proporciona una visión de las tecnologías de modelado de superficies. Bossomaier y Green (2002) es una obra elemental en las operaciones, lenguajes, paradigmas de metadatos y estándares GIS. Peng y Tsou (2003) comenta el GIS para Internet, que incluye un conjunto de las nuevas tecnologías emergentes que apuntan a que GIS sea más móvil, potente y flexible, así como más capaz de compartir y comunicar información geográfica. El sitio Web de Laser-Scan (*http://www.lsl.co.uk/papers*) es una buena fuente de información. El ESRI (Instituto de investigación de sistemas medioambientales, *Environmental System Research Institute*) cuenta con una excelente librería de libros sobre GIS (*http://www.esri.com*). La terminología GIS está definida en *http://www.esri.com/library/glossary/glossary.html*. Flynn y Pitts (2001) muestra un examen técnico de la base de arcINFO. Zeiler (1999) proporciona una guía para el modelado de datos en GIS prestando especial atención a ESRI Geodatabase e Incluye un informe de sus mejoras. La Universidad de Edimburgo mantiene una lista web de recursos GIS en *http://www.geo.ed.ac.uk/home/giswww.html*. La USGS (*U.S. Geological Survey*) tiene información y datos disponibles en *http://www.usgs.gov/*.

**Bibliografía seleccionada sobre bases de datos del genoma.** La bioinformática se ha convertido durante los últimos tiempos en un área de investigación muy popular, y alrededor de este tema se han organi-

zado muchos talleres y conferencias, incluyendo CSB, ISMB, RECOMB y PSB. Robbins (1993) muestra una buena panorámica mientras que Frenkel (1991) inspecciona el proyecto del genoma humano y su papel especial en la bioinformática. Cutticchia y otros (1993), Benson y otros (2002) y Pearson y otros (1994) son referencias sobre GDB, GenBank y OMIM. Gracias a la colaboración internacional entre el GeneBank (Estados Unidos), el DDBJ (Banco de datos de ADN de Japón, *DNA Data Bank of Japan*) (http://www.ddbj.nig.ac.jp/E-mail/homology.html) y el EMBL (Laboratorio de biología molecular europeo, *European Molecular Biology Laboratory*) (consulte Stoesser (2003)), los datos se intercambian diariamente entre las bases de datos colaboradoras para conseguir una sincronización óptima. Wheeler y otros (2000) comenta las distintas herramientas que permiten a los usuarios acceder y analizar los datos disponibles en las bases de datos. Olken y Jagadish (2003) aborda los desafíos de la administración de los datos en la biología integrativa. Navathe y Patil (2004) estudia el desarrollo de las bases de datos genómicas y proteómicas y apuntan hacia los importantes desafíos y el potencial futuro de las aplicaciones.

Wallace (1995) ha sido un pionero en la investigación del genoma mitocondrial, que versa sobre una parte específica del genoma humano; la secuencia y los detalles organizativos de esta área aparecen en Anderson y otros (1981). Kogelnik y otros (1997, 1998) y Kogelnik (1998) apuntan al desarrollo de una solución genérica al problema de la gestión de los datos en las ciencias biológicas mediante la elaboración de un prototipo en la base de datos MITOMAP. Apweiler y otros (2003) revisa el núcleo de los recursos bioinformáticos mantenidos en el EBI (Instituto Europeo de Bioinformática, *European Bioinformatics Institute*), como SWISS-PROT + TrEMBL, y resume los importantes desafíos que suponen la administración de las bases de datos de estos recursos. Informan de tres tipos de bases de datos principales: de secuencia, como la Base de Datos de Secuencia Nucleótida DDBJ/EMBL/GENEBANK; secundarias, como PROSITE, PRINTS y PFAM; e integradas como InterPro, la cual genera datos procedentes de las seis principales bases de datos de firmas de proteínas (PFAM, PRINTS, ProDom, PROSITE, SMART y TIGRFAM).

La EMSD (Base de datos de estructura macromolecular del Instituto Europeo de Bioinformática, *European Bioinformatics Institute Macromolecular Structure Database*), que es una base de datos relacional (*http://www.ebi.ac.uk/msd*) (Boutselakis y otros, 2003), está diseñada para ser un punto de acceso único a las estructuras del ácido nucleico y las proteínas, así como a información relacionada. La base de datos se deriva de las entradas de la PDB (Banco de datos de proteínas, *Protein Data Bank*). La base de datos de búsqueda contiene un extenso conjunto de propiedades derivadas, indicadores *goodness-of-fit* y enlaces a otras bases de datos EBI como InterPro, GO y SWISS-PROT, junto con enlaces a SCOP, CATH, PFAM y PROSITE. Karp (1996) comenta los problemas inherentes a la interrelación de todas las bases de datos comentadas en esta sección. Define dos tipos de enlaces: los que integran los datos y los que relacionan esos datos entre las bases de datos. Éstos fueron usados para el diseño de EcoCyc.

Los siguientes son algunos enlaces web que conviene conocer. La información sobre la secuencia del genoma humano está en *http://www.ncbi.nlm.nih.gov/genome/seq/*. A MITOMAP, desarrollada en Kogelnik (1998), se puede acceder desde *http://www.mitomap.org/*. Su más reciente actualización, junto con el desarrollo de MITOMASTER, está accesible en Brandon y otros (2005). La mayor base de datos de proteínas, SWISS-PROT, puede consultarse en *http://expasy.hcuge.ch/sprot/*. La información de ACEDB está disponible en *http://probe.nalusda.gov:8080/acedocs/*. El Consorcio GO está en *http://www.godatabase.org*. La base de datos *Gene Expression Omnibus* se encuentra en *http://www.ncbi.nlm.nih.gov/geo*. Barrett y otros (2005) describe la estructura y el desarrollo de la base de datos GEO y las herramientas relacionadas. Cada año, el número de enero de *Nucleic Acid Research* está dedicado a las bases de datos con aplicaciones biológicas.

## Créditos

Este capítulo contiene referencias a empresas y productos que tienen las siguientes **marcas registradas**:

- ESRI, ArcEdit, ArcPlot, AML, ArcINFO, ArcGIS, ArcPad, ArcIMS y ArcSDE son marcas registradas del Environmental Systems Research Institute, Inc.
- MapInfo y MapXtreme® son marcas registradas de MapInfo Corporation.

- Intergraph y GeoMedia Web Map son marcas registradas de Intergraph Corporation.
- Microsoft es una marca registrada de Microsoft Corporation.
- Informix y DataBlade son marcas registradas de Informix Corporation.
- Oracle es una marca registrada de Oracle Corporation y/o sus filiales.
- OpenGIS® es una marca registrada de Open Geospatial Consortium, Inc.
- IBM, DB2 y DB2 Universal Database son marcas registradas de International Business Machines Corporation.
- ERDAS IMAGINE es una marca registrada y propiedad exclusiva de Leica Geosystems GIS & Mapping, LLC.
- Otros nombres pueden ser marcas registradas de sus respectivos propietarios.

# A

# Notaciones diagramáticas alternativas para los modelos ER

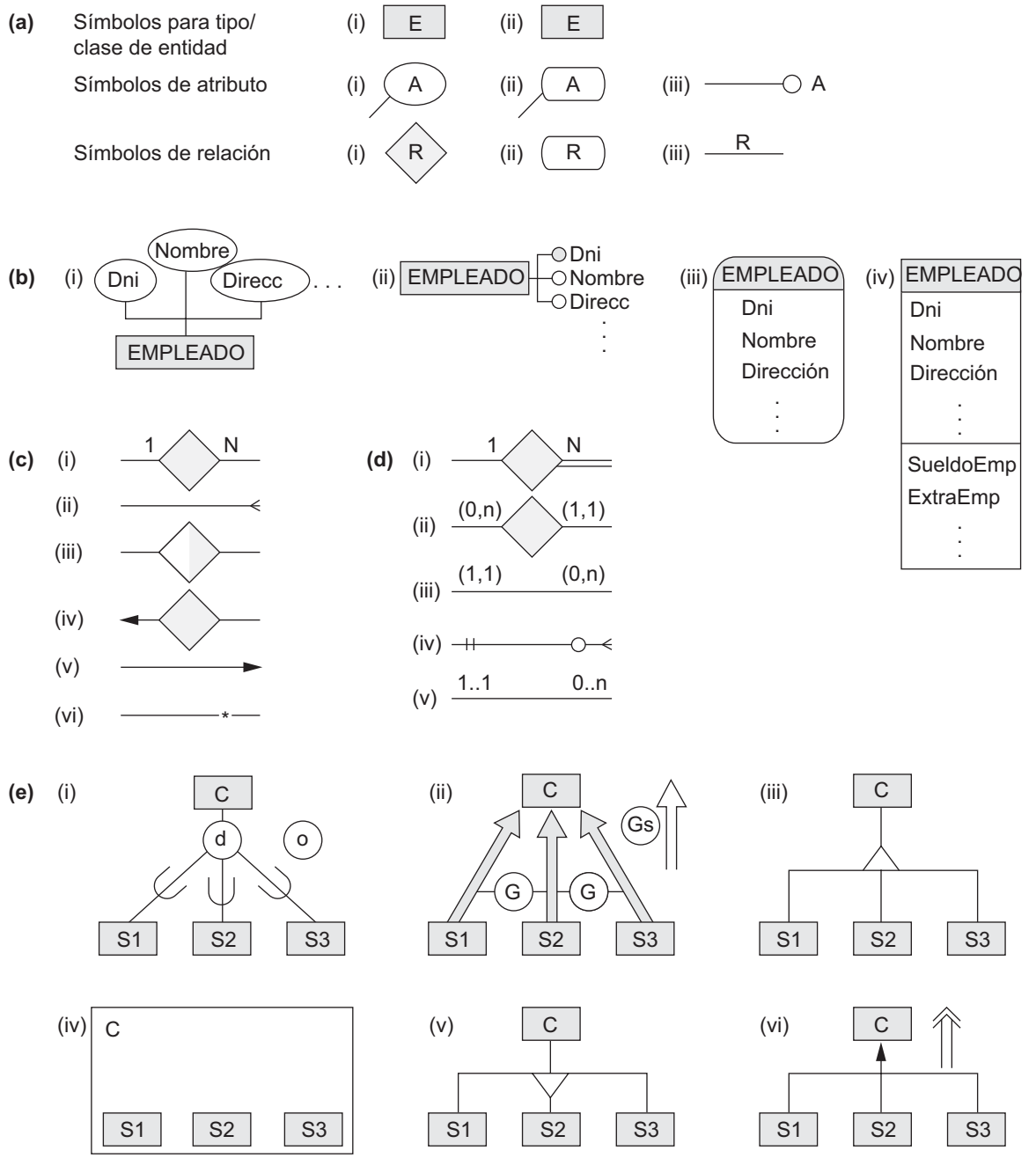
La Figura A.1 muestra varias notaciones diagramáticas para representar los conceptos de modelo ER y EER. Por desgracia, no hay una notación estándar: los diferentes diseñadores de bases de datos prefieren distintas notaciones. De forma parecida, las diferentes herramientas **CASE** (ingeniería de software computerizado) y metodologías **OOA** (análisis orientado a objetos) utilizan varias notaciones. Algunas notaciones están asociadas con modelos que cuentan con conceptos y restricciones adicionales, más allá de los descritos en los Capítulos 3 y 24 para los modelos ER y EER, mientras que otros modelos tienen menos conceptos y restricciones. La notación que utilizamos en el Capítulo 3 está muy cercana a la notación original de los diagramas ER, que todavía se utilizan ampliamente. Vamos a explicar algunas notaciones alternativas.

La Figura A.1(a) muestra diferentes notaciones para visualizar los tipos/clases de entidad, atributos y relaciones. En los Capítulos 3 y 24 utilizamos los símbolos marcados como (i) en la Figura A.1(a): rectángulo, óvalo y rombo. El símbolo (ii) para los tipos/clases de entidad, el símbolo (ii) para los atributos y el símbolo (ii) para las relaciones son similares, pero son utilizados por diferentes metodologías para representar tres conceptos diferentes. El símbolo de la línea recta (iii) para representar relaciones lo utilizan varias herramientas y metodologías.

La Figura A.1(b) muestra algunas notaciones para adjuntar atributos a los tipos de entidad. Utilizamos la notación (i). La notación (ii) utiliza la tercera notación (iii) para los atributos de la Figura A.1(a). Las dos últimas notaciones de la Figura A.1(b), (iii) y (iv), son populares en las metodologías OOA y en algunas herramientas CASE. En particular, la última notación muestra tanto los atributos como los métodos de una clase, separados por una línea horizontal.

La Figura A.1(c) muestra varias notaciones para representar la razón de cardinalidad de las relaciones binarias. Utilizamos la notación (i) en los Capítulos 3 y 24. La notación (ii), conocida como la notación *chicken feet* (patas de pollo) es muy popular. La notación (iv) utiliza la flecha como referencia funcional (desde el lado N al lado 1) y se parece a nuestra notación para las claves externas en el modelo relacional (véase la Figura 7.2); la notación (v), utilizada en los diagramas Bachman y en el modelo de datos de red, utiliza la flecha en la *dirección contraria* (desde el lado 1 hasta el lado N). Para una relación 1:1, (ii) utiliza una línea recta sin "patas de pollo"; (iii) divide en dos mitades el rombo blanco; y (iv) coloca puntas de flecha en los dos lados. Para una relación M:N, (ii) utiliza las "patas de pollo" en los dos extremos de la línea; (iii) divide en dos mitades el rombo negro; y (iv) no muestra puntas de flecha.

**Figura A.1.** Notaciones alternativas. (a) Símbolos para el tipo/clase de entidad, atributo y relación. (b) Visualización de atributos. (c) Visualización de razones de cardinalidad. (d) Varias notaciones (mín, máx). (e) Notaciones para visualizar la especialización/generalización.




La Figura A.1(d) muestra algunas variaciones de la visualización de restricciones (mín, máx), que se utilizan para visualizar la razón de cardinalidad y la participación total/parcial. Nosotros utilizamos principalmente (i). La notación (ii) es la notación alternativa que utilizamos en la Figura 3.15 y que explicamos en la Sección 3.7.4. Recuerde que nuestra notación especifica la restricción de que cada entidad debe participar en al menos

"mín" y en a lo sumo "máx" instancias de relación. Por tanto, para una relación 1:1, los dos valores máx son 1; para una relación M:N, los dos valores máx son n. Un valor mín mayor que 0 (cero) especifica una participación total (dependencia de existencia). En las metodologías que utilizan la línea recta para visualizar las relaciones, es frecuente *invertir la posición* de las restricciones (mín, máx), como se muestra en (iii); una variación común en algunas herramientas (y en la notación UML) se muestra en (v). Otra técnica popular, que sigue el mismo posicionamiento que (iii), es visualizar *mín* como o (letra "o" o un círculo, que representa el cero) o como | (barra vertical, que representa 1), y visualizar *máx* como | (barra vertical, que representa el 1) o como "patas de pollo" (que representa n), como se muestra en (iv).

La Figura A.1(e) muestra algunas notaciones para visualizar la especialización/generalización. Nosotros utilizamos (i) en el Capítulo 4, donde una d en el círculo especifica que las subclases (S1, S2 y S3) son disjuntas, y una o en el círculo especifica la superposición de subclases. La notación (ii) utiliza G (por "generalización") para especificar la disyunción, y Gs para especificar la superposición; algunas notaciones utilizan la flecha sólida, mientras que otras utilizan la flecha vacía (se muestra al lado). La notación (iii) utiliza un triángulo apuntando hacia la superclase, y la notación (v) utiliza un triángulo apuntando hacia las subclases; también es posible utilizar las dos notaciones en la misma metodología, con (iii) indicando la generalización y (v) indicando la especialización. La notación (iv) coloca los recuadros que representan las subclases dentro de la caja que representa la superclase. De las notaciones basadas en (vi), algunas utilizan una flecha de una sola línea, y otras utilizan una flecha de línea doble (se muestra al lado).

Las notaciones de la Figura A.1 sólo muestran algunos de los símbolos diagramáticos que se han utilizado o se han sugerido para visualizar el esquema conceptual de una base de datos. También se han utilizado otras notaciones, así como distintas combinaciones de las anteriores. Sería útil establecer un estándar al que cualquiera pudiera adherirse, a fin de evitar malentendidos y reducir la confusión.

# B

# Parámetros de disco

El parámetro más importante de un disco es el tiempo necesario para localizar un bloque de disco arbitrario, dada su dirección de bloque, y después transferir ese bloque desde el disco a un búfer de la memoria principal. Se trata del tiempo de acceso aleatorio para acceder a un bloque de disco. Es preciso considerar tres componentes de tiempo:

1. **Tiempo o tiempos de búsqueda.** Es el tiempo necesario para posicionar mecánicamente la cabeza de lectura/escritura en la pista correcta en los discos de cabeza móvil. (En los discos de cabeza fija, es el tiempo necesario para cambiar electrónicamente la cabeza de lectura/escritura apropiada.) En el caso de los discos de cabeza móvil, este tiempo varía, en función de la distancia entre la pista actual situada debajo de la cabeza de lectura/escritura y la pista especificada en la dirección del bloque. Normalmente, el fabricante del disco proporciona un tiempo de búsqueda medio en milisegundos. El rango típico del tiempo de búsqueda medio es de 10 a 60 mseg. Es el principal *culpable* del retraso en la transferencia de bloques entre el disco y la memoria.
2. **Retraso o retardo rotacional (*rr*).** Una vez que la cabeza de lectura/escritura se encuentra en la pista correcta, el usuario debe esperar a que el principio del bloque requerido rote o gire hasta situarse bajo la cabeza de lectura/escritura. Como promedio, esto lleva la mitad del tiempo necesario para una vuelta (revolución) del disco, pero realmente varía entre un acceso inmediato (si el inicio del bloque requerido ya se encuentra bajo la cabeza de lectura/escritura correcta después de la búsqueda) y una revolución o vuelta completa del disco (si el inicio del bloque requerido está justo pasada la posición bajo la cabeza de lectura/escritura después de la búsqueda). Si la velocidad de rotación del disco es de $p$ revoluciones por minuto (rpm), entonces el retraso rotacional medio $rr$ viene dado por esta fórmula:

   $$rr = (1/2)*(1/p) \text{ min} = (60*1000)/(2*p) \text{ mseg} = 30000/p \text{ mseg}$$

   Un valor típico de $p$ es 10.000 rpm, que arroja un retraso rotacional de $rr = 3$ mseg. En los discos de cabeza fija, donde el tiempo de búsqueda es insignificante, este componente provoca el retraso más grande en la transferencia de un bloque de disco.
3. **Tiempo de transferencia de bloque (*ttb*).** Una vez que la cabeza de lectura/escritura está al principio del bloque requerido, se necesita algo de tiempo para transferir los datos del bloque. Este tiempo de transferencia de bloques depende del tamaño del bloque, el tamaño de la pista y la velocidad de rotación. Si la **velocidad de transferencia** del disco es $vt$ bytes/mseg y el tamaño de bloque es $B$ bytes, entonces:

$ttb = B/vt$ mseg

Si tenemos un tamaño de pista de 50 Kbytes y $p$ es 3.600 rpm, entonces la velocidad de transferencia en bytes/mseg es:

$vt = (50*1000)/(60*1000/3600) = 3000$ bytes/mseg

En este caso, $ttb = B/3000$ mseg, donde $B$ es el tamaño de bloque en bytes.

El tiempo medio ($s$) necesario para encontrar y transferir un bloque, dada su dirección de bloque, queda estimado por:

$(s + rr + ttb)$ mseg

Esto se cumple tanto para la lectura como para la escritura de un bloque. El método principal para reducir este tiempo consiste en transferir varios bloques que están almacenados en una o más pistas del mismo cilindro; después, sólo se necesita el tiempo de búsqueda para el primer bloque. Para transferir consecutivamente *k bloques no contiguos* que se encuentran en el mismo cilindro, necesitamos aproximadamente:

$s + (k * (rr + ttb))$ mseg

En este caso, necesitamos dos o más búferes en el almacenamiento principal porque estamos leyendo o escribiendo de forma continua los $k$ bloques, como explicamos en el Capítulo 13. El tiempo de transferencia por bloque se reduce incluso más cuando se transfieren *bloques consecutivos* de la misma pista o cilindro. Esto elimina el retraso rotacional para todos los bloques, excepto el primero, por lo que lo estimado para transferir $k$ bloques consecutivos es:

$s + rr + (k * ttb)$ mseg

Una estimación más precisa para transferir bloques consecutivos tiene en cuenta el hueco entre bloques (consulte la Sección 13.2.1), que incluye la información que permite a la cabeza de lectura/escritura determinar el bloque que está a punto de leer. Normalmente, el fabricante del disco proporciona una **velocidad de transferencia bruta (vtb)** o global que tiene en cuenta el hueco entre bloques a la hora de leer bloques almacenados consecutivamente. Si el tamaño del hueco es de $G$ bytes, entonces:

$vtb = (B/(B + G)) * tr$ bytes/mseg

La velocidad de transferencia global es la velocidad de transferencia de los *bytes útiles* de los bloques de datos. La cabeza de lectura/escritura debe pasar por todos los bytes de una pista mientras gira el disco, incluyendo los bytes de los huecos entre bloques, que almacenan información de control pero no datos reales. Cuando se utiliza una velocidad de transferencia bruta, el tiempo necesario para transferir los datos útiles de un bloque fuera de varios bloques consecutivos es $B/vtb$. Por tanto, el tiempo estimado para leer $k$ bloques consecutivamente almacenados en el mismo cilindro se convierte en:

$s + rr + (k * (B/vtb))$ mseg

Otro parámetro de los discos es el **tiempo de reescritura**. Este tiempo resulta de utilidad cuando se lee un bloque del disco para almacenarlo en un búfer de la memoria principal, se actualiza el búfer y se vuelve a escribir el contenido del búfer de nuevo en el bloque de disco en el que estaba almacenado. En muchos casos, el tiempo requerido para actualizar el búfer de la memoria principal es inferior al tiempo requerido para una revolución del disco. Si sabemos que el búfer está listo para la reescritura, el sistema puede mantener las cabezas del disco en la misma pista, y durante la siguiente revolución del disco se reescribe el búfer actualizado en el bloque de disco. Por tanto, el tiempo de reescritura, *Trw*, se estima normalmente como el tiempo necesario para una revolución del disco:

$Trw = 2 * rr$ mseg $= 60000/p$ mseg

A modo de resumen, la siguiente lista enumera los parámetros que acabamos de explicar y los símbolos que se utilizan para ellos:

| | |
|---|---|
| Tiempo de búsqueda: | *s* mseg |
| Retraso rotacional: | *rr* mseg |
| Tiempo de transferencia de bloque: | *ttb* mseg |
| Tiempo de reescritura: | *Trw* mseg |
| Velocidad de transferencia: | *tr* bytes/mseg |
| Ratio o velocidad de transferencia aproximada: | *vtb* bytes/mseg |
| Tamaño de bloque: | *B* bytes |
| Tamaño del hueco entre bloques: | *G* bytes |
| Velocidad del disco: | *p* rpm (revoluciones por minuto) |

# C

# Introducción al lenguaje QBE

El lenguaje de Consulta mediante ejemplo (QBE) debe su importancia a que fue uno de los primeros lenguajes de consulta gráficos con una sintaxis mínima desarrollada para los sistemas de bases de datos. Fue desarrollado en IBM Research y está disponible como un producto comercial de IBM formando parte de la opción de interfaz QMF (*Query Management Facility*) para DB2. El lenguaje también está implementado en Paradox DBMS, y está relacionado con una interfaz de tipo "apuntar y hacer clic" del DBMS Microsoft Access (consulte el Capítulo 10). Difiere de SQL en que el usuario no tiene que especificar explícitamente una consulta utilizando una sintaxis fija; en su lugar, la consulta se formula rellenando **plantillas** de relaciones que se visualizan en el monitor. La Figura C.1 muestra el aspecto de estas plantillas para la base de datos de la Figura 7.1. El usuario no tiene que recordar los nombres de los atributos o de las relaciones, porque dichos nombres aparecen como parte de las plantillas. Además, no es preciso seguir unas reglas de sintaxis rígidas para especificar la consulta; en su lugar, se introducen constantes y variables en las columnas de las plantillas para construir un **ejemplo** relacionado con la consulta de recuperación o actualización. QBE está relacionado con el cálculo relacional de dominio, como veremos, y su especificación original ha mostrado ser relacionalmente completa.

## C.1 Recuperaciones básicas en QBE

En QBE, las consultas de recuperación se especifican rellenando una o más filas de las plantillas de las tablas. En una consulta de una sola relación, introducimos constantes o **elementos de ejemplo** (un término QBE) en las columnas de la plantilla de esa relación. Un elemento de ejemplo simboliza una variable de dominio y se especifica como un valor de ejemplo precedido por el carácter de subrayado (_). Además, se introduce un prefijo P. (denominado operador de punto P) en determinadas columnas para indicar que nos gustaría imprimir (o visualizar) los valores de esas columnas para nuestro resultado. Las constantes especifican valores que deben ser exactamente iguales a los presentes en esas columnas.

Por ejemplo, considere la consulta C0: *Recuperar la fecha de nacimiento y la dirección de José Pérez Pérez*. Las Figuras C.2(a) a C.2(d) muestran cómo debemos especificar esta consulta de una forma progresivamente más concisa en QBE. La Figura C.2(a) presenta un ejemplo de un empleado como el tipo de fila en que estamos interesados. Dejando "José Pérez Pérez" como constantes de las columnas Nombre, Apellido1 y Apellido2, estamos especificando una coincidencia exacta para esas columnas. El resto de las columnas van precedidas por un guión de subrayado para indicar que son variables de dominio (elementos de ejemplo).

**Figura C.1.** Esquema relacional de la Figura 5.5 como QBE podría visualizarlo.

EMPLEADO

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

DEPARTAMENTO

| NombreDpto | NúmeroDpto | DniDirector | FechaIngresoDirector |
|---|---|---|---|
| | | | |

LOCALIZACIONES_DPTO

| NúmeroDpto | UbicaciónDpto |
|---|---|
| | |

PROYECTO

| NombreProyecto | NumProyecto | UbicaciónProyecto | NumDptoProyecto |
|---|---|---|---|
| | | | |

TRABAJA_EN

| DniEmpleado | NumProy | Horas |
|---|---|---|
| | | |

SUBORDINADO

| DniEmpleado | NombSubordinado | Sexo | FechaNac | Relación |
|---|---|---|---|---|
| | | | | |

Colocamos el prefijo P. en las columnas FechaNac y Dirección para indicar que nos gustaría disponer de valores de salida en esas columnas.

La consulta C0 puede abreviarse como se muestra en la Figura C.2(b). No es necesario especificar valores de ejemplo para las columnas que no nos interesan. Además, como los valores de ejemplo son completamente arbitrarios, podemos introducir nombres variables para ellos, como se aprecia en la Figura C.2(c). Por último, también podemos omitir los valores de ejemplo, como muestra la Figura C.2(d), y especificar únicamente P. en las columnas a recuperar.

Para ver cuán similares son las consultas de recuperación en QBE al cálculo relacional de dominio, compare la Figura C.2(d) con C0 (simplificada) en el cálculo de dominio:

**C0 :** { uv | EMPLEADO(qrstuvwxyz) **and** *q*='José' **and** *r*='Pérez' **and** *s*='Pérez'}

Podemos asemejar cada columna de una plantilla QBE a una *variable de dominio implícita*; por tanto, Nombre se corresponde con la variable de dominio *q*, Apellido1 se corresponde con *r*, ..., y Dno se corresponde con *z*. En la consulta QBE, las columnas con P. se corresponden a las variables especificadas a la izquierda de la barra en el cálculo de dominio, mientras que las columnas con valores constantes se corresponden a las variables de tupla con condiciones de selección de igualdad. La condición EMPLEADO(qrstuvwxyz) y los cuantificadores existenciales están implícitos en la consulta QBE porque se utiliza la plantilla correspondiente a la relación EMPLEADO.

En QBE, la interfaz de usuario primero permite al usuario elegir las tablas (relaciones) necesarias para formular una consulta visualizando una lista con los nombres de todas las relaciones. Después, aparecen las plantillas para las relaciones elegidas. El usuario se desplaza a las columnas adecuadas de las plantillas y especifica la consulta. Se suministran unas teclas de función especiales para moverse por las plantillas y ejecutar determinadas funciones.

Vamos a ofrecer ahora algunos ejemplos para ilustrar lo básico de QBE. En una columna se pueden introducir operadores de comparación distintos a = (como > o ≻) antes de escribir un valor constante. Por ejemplo,

**Figura C.2.** Cuatro formas de especificar la consulta C0 en QBE.

**(a)** **EMPLEADO**

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|
| José | Pérez | Pérez | _123456789 | P._1/9/60 | P._Gonzalo, 10 | _H | _25000 | _123456789 | _3 |

**(b)** **EMPLEADO**

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|
| José | Pérez | Pérez | | P._1/9/60 | P._Gonzalo, 10 | | | | |

**(c)** **EMPLEADO**

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|
| José | Pérez | Pérez | | P._X | P._Y | | | | |

**(d)** **EMPLEADO**

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|
| José | Pérez | Pérez | | P. | P. | | | | |

**Figura C.3.** Especificación de condiciones complejas en QBE. (a) La consulta C0A. (b) La consulta C0B con un cuadro de condición. (c) La consulta C0B sin un cuadro de condición.

**(a)** **TRABAJA_EN**

| DniEmpleado | NumProy | Horas |
|---|---|---|
| P. | | >20 |

**(b)** **TRABAJA_EN**

| DniEmpleado | NumProy | Horas |
|---|---|---|
| P. | _PX | _HX |

**CONDICIONES**

| _HX>20 and (PX=1 OR PX=2) |
|---|

**(c)** **TRABAJA_EN**

| DniEmpleado | NumProy | Horas |
|---|---|---|
| P. | 1 | >20 |
| P. | 2 | >20 |

la consulta C0A: "*Listar los números del documento nacional de identidad de los empleados que trabajan más de 20 horas a la semana en el proyecto número 1*" puede especificarse como se muestra en la Figura C.3(a). En el caso de condiciones más complejas, podemos solicitar un **cuadro de condición**, que se crea pulsando una tecla de función concreta. Después, ya podemos escribir la condición compleja.[1]

Por ejemplo, la consulta C0B: "Listar el número del documento nacional de identidad de los empleados que trabajan más de 20 horas a la semana en el proyecto 1 o en el proyecto 2" puede especificarse como se muestra en la Figura C.3(b).

[1] En un cuadro de condición no está permitida la negación con el símbolo ¬.

Algunas condiciones complejas pueden especificarse sin un cuadro de condición. La regla es que todas las condiciones especificadas en la misma fila de una plantilla de relación estén conectadas con un conector lógico **and** (debe cumplirse todo para que la tupla sea seleccionada), mientras que las condiciones especificadas en filas diferentes deben estarlo mediante **or** (al menos una condición debe cumplirse). Por tanto, C0B también puede especificarse, como se muestra en la Figura C.3(c), introduciendo dos filas distintas en la plantilla.

Ahora, consideremos la consulta C0C: "Listar el número del documento nacional de identidad de los empleados que trabajan tanto en el proyecto 1 como en el proyecto 2"; esto no puede especificarse como en la Figura C.4(a), que lista todos los que trabajan en alguno de los dos proyectos: el proyecto 1 o el proyecto 2. La variable de ejemplo _ES se asociará a los valores DniEmpleado en las tuplas <-, 1, -> así como a los de las tuplas <-, 2, ->. La Figura C.4(b) muestra cómo especificar correctamente la consulta C0C, donde la condición (_EX = _EY) del cuadro hace que las variables _EX y _EY se enlacen únicamente a valores DniEmpleado idénticos.

En general, una vez especificada una consulta, los valores resultantes se muestran en la plantilla bajo las columnas adecuadas. Si el resultado contiene más filas que las que encajan en pantalla, la mayoría de las implementaciones de QBE tienen teclas de función que permiten desplazar las filas arriba y abajo. De forma parecida, si una o varias plantillas son demasiado anchas para aparecer en pantalla, es posible desplazarse horizontalmente para examinarlas todas.

Una operación de concatenación se especifica en QBE utilizando la misma variable[2] en las columnas que van a concatenarse. Por ejemplo, la consulta C1: "Listar el nombre y la dirección de todos los empleados que trabajan para el departamento 'Investigación'" puede especificarse como se muestra en la Figura C.5(a). En una sola consulta puede especificarse cualquier cantidad de concatenaciones. También es posible especificar una **tabla de resultado** para visualizar el resultado de la consulta de concatenación (véase la Figura C.5[a]); esto es necesario si el resultado incluye atributos de dos o más relaciones. Si no se especifica una tabla de resultado, el sistema proporciona el resultado de la consulta en las columnas de las distintas relaciones, cuya interpretación puede resultar más compleja. La Figura C.5(a) también ilustra la característica de QBE para especificar que todos los atributos de una relación deben ser recuperados, colocando el operador P. debajo del nombre de la relación en la plantilla de la misma.

**Figura C.4.** Listado de los EMPLEADOS que trabajan en los dos proyectos. (a) Especificación incorrecta de una condición AND. (b) Especificación correcta.

(a)

TRABAJA_EN

| DniEmpleado | NumProy | Horas |
|---|---|---|
| P._ES | 1 | |
| P._ES | 2 | |

(b)

TRABAJA_EN

| DniEmpleado | NumProy | Horas |
|---|---|---|
| P._EX | 1 | |
| P._EY | 2 | |

CONDICIONES

_EX = _EY

[2] En los manuales de QBE una variable se denomina **elemento de ejemplo**.

**Figura C.5.** Ilustración de JOIN y relaciones resultantes en QBE. (a) Consulta C1. (b) Consulta C8.

**(a)** EMPLEADO

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|
| _NP | | _AP | | | _Direct | | | | _DX |

DEPARTMENTO

| NombreDpto | NúmeroDpto | DniDirector | FechaIngresoDirector |
|---|---|---|---|
| Investigación | _DX | | |

| RESULTADO | | | |
|---|---|---|---|
| P. | _NP | _AP | _Direct |

**(b)** EMPLEADO

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|
| _E1 | | _E2 | | | | | | _Xdni | |
| _S1 | | _S2 | _Xdni | | | | | | |

| RESULTADO | | | | |
|---|---|---|---|---|
| P. | _E1 | _E2 | _S1 | _S2 |

Para concatenar una tabla consigo misma, especificamos distintas variables para representar las diferentes referencias a la tabla. Por ejemplo, la consulta C8: "*Por cada empleado, recuperar su nombre y su primer apellido, así como el nombre y primer apellido de su supervisor inmediato*" puede especificarse como muestra la Figura C.5(b), donde las variables que empiezan por E se refieren a un empleado y las que empiezan por S se refieren a un supervisor.

# C.2 Agrupamiento, agregación y modificación de la base de datos en QBE

A continuación, vamos a considerar los tipos de consultas que requieren funciones de agrupamiento o de agregación. Un operador de agrupamiento *G*. puede especificarse en una columna para indicar que esas tuplas deben agruparse por el valor de esa columna. Es posible especificar funciones comunes como AVG., SUM., CNT. (contar), MAX. y MIN. En QBE las funciones AVG., SUM. y CNT. se aplican a valores distintos dentro de un grupo en el caso predeterminado. Si queremos que estas funciones se apliquen a todos los valores, debemos utilizar el prefijo ALL.[3] Esta convención es *diferente* en SQL, donde lo predeterminado es aplicar una función a todos los valores.

La Figura C.6(a) muestra la consulta C23, que cuenta el número de valores de sueldo *distintos* de la relación EMPLEADO. La consulta C23A [Figura C.6(b)] cuenta todos los valores de sueldo, que es lo mismo que contar el número de empleados. La Figura C.6(c) muestra la consulta C24, que recupera el número de cada departamento y el número de empleados y el sueldo medio dentro de cada departamento; por tanto, la columna Dno se utiliza para agrupar como lo indica la función *G*. En una columna podemos especificar varios de los operadores G., P. y ALL. La Figura C.6(d) muestra la consulta C26, que visualiza, por cada proyecto en el que trabajan más de dos empleados, el nombre del proyecto y el número de empleados que trabajan en él.

[3] ALL en QBE no está relacionado con el cuantificador universal.

**Figura C.6.** Funciones y agrupamiento en QBE. (a) Consulta C23. (b) Consulta C23A. (c) Consulta C24. (d) Consulta C26.

**(a)** EMPLEADO

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | P.CNT. | | |

**(b)** EMPLEADO

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | P.CNT.ALL | | |

**(c)** EMPLEADO

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|
| | | | P.CNT.ALL | | | | P.AVG.ALL | | P.G. |

**(d)** PROYECTO

| NombreProyecto | NumProyecto | UbicaciónProyecto | NumDptoProyecto |
|---|---|---|---|
| P. | _PX | | |

TRABAJA_EN

| DniEmpleado | NumProy | Horas |
|---|---|---|
| P.CNT.EX | G._PX | |

CONDICIONES

| CNT._EX > 2 |
|---|

QBE tiene un símbolo de negación, ¬, que se utiliza de una forma parecida a la función NOT EXISTS de SQL. La Figura C.7 muestra la consulta C6, que lista los nombres de los empleados que no tienen subordinados. El símbolo de negación ¬ dice que seleccionaremos los valores de la variable _SX de la relación EMPLEADO sólo si no tienen lugar en la relación SUBORDINADO. El mismo efecto puede producirse colocando ¬ _SX en la columna DniEmpleado.

Aunque el lenguaje QBE como originalmente se propuso soportaba el equivalente de las funciones EXISTS y NOT EXISTS de SQL, la implementación QBE de QMF (bajo el sistema DB2) *no* ofrece este soporte. Por tanto, la versión QMF de QBE, que aquí explicamos, *no es relacionalmente completa*, de modo que no es posible especificar consultas como la C3: "*Encontrar los empleados que trabajan en todos los proyectos controlados por el departamento 5*".

En QBE hay tres operadores para modificar la base de datos: I. para insertar, D. para eliminar y U. para actualizar. Los operadores de inserción y eliminación se especifican en la columna de plantilla bajo el nombre de la relación, mientras que el operador de actualización se especifica bajo las columnas a modificar. La Figura C.8(a) muestra la inserción de una nueva tupla EMPLEADO. En el caso de la eliminación, primero introducimos el operador D. y después especificamos, mediante una condición, las tuplas que vamos a eliminar [Figura C.8(b)]. Para actualizar una tupla, especificamos el operador U. debajo del nombre del atributo, seguido por el valor nuevo de dicho atributo. También debemos seleccionar la tupla o tuplas que queremos actualizar de la forma habitual. La Figura C.8(c) muestra una solicitud de actualización para aumentar el sueldo de José Pérez un 10 por ciento, a la vez que le reasignamos al departamento número 4.

QBE también permite la definición de datos. Las tablas de una base de datos pueden especificarse interactivamente, y la definición de una tabla puede actualizarse añadiendo, renombrando o eliminando una columna.

**Figura C.7.** Ilustración de la negación por la consulta C6.

EMPLEADO

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|
| P. | | P. | _SX | | | | | | |

SUBORDINADO

| ¬ | DniEmpleado | NombSubordinado | Sexo | FechaNac | Relación |
|---|---|---|---|---|---|
| | _SX | | | | |

**Figura C.8.** Modificación de la base de datos en QBE. (a) Inserción. (b) Eliminación. (c) Actualización en QBE.

**(a)** EMPLEADO

| | Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|---|
| I. | Ricardo | Marino | García | 653298653 | 30-dic-52 | Delicias, 34 | M | 37000 | 987654321 | 4 |

**(b)** EMPLEADO

| | Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|---|
| E. | | | | 653298653 | | | | | | |

**(c)** EMPLEADO

| Nombre | Apellido1 | Apellido2 | Dni | FechaNac | Dirección | Sexo | Sueldo | SuperDni | Dno |
|---|---|---|---|---|---|---|---|---|---|
| José | | Pérez | | | | | U._S*1.1 | | U.4 |

También podemos especificar varias características para cada columna, como si es una clave de la relación, su tipo de datos, y si debe crearse un índice en ese campo. QBE también tiene funciones para definir vistas, temas de autorización y almacenamiento de definiciones de consulta para un uso posterior, etcétera.

QBE no utiliza el estilo *lineal* de SQL; en su lugar, es un lenguaje *bidimensional* porque la consulta se especifica moviéndose por todo el área de la pantalla. Diversas pruebas realizadas con usuarios han mostrado que QBE es más fácil de aprender que SQL, especialmente para los usuarios menos experimentados. En este sentido, QBE ha sido el primer lenguaje de bases de datos relacional visual amigable para el usuario.

Más recientemente, se han desarrollado muchas otras interfaces visuales para los sistemas de bases de datos comerciales. El uso de menús, gráficos y formularios es ahora muy común. El rellenado parcial de formularios es una solicitud de búsqueda que se parece a utilizar QBE. Los lenguajes de consulta visuales, que todavía no son demasiado comunes, probablemente se ofrecerán en el futuro con las bases de datos relacionales comerciales.

# Bibliografía seleccionada

## Abreviaturas utilizadas en la bibliografía

ACM: Association for Computing Machinery.

AFIPS: American Federation of Information Processing Societies.

CACM: Communications of the ACM (publicación).

CIKM: Acrónimo de International Conference on Information and Knowledge Management.

DASFAA: Acrónimo de International Conference on Database Systems for Advanced Applications.

DKE: Data and Knowledge Engineering, Elsevier Publishing (publicación).

EDS: Acrónimo de International Conference on Expert Database Systems.

ER, Conferencia: Acrónimo de International Conference on Entity-Relationship Approach (ahora denominada International Conference on Conceptual Modeling).

ICDCS: Acrónimo de IEEE International Conference on Distributed Computing Systems.

ICDE: Acrónimo de IEEE International Conference on Data Engineering.

IEEE: Institute of Electrical and Electronics Engineers.

IEEE Computer: revista Computer (publicación) de la IEEE CS.

IEEE CS: IEEE Computer Society.

IFIP: International Federation for Information Processing.

JACM: Publicación de la ACM.

KDD: Knowledge Discovery in Databases.

LNCS: Lecture Notes in Computer Science.

NCC: Acrónimo de National Computer Conference (publicada por la AFIPS).

OOPSLA: Acrónimo de ACM Conference on Object-Oriented Programming Systems, Languages, and Applications.

PODS: Acrónimo de ACM Symposium on Principles of Database Systems.

SIGMOD: Acrónimo de ACM SIGMOD International Conference on Management of Data.

TKDE: IEEE Transactions on Knowledge and Data Engineering (publicación).

TOCS: ACM Transactions on Computer Systems (publicación).

TODS: ACM Transactions on Database Systems (publicación).

TOIS: ACM Transactions on Information Systems (publicación).

TOOIS: ACM Transactions on Office Information Systems (publicación).

TSE: IEEE Transactions on Software Engineering (publicación).

VLDB: Acrónimo de International Conference on Very Large Data Bases (los ejemplares posteriores a 1981 los tienen disponibles en Morgan Kaufmann, Menlo Park, California).

## Formato de las reseñas bibliográficas

Los títulos de los libros aparecen en negrita; por ejemplo, **Database Computers**. Los nombres procedentes de conferencias aparecen en cursiva; por ejemplo, *ACM Pacific Conference*. Los nombres de las publicaciones también aparecen en negrita; por ejemplo, **TODS** o **Information Systems**. En las menciones de publicaciones, ofrecemos el número de volumen y el número de ejemplar (dentro del volumen, si lo hay) y la fecha de éste. Por ejemplo, "**TODS**, 3:4, diciembre de 1978" se refiere al ejemplar de diciembre de 1978 de la *ACM Transactions on Database Systems*, correspondiente al Volumen 3, Número 4. Los artículos que aparecen en libros o que proceden de una conferencia que se citan como tales en la bibliografía, aparecen como "en" en estas referencias; por ejemplo, "en *VLDB* [1978]" o "en Rustin [1974]". Cuando son aplicables, los números

de página se ofrecen con la abreviatura págs. al final de la reseña. En las reseñas con más de cuatro autores, sólo aparece el primero de ellos, seguido por la expresión "y otros". En la bibliografía seleccionada que aparece al final de cada capítulo utilizamos "y otros" si son más de dos autores.

## Referencias bibliográficas


Abbott, R. y García-Molina, H. [1989] "Scheduling Real-Time Transactions with Disk Resident Data", en *VLDB* [1989].

Abiteboul, S. y Kanellakis, P. [1989] "Object Identity as a Query Language Primitive", en *SIGMOD* [1989].

Abiteboul, S., Hull, R. y Vianu, V. [1995] **Foundations of Databases**, Addison-Wesley, 1995.

Abrial, J. [1974] "Data Semantics", en Klimbie and Koffeman [1974].

Acharya, S., Alonso, R., Franklin, M. y Zdonik, S. [1995] "Broadcast Disks: Data Management for Asymmetric Communication Environments", en *SIGMOD* [1995].

Adam, N. y Gongopadhyay, A. [1993] "Integrating Functional and Data Modeling in a Computer Integrated Manufacturing System", en *ICDE* [1993].

Adriaans, P. y Zantinge, D. [1996] **Data Mining**, Addison-Wesley, 1996.

Afsarmanesh, H., McLeod, D., Knapp, D. y Parker, A. [1985] "An Extensible Object-Oriented Approach to Databases for VLSI/CAD", en *VLDB* [1985].

Agrawal, D. y El Abbadi, A. [1990] "Storage Efficient Replicated Databases", **TKDE**, 2:3, septiembre de 1990.

Agrawal, R. y Gehani, N. [1989] "ODE: The Language and the Data Model", en *SIGMOD* [1989].

Agrawal, R., Gehani, N. y Srinivasan, J. [1990] "OdeView: The Graphical Interface to Ode", en *SIGMOD* [1990].

Agrawal, R., Imielinski, T. y Swami, A. [1993] "Mining Association Rules Between Sets of Items in Databases", en *SIGMOD* [1993].

Agrawal, R., Imielinski, T. y Swami, A. [1993b] "Database Mining: A Performance Perspective", **TKDE** 5:6, diciembre de 1993.

Agrawal, R., Mehta, M., Shafer, J. y Srikant, R. [1996] "The Quest Data Mining System", en KDD [1996].

Agrawal, R. y Srikant, R. [1994] "Fast Algorithms for Mining Association Rules in Large Databases", en *VLDB* [1994].

Ahad, R. y Basu, A. [1991] "ESQL: A Query Language for the Relational Model Supporting Image Domains", en *ICDE* [1991].

Aho, A., Beeri, C. y Ullman, J. [1979] "The Theory of Joins in Relational Databases", **TODS**, 4:3, septiembre de 1979.

Aho, A., Sagiv, Y. y Ullman, J. [1979a] "Efficient Optimization of a Class of Relational Expressions", **TODS**, 4:4, diciembre de 1979.

Aho, A. y Ullman, J. [1979] "Universality of Data Retrieval Languages", *Proc. POPL Conference,* San Antonio, TX, ACM, 1979.

Akl, S. [1983] "Digital Signatures: A Tutorial Survey", **IEEE Computer**, 16:2, febrero de 1983.

Alagic, S. [1999], "A Family of the ODMG Object Models", en Advances in Databases and Information Systems, Third East European Conference, ADBIS'99, Maribor, Slovenia, J. Eder, I. Rozman, T. Welzer (eds.), septiembre de 1999, LNCS, Núm. 1691, Springer.

Alashqur, A., Su, S. y Lam, H. [1989] "OQL: A Query Language for Manipulating Object-Oriented Databases", en *VLDB* [1989].

Albano, A., Cardelli, L. y Orsini, R. [1985] "GALILEO: A Strongly Typed Interactive Conceptual Language", **TODS**, 10:2, junio de 1985.

Allen, F., Loomis, M. y Mannino, M. [1982] "The Integrated Dictionary/Directory System", **ACM Computing Surveys**, 14:2, junio de 1982.

Alonso, G., Agrawal, D., El Abbadi, A. y Mohan, C. [1997] "Functionalities and Limitations of Current Workflow Management Systems", **IEEE Expert**, 1997.

Amir, A., Feldman, R. y Kashi, R. [1997] "A New and Versatile Method for Association Generation", en **Information Systems**, 22:6, septiembre de 1997.

Anderson, S., Bankier, A., Barrell, B., de Bruijn, M., Coulson, A., Drouin, J., Eperon, I., Nierlich, D., Rose, B., Sanger, F., Schreier, P., Smith, A., Staden, R. y Young, I. [1981] "Sequence and

Organization of the Human Mitochondrial Genome". **Nature**, 290: 457–465, 1981.

Andrews, T. y Harris, C. [1987] "Combining Language and Database Advances in an Object-Oriented Development Environment", *OOPSLA*, 1987.

ANSI [1975] American National Standards Institute Study Group on Data Base Management Systems: Interim Report, FDT, 7:2, ACM, 1975.

ANSI [1986] American National Standards Institute: **The Database Language SQL**, Documento ANSI X3.135, 1986.

ANSI [1986a] American National Standards Institute: **The Database Language NDL**, Documento ANSI X3.133, 1986.

ANSI [1989] American National Standards Institute: **Information Resource Dictionary Systems**, Documento ANSI X3.138, 1989.

Antenucci, J. y otros [1998] **Geographic Information Systems: A Guide to the Technology**, Chapman and Hall, mayo de 1998.

Anwar, T., Beck, H. y Navathe, S. [1992] "Knowledge Mining by Imprecise Querying: A Classification Based Approach", en *ICDE* [1992].

Apers, P., Hevner, A. y Yao, S. [1983] "Optimization Algorithms for Distributed Queries", **TSE**, 9:1, enero de 1983.

Apweiler, R., Martin, M., O'Donovan, C. y Prues, M. [2003] "Managing Core Resources for Genomics and Proteomics", **Pharmacogenomics**, 4:3, mayo de 2003, págs. 343–350.

Arisawa, H. y Catarci, T. [2000] Advances in Visual Information Management, *Proc. Fifth Working Conf. On Visual Database Systems,* Arisawa, H., Catarci, T. (eds.), Fujkuoka, Japón, *IFIP Conference Proceedings 168*, Kluwer, 2000.

Armstrong, W. [1974] "Dependency Structures of Data Base Relationships", *Proc. IFIP Congress,* 1974.

Ashburner, M., Eppig, J. y otros [2000] "Gene Ontology: Tool for the unification of biology", **Nature Genetics**, Vol. 25, págs. 25–29, mayo de 2000.

Astrahan, M. y otros [1976] "System R: A Relational Approach to Data Base Management", **TODS**, 1:2, junio de 1976.

Atkinson, Malcolm y otros [1990] The Object-Oriented Database System Manifesto, *Proc. Deductive and Object Oriented Database Conf. (DOOD)*, Kyoto, Japan, 1990.

Atkinson, M. y Buneman, P. [1987] "Types and Persistence in Database Programming Languages", en **ACM Computing Surveys**, 19:2, junio de 1987.

Atluri, V., Jajodia, S., Keefe, T.F., McCollum, C. y Mukkamala, R. [1997] "Multilevel Secure Transaction Processing: Status and Prospects", en **Database Security: Status and Prospects**, Chapman and Hall, 1997, págs. 79–98.

Atzeni, P. y De Antonellis, V. [1993] **Relational Database Theory**, Benjamin/Cummings, 1993.

Atzeni, P., Mecca, G. y Merialdo, P. [1997] "To Weave the Web", en *VLDB* [1997].

Bachman, C. [1969] "Data Structure Diagrams", **Data Base** (Bulletin of ACM SIGFIDET), 1:2, marzo de 1969.

Bachman, C. [1973] "The Programmer as a Navigator", **CACM**, 16:1, noviembre de 1973.

Bachman, C. [1974] "The Data Structure Set Model", en Rustin [1974].

Bachman, C. y Williams, S. [1964] "A General Purpose Programming System for Random Access Memories", *Proc. Fall Joint Computer Conference,* AFIPS, 26, 1964.

Badal, D. y Popek, G. [1979] "Cost and Performance Analysis of Semantic Integrity Validation Methods", en *SIGMOD* [1979].

Badrinath, B., Imielinski, T. [1992] Replication and Mobility, *Proc. Workshop on the Management of Replicated Data 1992*: págs. 9–12.

Badrinath, B. y Ramamritham, K. [1992] "Semantics-Based Concurrency Control: Beyond Commutativity", **TODS**, 17:1, marzo de 1992.

Baeza-Yates, R. y Larson, P. A. [1989] "Performance of B1-trees with Partial Expansions", **TKDE**, 1:2, junio de 1989.

Baeza-Yates, R. y Ribero-Neto, B. [1999] **Modern Information Retrieval**, Addison-Wesley, 1999.

Balbin, I. y Ramamohanrao, K. [1987] "A Generalization of the Different Approach to Recursive Query Evaluation", **Journal of Logic Programming**, 15:4, 1987.

Bancilhon, F. y Buneman, P., eds. [1990] **Advances in Database Programming Languages**, ACM Press, 1990.

Bancilhon, F., Delobel, C. y Kanellakis, P., eds. [1992] **Building an Object-Oriented Database System: The Story of O2**, Morgan Kaufmann, 1992.

Bancilhon, F. y Ferran, G. [1995], "The ODMG Standard for Object Databases", *DASFAA 1995*, Singapore, págs. 273–283.

Bancilhon, F., Maier, D., Sagiv, Y. y Ullman, J. [1986] "Magic Sets and Other Strange Ways to Implement Logic Programs", *PODS* [1986].

Bancilhon, F. y Ramakrishnan, R. [1986] "An Amateur's Introduction to Recursive Query Processing Strategies", en *SIGMOD* [1986].

Banerjee, J. y otros [1987] "Data Model Issues for Object-Oriented Applications", **TOOIS**, 5:1, enero de 1987.

Banerjee, J., Kim, W., Kim, H. y Korth, H. [1987a] "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", en *SIGMOD* [1987].

Barbara, D. [1999] "Mobile Computing and Databases – A Survey", **TKDE**, 11:1, enero de 1999.

Baroody, A. y DeWitt, D. [1981] "An Object-Oriented Approach to Database System Implementation", **TODS**, 6:4, diciembre de 1981.

Barrett T. y otros "NCBI GEO: mining millions of expression profiles—database and tools", **Nucleic Acid Research**, 33: database issue, 2005, págs. 562–566.

Barsalou, T., Siambela, N., Keller, A. y Wiederhold, G. [1991] "Updating Relational Databases Through Object-Based Views", en *SIGMOD* [1991].

Bassiouni, M. [1988] "Single-Site and Distributed Optimistic Protocols for Concurrency Control", **TSE**, 14:8, agosto de 1988.

Batini, C., Ceri, S. y Navathe, S. [1992] **Database Design: An Entity-Relationship Approach**, Benjamin/Cummings, 1992.

Batini, C., Lenzerini, M. y Navathe, S. [1987] "A Comparative Analysis of Methodologies for Database Schema Integration", **ACM Computing Surveys**, 18:4, diciembre de 1987.

Batory, D. y Buchmann, A. [1984] "Molecular Objects, Abstract Data Types y Data Models: A Framework", en *VLDB* [1984].

Batory, D. y otros [1988] "GENESIS: An Extensible Database Management System", **TSE**, 14:11, noviembre de 1988.

Bayer, R., Graham, M. y Seegmuller, G., eds. [1978] **Operating Systems: An Advanced Course**, Springer-Verlag, 1978.

Bayer, R. y McCreight, E. [1972] "Organization and Maintenance of Large Ordered Indexes", **Acta Informatica**, 1:3, febrero de 1972.

Beck, H., Anwar, T. y Navathe, S. [1994] "A Conceptual Clustering Algorithm for Database Schema Design", **TKDE**, 6:3, junio de 1994.

Beck, H., Gala, S. y Navathe, S. [1989] "Classification as a Query Processing Technique in the CANDIDE Semantic Data Model", en *ICDE* [1989].

Beeri, C., Fagin, R. y Howard, J. [1977] "A Complete Axiomatization for Functional and Multivalued Dependencies", en *SIGMOD* [1977].

Beeri, C. y Ramakrishnan, R. [1987] "On the Power of Magic" en *PODS* [1987].

Benson, D., Boguski, M., Lipman, D. y Ostell, J., "GenBank", **Nucleic Acids Research**, 24:1, 1996.

Benson, D., Karsch-Mizrachi, I., Lipman, D. y otros [2002] "GenBank", **Nucleic Acids Research**, 30: 1, págs. 17–20, enero de 2002.

Ben-Zvi, J. [1982] "The Time Relational Model", Ph.D. dissertation, University of California, Los Ángeles, 1982.

Berg, B. y Roth, J. [1989] **Software for Optical Disk**, Meckler, 1989.

Berners-Lee, T., Caillian, R., Grooff, J. y Pollermann, B. [1992] "World-Wide Web: The Information Universe", **Electronic Networking: Research**, **Applications and Policy**, 1:2, 1992.

Berners-Lee, T., Caillian, R., Lautonen, A., Nielsen, H. y Secret, A. [1994] "The World Wide Web", **CACM**, 13:2, agosto de 1994.

Bernstein, P. [1976] "Synthesizing Third Normal Form Relations from Functional Dependencies", **TODS**, 1:4, diciembre de 1976.

Bernstein, P., Blaustein, B. y Clarke, E. [1980] "Fast Maintenance of Semantic Integrity Assertions

Using Redundant Aggregate Data", en *VLDB* [1980].

Bernstein, P. y Goodman, N. [1980] "Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems", en *VLDB* [1980].

Bernstein, P. y Goodman, N. [1981] "The Power of Natural Semijoins", **SIAM Journal of Computing**, 10:4, diciembre de 1981.

Bernstein, P. y Goodman, N. [1981a] "Concurrency Control in Distributed Database Systems", **ACM Computing Surveys**, 13:2, junio de 1981.

Bernstein, P. y Goodman, N. [1984] "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases", **TODS**, 9:4, diciembre de 1984.

Bernstein, P., Hadzilacos, V. y Goodman, N. [1988] **Concurrency Control and Recovery in Database Systems**, Addison-Wesley, 1988.

Bertino, E. [1992] "Data Hiding and Security in Object-Oriented Databases", en *ICDE* [1992].

Bertino, E., Catania, B. y Ferrari, E. [2001] "A Nested Transaction Model for Multilevel SecureDatabase Management Systems", **ACM Transactions on Information and System Security (TISSEC)**, 4:4, noviembre de 2001, págs. 321–370.

Bertino, E. y Ferrari, E. [1998] "Data Security", *Proc. Twenty-Second Annual International Conference on Computer Software and Applications*, agosto de 1998, págs. 228–237.

Bertino, E. y Guerrini, G. [1998], "Extending the ODMG Object Model with Composite Objects", *OOPSLA*, Vancouver, Canadá, 1998, págs. 259–270.

Bertino, E. y Kim, W. [1989] "Indexing Techniques for Queries on Nested Objects", **TKDE**, 1:2, junio de 1989.

Bertino, E., Negri, M., Pelagatti, G. y Sbattella, L. [1992] "Object-Oriented Query Languages: The Notion and the Issues", **TKDE**, 4:3, junio de 1992.

Bertino, E., Pagani, E. y Rossi, G. [1992] "Fault Tolerance and Recovery in Mobile Computing Systems", en Kumar and Han [1992].

Bertino, F., Rabitti, F. y Gibbs, S. [1988] "Query Processing in a Multimedia Document System", **TOIS**, 6:1, 1988.

Bhargava, B., ed. [1987] **Concurrency and Reliability in Distributed Systems**, Van Nostrand-Reinhold, 1987.

Bhargava, B. y Helal, A. [1993] "Efficient Reliability Mechanisms in Distributed Database Systems", *CIKM,* noviembre de 1993.

Bhargava, B. y Reidl, J. [1988] "A Model for Adaptable Systems for Transaction Processing", en *ICDE* [1988].

Biliris, A. [1992] "The Performance of Three Database Storage Structures for Managing Large Objects", en *SIGMOD* [1992].

Biller, H. [1979] "On the Equivalence of Data Base Schemas—A Semantic Approach to Data Translation", **Information Systems**, 4:1, 1979.

Bischoff, J. y T. Alexander, eds., **Data Warehouse: Practical Advice from the Experts**, Prentice-Hall, 1997.

Biskup, J., Dayal, U. y Bernstein, P. [1979] "Synthesizing Independent Database Schemas", en *SIGMOD* [1979].

Bjork, A. [1973] "Recovery Scenario for a DB/DC System", *Proc. ACM National Conference,* 1973.

Bjorner, D. y Lovengren, H. [1982] "Formalization of Database Systems and a Formal Definition of IMS", en *VLDB* [1982].

Blaha, M., Premerlani, W. [1998] **Object-Oriented Modeling and Design for Database Applications**, Prentice-Hall, 1998.

Blakeley, J., Coburn, N. y Larson, P. [1989] "Updated Derived Relations: Detecting Irrelevant and Autonomously Computable Updates", **TODS**, 14:3, septiembre de 1989.

Blakeley, J. y Martin, N. [1990] "Join Index, Materialized View and Hybrid-Hash Join: A Performance Analysis", en *ICDE* [1990].

Blasgen, M., y Eswaran, K. [1976] "On the Evaluation of Queries in a Relational Database System", **IBM Systems Journal**, 16:1, enero de 1976.

Blasgen, M. y otros [1981] "System R: An Architectural Overview", **IBM Systems Journal**, 20:1, enero de 1981.

Bleier, R. y Vorhaus, A. [1968] "File Organization in the SDC TDMS", *Proc. IFIP Congress.*

Bocca, J. [1986] "EDUCE—A Marriage of Convenience: Prolog and a Relational DBMS", *Proc. Third International Conference on Logic Programming,* Springer-Verlag, 1986.

Bocca, J. [1986a] "On the Evaluation Strategy of EDUCE", in *SIGMOD* [1986].

Bodorick, P., Riordon, J. y Pyra, J. [1992] "Deciding on Correct Distributed Query Processing", **TKDE**, 4:3, junio de 1992.

Booch, G., Rumbaugh, J. y Jacobson, I., **Unified Modeling Language User Guide**, Addison-Wesley, 1999.

Borgida, A., Brachman, R., McGuinness, D. y Resnick, L. [1989] "CLASSIC: A Structural Data Model for Objects", en *SIGMOD* [1989].

Borges, K., Laender, A. y Davis, C. [1999], "Spatial data integrity constraints in object oriented geographic data modeling", *Proc. 7th ACM International Symposium on Advances in Geographic Information Systems*, 1999.

Borkin, S. [1978] "Data Model Equivalence", en *VLDB* [1978].

Bossomaier, T. y Green, D.[2002] **Online GIS and Metadata**, Taylor and Francis, 2002.

Boutselakis, H. y otros [2003] "E-MSD: the European Bioinformatics Institute Macromolecular Structure Database", **Nucleic Acids Research**, 31:1, págs. 458–462, enero de 2003.

Bouzeghoub, M. y Metais, E. [1991] "Semantic Modelling of Object-Oriented Databases", en *VLDB* [1991].

Boyce, R., Chamberlin, D., King, W. y Hammer, M. [1975] "Specifying Queries as Relational Expressions", **CACM**, 18:11, noviembre de 1975.

Bracchi, G., Paolini, P. y Pelagatti, G. [1976] "Binary Logical Associations in Data Modelling", en Nijssen [1976].

Brachman, R. y Levesque, H. [1984] "What Makes a Knowledge Base Knowledgeable? A View of Databases from the Knowledge Level", en *EDS* [1984].

Brandon, M. y otros [2005] MITOMAP: A human mitochondrial genome database—2004 Update, Nucleic Acid Research,Vol. 34 (1), enero de 2005.

Bratbergsengen, K. [1984] "Hashing Methods and Relational Algebra Operators", en *VLDB* [1984].

Bray, O. [1988] **Computer Integrated Manufacturing —The Data Management Strategy**, Digital Press, 1988.

Breitbart, Y., Komondoor, R., Rastogi, R., Seshadri, S. y Silberschatz, A. [1999] "Update Propagation Protocols for Replicated Databases", en *SIGMOD* [1999], págs. 97–108

Breitbart, Y., Silberschatz, A. y Thompson, G. [1990] "Reliable Transaction Management in a Multidatabase System", en *SIGMOD* [1990].

Brodeur, J., Bédard, Y. y Proulx, M. [2000] "Modelling Geospatial Application Databases Using UML-Based Repositories Aligned with International Standards in Geomatics", *Proc. 8th ACM International Symposium on Advances in Geographic Information Systems*. Washington, DC, ACM Press, 2000, págs. 39–46.

Brodie, M. y Mylopoulos, J., eds. [1985] **On Knowledge Base Management Systems**, Springer-Verlag, 1985.

Brodie, M., Mylopoulos, J. y Schmidt, J., eds. [1984] **On Conceptual Modeling**, Springer-Verlag, 1984.

Brosey, M. y Shneiderman, B. [1978] "Two Experimental Comparisons of Relational and Hierarchical Database Models", **International Journal of Man-Machine Studies**, 1978.

Bry, F. [1990] "Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled", **DKE**, 5, 1990, págs. 289–312.

Bukhres, O. [1992] "Performance Comparison of Distributed Deadlock Detection Algorithms", en *ICDE* [1992].

Buneman, P. y Frankel, R. [1979] "FQL: A Functional Query Language", en *SIGMOD* [1979].

Burkhard, W. [1976] "Hashing and Trie Algorithms for Partial Match Retrieval", **TODS**, 1:2, junio de 1976, págs. 175–187.

Burkhard,W. [1979] "Partial-match Hash Coding: Benefits of Redunancy", **TODS**, 4:2, junio de 1979, págs. 228–239.

Bush, V. [1945] "As We May Think", *Atlantic Monthly,* 176:1, enero de 1945. Reeditado en

Kochen, M., ed., **The Growth of Knowledge**, Wiley, 1967.

Byte [1995] Special Issue on Mobile Computing, junio de 1995.

**CACM** [1995] Special issue of the **Communications of the ACM**, on Digital Libraries, 38:5, mayo de 1995.

**CACM** [1998] Special issue of the **Communications of the ACM** on Digital Libraries: Global Scope and Unlimited Access, 41:4, abril de 1998.

Cammarata, S., Ramachandra, P. y Shane, D. [1989] "Extending a Relational Database with Deferred Referential Integrity Checking and Intelligent Joins", en *SIGMOD* [1989].

Campbell, D., Embley, D. y Czejdo, B. [1985] "A Relationally Complete Query Language for the Entity-Relationship Model", en la Conferencia ER [1985].

Cárdenas, A. [1985] **Data Base Management Systems**, 2ª ed., Allyn and Bacon, 1985.

Carey, M. y otros [1986] "The Architecture of the EXODUS Extensible DBMS", en Dittrich and Dayal [1986].

Carey, M., DeWitt, D., Richardson, J. y Shekita, E. [1986a] "Object and File Management in the EXODUS Extensible Database System", en *VLDB* [1986].

Carey, M., DeWitt, D. y Vandenberg, S. [1988] "A Data Model and Query Language for Exodus", en *SIGMOD* [1988].

Carey, M., Franklin, M., Livny, M. y Shekita, E. [1991] "Data Caching Tradeoffs in Client-Server DBMS Architectures", en *SIGMOD* [1991].

Carlis, J. [1986] "HAS, a Relational Algebra Operator or Divide Is Not Enough to Conquer", en *ICDE* [1986].

Carlis, J. y March, S. [1984] "A Descriptive Model of Physical Database Design Problems and Solutions", en *ICDE* [1984].

Carroll, J.M., [1995] **Scenario Based Design: Envisioning Work and Technology in System Development**, Wiley, 1995.

Casanova, M., Fagin, R. y Papadimitriou, C. [1981] "Inclusion Dependencies and Their Interaction with Functional Dependencies", en *PODS* [1981].

Casanova, M., Furtado, A. y Tuchermann, L. [1991] "A Software Tool for Modular Database Design", **TODS**, 16:2, junio de 1991.

Casanova, M., Tuchermann, L., Furtado, A. y Braga, A. [1989] "Optimization of Relational Schemas Containing Inclusion Dependencies", en *VLDB* [1989].

Casanova, M. y Vidal, V. [1982] "Toward a Sound View Integration Method", en *PODS* [1982].

Castano, S., De Antonellio,V., Fugini, M.G. y Pernici, B. [1998] "Conceptual Schema Analysis: Techniques and Applications", **TODS**, 23:3, septiembre de 1998, págs. 286–332.

Castano, S., Fugini, M., Martella G. y Samarati, P. [1995] **Database Security**, ACM Press and Addison-Wesley, 1995.

Catarci, T., Costabile, M.F., Santucci, G. y Tarantino, L., eds. [1998] *Proc. Fourth International Workshop on Advanced Visual Interfaces,* ACM Press, 1998.

Catarci, T., Costabile, M.F., Levialdi, S. y Batini, C. [1997] "Visual Query Systems for Databases: A Survey", **Journal of Visual Languages and Computing**, 8:2, junio de 1997, págs. 215–260.

Cattell, R., ed. [1993] **The Object Database Standard: ODMG-93**, **Release 1.2**, Morgan Kaufmann, 1993.

Cattell, R., ed. [1997] **The Object Database Standard: ODMG**, **Release 2.0**, Morgan Kaufmann, 1997.

Cattell, R. y otros (2000), The Object Data Standard: ODMG 3.0, Morgan Kaufmann, 2000.

Cattell, R. y Skeen, J. [1992] "Object Operations Benchmark", **TODS**, 17:1, marzo de 1992.

Ceri, S. y Fraternali, P. [1997] **Designing Database Applications with Objects and Rules: The IDEA Methodology**, Addison-Wesley, 1997.

Ceri, S., Gottlob, G., Tanca, L. [1990], **Logic Programming and Databases**, Springer-Verlag, 1990.

Ceri, S., Navathe, S. y Wiederhold, G. [1983] "Distribution Design of Logical Database Schemas", **TSE**, 9:4, julio de 1983.

Ceri, S., Negri, M. y Pelagatti, G. [1982] "Horizontal Data Partitioning in Database Design", en *SIGMOD* [1982].

Ceri, S. y Owicki, S. [1983] "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases", *Proc. Sixth Berkeley Workshop on Distributed Data Management and Computer Networks,* febrero de 1983.

Ceri, S. y Pelagatti, G. [1984] "Correctness of Query Execution Strategies in Distributed Databases", **TODS**, 8:4, diciembre de 1984.

Ceri, S. y Pelagatti, G. [1984a] **Distributed Databases: Principles and Systems**, McGraw-Hill, 1984.

Ceri, S. y Tanca, L. [1987] "Optimization of Systems of Algebraic Equations for Evaluating Datalog Queries", en *VLDB* [1987].

Cesarini, F. y Soda, G. [1991] "A Dynamic Hash Method with Signature", **TODS**, 16:2, junio de 1991.

Chakravarthy, S. [1990] "Active Database Management Systems: Requirements, State-of-the-Art, and an Evaluation", en la Conferencia ER [1990].

Chakravarthy, S. [1991] "Divide and Conquer: A Basis for Augmenting a Conventional Query Optimizer with Multiple Query Processing Capabilities", en *ICDE* [1991].

Chakravarthy, S., Anwar, E., Maugis, L. y Mishra, D. [1994] Design of Sentinel: An Object-oriented DBMS with Event-based Rules, **Information and Software Technology**, 36:9, 1994.

Chakravarthy, S. y otros [1989] "HiPAC: A Research Project in Active, Time Constrained Database Management", Final Technical Report, XAIT-89-02, Xerox Advanced Information Technology, agosto de 1989.

Chakravarthy, S., Karlapalem, K., Navathe, S. y Tanaka, A. [1993] "Database Supported Cooperative Problem Solving", en **International Journal of Intelligent Co-operative Information Systems**, 2:3, septiembre de 1993.

Chakravarthy, U., Grant, J. y Minker, J. [1990] "Logic-Based Approach to Semantic Query Optimization", **TODS**, 15:2, junio de 1990.

Chalmers, M. y Chitson, P. [1992] "Bead: Explorations in Information Visualization", *Proc. ACM SIGIR International Conference,* junio de 1992.

Chamberlin, D. y Boyce, R. [1974] "SEQUEL: A Structured English Query Language", en *SIGMOD* [1984].

Chamberlin, D. y otros [1976] "SEQUEL 2: A Unified Approach to Data Definition, Manipulation y Control", **IBM Journal of Research and Development**, 20:6, noviembre de 1976.

Chamberlin, D. y otros [1981] "A History and Evaluation of System R", **CACM**, 24:10, octubre de 1981.

Chan, C., Ooi, B. y Lu, H. [1992] "Extensible Buffer Management of Indexes", en *VLDB* [1992].

Chandy, K., Browne, J., Dissley, C. y Uhrig, W. [1975] "Analytical Models for Rollback and Recovery Strategies in Database Systems", **TSE**, 1:1, marzo de 1975.

Chang, C. [1981] "On the Evaluation of Queries Containing Derived Relations in a Relational Database", en Gallaire y otros [1981].

Chang, C. y Walker, A. [1984] "PROSQL: A Prolog Programming Interface with SQL/DS", en *EDS* [1984].

Chang, E. y Katz, R. [1989] "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in Object-Oriented Databases", en *SIGMOD* [1989].

Chang, N. y Fu, K. [1981] "Picture Query Languages for Pictorial Databases", **IEEE Computer**, 14:11, noviembre de 1981.

Chang, P. y Myre, W. [1988] "OS/2 EE Database Manager: Overview and Technical Highlights", **IBM Systems Journal**, 27:2, 1988.

Chang, S., Lin, B. y Walser, R. [1979] "Generalized Zooming Techniques for Pictorial Database Systems", *NCC*, AFIPS, 48, 1979.

Chatzoglu, P.D. y McCaulay, L.A. [1997] "Requirements Capture and Analysis: A Survey of Current Practice", **Requirements Engineering**, 1997, págs. 75–88.

Chaudhri, A., Rashid, A. y Zicari, R. (eds.) [2003], **XML Data Management: Native XML and XML-Enabled Database Systems**, Addison-Wesley, 2003.

Chaudhuri, S. y Dayal, U. [1997] "An Overview of Data Warehousing and OLAP Technology", *SIGMOD* **Record**, Vol. 26, Núm. 1, marzo de 1997.

Chen, M., Han, J. y Yu, P.S., [1996] "Data Mining: An Overview from a Database Perspective", **TKDE**, 8:6, diciembre de 1996.

Chen, M. y Yu, P. [1991] "Determining Beneficial Semijoins for a Join Sequence in Distributed Query Processing", en *ICDE* [1991].

Chen, P. [1976] "The Entity Relationship Mode—Toward a Unified View of Data", **TODS**, 1:1, marzo de 1976.

Chen, P., Lee E., Gibson G., Katz, R. y Patterson, D. [1994] RAID High Performance, Reliable Secondary Storage, **ACMComputing Surveys**, 26:2, 1994.

Chen, P. y Patterson, D. [1990]. "Maximizing performance in a striped disk array", en *Proceedings of Symposium on Computer Architecture, IEEE,* Nueva York, 1990.

Chen, Q. y Kambayashi, Y. [1991] "Nested Relation Based Database Knowledge Represen-tation", en *SIGMOD* [1991].

Cheng, J. [1991] "Effective Clustering of Complex Objects in Object-Oriented Databases", en *SIGMOD* [1991].

Cheung, D., Han, J., Ng, V., Fu, A.W. y Fu, A.Y., "A Fast and Distributed Algorithm for Mining Association Rules", en *Proceedings of International Conference on Parallel and Distributed Information Systems,* PDIS [1996].

Childs, D. [1968] "Feasibility of a Set Theoretical Data Structure—A General Structure Based on a Reconstituted Definition of Relation", *Proc. IFIP Congress,* 1968.

Chimenti, D. y otros [1987] "An Overview of the LDL System", MCC Technical Report #ACA-ST-370-87, Austin, TX, noviembre de 1987.

Chimenti, D. y otros [1990] "The LDL System Prototype", **TKDE**, 2:1, marzo de 1990.

Chin, F. [1978] "Security in Statistical Databases for Queries with Small Counts", **TODS**, 3:1, marzo de 1978.

Chin, F. y Ozsoyoglu, G. [1981] "Statistical Database Design", **TODS**, 6:1, marzo de 1981.

Chintalapati, R., Kumar, V. y Datta, A. [1997] "An Adaptive Location Management Algorithm for Mobile Computing", *Proceedings of 22nd Annual Conference on Local Computer Networks (LCN '97),* Minneapolis, 1997.

Chou, H. y Kim,W. [1986] "A Unifying Framework for Version Control in a CAD Environment", en *VLDB* [1986].

Christodoulakis, S. y otros [1984] "Development of a Multimedia Information System for an Office Environment", en *VLDB* [1984].

Christodoulakis, S. y Faloutsos, C. [1986] "Design and Performance Considerations for an Optical Disk-Based Multimedia Object Server", **IEEE Computer**, 19:12, diciembre de 1986.

Chrysanthis, P. [1993] "Transaction Processing in a Mobile Computing Environment", *Proc. IEEE Workshop on Advances in Parallel and Distributed Systems*, octubre de 1993, págs. 77–82.

Chu,W. y Hurley, P. [1982] "Optimal Query Processing for Distributed Database Systems", **IEEE Transactions on Computers**, 31:9, septiembre de 1982.

Ciborra, C., Migliarese, P. y Romano, P. [1984] "A Methodological Inquiry of Organizational Noise in Socio Technical Systems", **Human Relations**, 37:8, 1984.

Claybrook, B. [1983] **File Management Techniques**, Wiley, 1983.

Claybrook, B. [1992] **OLTP: OnLine Transaction Processing Systems**, Wiley, 1992.

Clifford, J. y Tansel, A. [1985] "On an Algebra for Historical Relational Databases: Two Views", en *SIGMOD* [1985].

Clocksin,W.F. y Mellish, C.S. [1984] **Programming in Prolog**, 2ª ed., Springer-Verlag, 1984.

Cockcroft, S. [1997] "A Taxonomy of Spatial Data Integrity Constraints", *GeoInformatica*, 1997, págs. 327–343.

CODASYL [1978] Data Description Language Journal of Development, Canadian Government Publishing Centre, 1978.

Codd, E. [1970] "A Relational Model for Large Shared Data Banks", **CACM**, 13:6, junio de 1970.

Codd, E. [1971] "A Data Base Sublanguage Founded on the Relational Calculus", *Proc. ACM SIGFIDET Workshop on Data Description, Access y Control,* noviembre de 1971.

Codd, E. [1972] "Relational Completeness of Data Base Sublanguages", en Rustin [1972].

Codd, E. [1972a] "Further Normalization of the Data Base Relational Model", en Rustin [1972].

Codd, E. [1974] "Recent Investigations in Relational Database Systems", *Proc. IFIP Congress,* 1974.

Codd, E. [1978] "How About Recently? (English Dialog with Relational Data Bases Using Rendezvous Version 1)", en Shneiderman [1978].

Codd, E. [1979] "Extending the Database Relational Model to Capture More Meaning", **TODS**, 4:4, diciembre de 1979.

Codd, E. [1982] "Relational Database: A Practical Foundation for Productivity", **CACM**, 25:2, diciembre de 1982.

Codd, E. [1985] "Is Your DBMS Really Relational?" and "Does Your DBMS Run By the Rules?", **Computer World**, 14 y 21 de octubre, 1985.

Codd, E. [1986] "An Evaluation Scheme for Database Management Systems That Are Claimed to Be Relational", en *ICDE* [1986].

Codd, E. [1990] **Relational Model for Data Management-Version 2**, Addison-Wesley, 1990.

Codd, E.F., Codd, S.B. y Salley, C.T. [1993] "Providing OLAP (On-Line Analytical Processing) to User Analyst: An IT Mandate", un documento disponible en *http://www.arborsoft*.com/OLAP.html, 1993.

Comer, D. [1979] "The Ubiquitous B-tree", **ACM Computing Surveys**, 11:2, junio de 1979.

Comer, D. [1997] **Computer Networks and Internets**, Prentice-Hall, 1997.

Corcho, C., Fernández-López, M. y Gómez-Pérez, A. [2003] "Methodologies, Tools and Languages for Building Ontologies.Where Is Their Meeting Point?", **DKE**, 46:1, julio de 2003.

Cornelio, A. y Navathe, S. [1993] "Applying Active Database Models for Simulation", en *Proceedings of 1993 Winter Simulation Conference,* IEEE, Los Ángeles, diciembre de 1993.

Corson, S. y Macker, J. [1999] Mobile Ad-Hoc Networking: Routing Protocol Performance Issues and Performance Considerations, IETF Request for Comments Núm. 2601, enero de 1999, disponible en: *http://www.ietf.org/rfc/rfc2501.txt.*

Cosmadakis, S., Kanellakis, P. C. y Vardi, M. [1990] "Polynomial-time Implication Problems for Unary Inclusion Dependencies", **JACM**, 37:1, 1990, págs. 15–46.

Cruz, I. [1992] "Doodle: A Visual Language for Object-Oriented Databases", en *SIGMOD* [1992].

Curtice, R. [1981] "Data Dictionaries: An Assessment of Current Practice and Problems", en *VLDB* [1981].

Cuticchia, A., Fasman, K., Kingsbury, D., Robbins, R. y Pearson, P. [1993] "The GDB Human Genome Database Anno 1993". **Nucleic Acids Research**, 21:13, 1993.

Czejdo, B., Elmasri, R., Rusinkiewicz, M. y Embley, D. [1987] "An Algebraic Language for Graphical Query Formulation Using an Extended Entity-Relationship Model", *Proc. ACM Computer Science Conference,* 1987.

Dahl, R. y Bubenko, J. [1982] "IDBD: An Interactive Design Tool for CODASYL DBTG Type Databases", en *VLDB* [1982].

Dahl, V. [1984] "Logic Programming for Constructive Database Systems", en *EDS* [1984].

Das, S. [1992] **Deductive Databases and Logic Programming**, Addison-Wesley, 1992.

Date, C. [1983] **An Introduction to Database Systems**, Vol. 2, Addison-Wesley, 1983.

Date, C. [1983a] "The Outer Join", *Proc. Second International Conference on Databases (ICOD-2),* 1983.

Date, C. [1984] "A Critique of the SQL Database Language", **ACM SIGMOD Record**, 14:3, noviembre de 1984.

Date, C. [1995] **An Introduction Database Systems**, 8ª ed., Addison-Wesley, 2004.

Date, C. [2001] **The Database Relational Model: A Retrospective Review and Analysis: A Historical Account and Assessment of E.F. Codd's Contribution to the Field of Database Technology**, Addison-Wesley, 2001.

Date, C.J. y Darwen, H. [1993] **A Guide to the SQL Standard**, 3ª ed., Addison-Wesley.

Date, C. y White, C. [1989] **A Guide to DB2**, 3ª ed., Addison-Wesley, 1989.

Date, C. y White, C. [1988a] **A Guide to SQL/DS**, Addison-Wesley, 1988.

Davies, C. [1973] "Recovery Semantics for a DB/DC System", *Proc. ACM National Conference,* 1973.

Dayal, U. y Bernstein, P. [1978] "On the Updatability of Relational Views", en *VLDB* [1978].

Dayal, U., Hsu, M. y Ladin, R. [1991] "A Transaction Model for Long-Running Activities", en *VLDB* [1991].

Dayal, U. y otros [1987] "PROBE Final Report", Technical Report CCA-87-02, Computer Corporation of America, diciembre de 1987.

DBTG [1971] **Report of the codasyl Data Base Task Group**, ACM, abril de 1971.

Delcambre, L., Lim, B. y Urban, S. [1991] "Object-Centered Constraints", en *ICDE* [1991].

DeMarco, T. [1979] **Structured Analysis and System Specification**, Prentice-Hall, 1979.

DeMers,M. [2002] **Fundamentals of GIS**, John Wiley, 2002.

DeMichiel, L. [1989] "Performing Operations Over Mismatched Domains", en *ICDE* [1989].

Denning, D. [1980] "Secure Statistical Databases with Random Sample Queries", **TODS**, 5:3, septiembre de 1980.

Denning, D.E. y Denning, P.J. [1979] "Data Security" **ACM Computing Surveys**, 11:3, septiembre de 1979, págs. 227–249.

Deshpande, A. [1989] "*An Implementation for Nested Relational Databases*", Technical Report, Ph.D. Dissertation, Universidad de Indiana, 1989.

Devor, C. y Weeldreyer, J. [1980] "DDTS: A Testbed for Distributed Database Research", *Proc. ACM Pacific Conference,* 1980.

Dewire, D. [1993] **Client Server Computing**, McGraw-Hill, 1993.

DeWitt, D. y otros [1984] "Implementation Techniques for Main Memory Databases", en *SIGMOD* [1984].

DeWitt, D. y otros [1990] "The Gamma Database Machine Project", **TKDE**, 2:1, marzo de 1990.

DeWitt, D., Futtersack, P., Maier, D. y Velez, F. [1990] "A Study of Three Alternative Workstation Server Architectures for Object-Oriented Database Systems", en *VLDB* [1990].

Dhawan, C. [1997] **Mobile Computing**, McGraw-Hill, 1997.

Dietrich, S., Friesen, O. y Calliss,W. [1998] "*On Deductive and Object Oriented Databases: The VALIDITY Experience*", Technical Report, Arizona State University, 1999.

Diffie,W. y Hellman, M. [1979] "Privacy and Authentication", **Proceedings of the IEEE**, 67:3, marzo de 1979.

Diffie,W. y Hellman, M. [1979] "Privacy and Authentication", **Proceedings of the IEEE**, 6:3, marzo de 1979, págs. 397–429.

Dimitrova, N. [1999] "Multimedia Content Analysis and Indexing for Filtering and Retrieval Applications", **Informing Science**, Special Issue on Multimedia Informing Technologies, Parte 1, 2:4, 1999.

Dipert, B. y Levy, M. [1993] **Designing with Flash Memory**, Annabooks 1993.

Dittrich,K. [1986] "Object-Oriented Database Systems: The Notion and the Issues", en Dittrich and Dayal [1986].

Dittrich, K. y Dayal, U., eds. [1986] *Proc. International Workshop on Object-Oriented Database Systems,* IEEE CS, Pacific Grove, CA, septiembre de 1986.

Dittrich, K., Kotz, A. y Mulle, J. [1986] "An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases", en **ACM SIGMOD Record**, 15:3, 1986.

**DKE** [1997] Special Issue on Natural Language Processing, **DKE**, 22:1, 1997.

Dodd, G. [1969] "APL—A Language for Associative Data Handling in PL/I", *Proc. Fall Joint Computer Conference,* AFIPS, 29, 1969.

Dodd, G. [1969] "Elements of Data Management Systems", **ACM Computing Surveys**, 1:2, junio de 1969.

Dogac, A., Ozsu, M.T., Biliris, A. y Sellis, T., eds. [1994] **Advances in Object-oriented Databases Systems**, NATO ASI Series. Series F: Computer and Systems Sciences, Vol. 130, Springer-Verlag, 1994.

Dogac, A., [1998] Special Section on Electronic Commerce, **ACMSIGMOD Record**, 27:4, diciembre de 1998.

Dos Santos, C., Neuhold, E. y Furtado, A. [1979] "A Data Type Approach to the Entity-Relationship Model", en la Conferencia ER [1979].

Du, D. y Tong, S. [1991] "Multilevel Extendible Hashing: A File Structure for Very Large Databases", **TKDE**, 3:3, septiembre de 1991.

Du, H. y Ghanta, S. [1987] "A Framework for Efficient IC/VLSI CAD Databases", en *ICDE* [1987].

Dumas, P. y otros [1982] "MOBILE-Burotique: Prospects for the Future", en Naffah [1982].

Dumpala, S. y Arora, S. [1983] "Schema Translation Using the Entity-Relationship Approach", en la Conferencia ER [1983].

Dunham, M. y Helal, A. [1995] "Mobile Computing and Databases: Anything New?", **ACM SIGMOD Record**, 24:4, diciembre de 1995.

Dwyer, S. y otros [1982] "A Diagnostic Digital Imaging System", *Proc. IEEE CS Conference on Pattern Recognition and Image Processing,* junio de 1982.

Eastman, C. [1987] "Database Facilities for Engineering Design", **Proceedings of the IEEE**, 69:10, octubre de 1981.

*EDS* [1984] **Expert Database Systems**, Kerschberg, L., ed. (*Proc. First International Workshop on Expert Database Systems,* Kiawah Island, SC, octubre de 1984), Benjamin/Cummings, 1986.

*EDS* [1986] **Expert Database Systems**, Kerschberg, L., ed. (*Proc. First International Conference on Expert Database Systems,* Charleston, SC, abril de 1986), Benjamin/Cummings, 1987.

*EDS* [1988] **Expert Database Systems**, Kerschberg, L., ed. (*Proc. Second International Conference on Expert Database Systems,* Tysons Corner, VA, abril de 1988), Benjamin/Cummings.

Eick, C. [1991] "A Methodology for the Design and Transformation of Conceptual Schemas", en *VLDB* [1991].

El Abbadi, A. y Toueg, S. [1988] "The Group Paradigm for Concurrency Control", en *SIGMOD* [1988].

El Abbadi, A. y Toueg, S. [1989] "Maintaining Availability in Partitioned Replicated Databases", **TODS**, 14:2, junio de 1989.

Ellis, C. y Nutt, G. [1980] "Office Information Systems and Computer Science", **ACM Computing Surveys**, 12:1, marzo de 1980.

Elmagarmid A.K., ed. [1992] **Database Transaction Models for Advanced Applications**, Morgan Kaufmann, 1992.

Elmagarmid, A., Leu, Y., Litwin,W. y Rusinkiewicz, M. [1990] "A Multidatabase Transaction Model for Interbase", en *VLDB* [1990].

Elmasri, R., James, S. y Kouramajian, V. [1993] "Automatic Class and Method Generation for Object-Oriented Databases", *Proc. Third International Conference on Deductive and Object-Oriented Databases (DOOD-93),* Phoenix, AZ, diciembre de 1993.

Elmasri, R., Kouramajian, V. y Fernando, S. [1993] "Temporal Database Modeling: An Object-Oriented Approach", *CIKM,* noviembre de 1993.

Elmasri, R. y Larson, J. [1985] "A Graphical Query Facility for ER Databases", en la Conferencia ER [1985].

Elmasri, R., Larson, J. y Navathe, S. [1986] "Schema Integration Algorithms for Federated Databases and Logical Database Design", Honeywell CSDD, Technical Report CSC-86-9: 8212, enero de 1986.

Elmasri, R., Srinivas, P. y Thomas, G. [1987] "Fragmentation and Query Decomposition in the ECRModel", en *ICDE* [1987].

Elmasri, R., Weeldreyer, J. y Hevner, A. [1985] "The Category Concept: An Extension to the Entity-Relationship Model", **DKE**, 1:1, mayo de 1985.

Elmasri, R. y Wiederhold, G. [1979] "Data Model Integration Using the Structural Model", en *SIGMOD* [1979].

Elmasri, R. y Wiederhold, G. [1980] "Structural Properties of Relationships and Their Representation", *NCC,* AFIPS, 49, 1980.

Elmasri, R. y Wiederhold, G. [1981] "GORDAS: A Formal, High-Level Query Language for the Entity-Relationship Model", en la Conferencia ER [1981].

Elmasri, R. y Wuu, G. [1990] "A Temporal Model and Query Language for ER Databases", en *ICDE* [1990].

Elmasri, R. y Wuu, G. [1990a] "The Time Index: An Access Structure for Temporal Data", en *VLDB* [1990].

Engelbart, D. y English, W. [1968] "A Research Center for Augmenting Human Intellect", *Proc. Fall Joint Computer Conference,* AFIPS, diciembre de 1968.

Epstein, R., Stonebraker, M. y Wong, E. [1978] "Distributed Query Processing in a Relational Database System", en *SIGMOD* [1978].

*ER, Conferencia* [1979] **Entity-Relationship Approach to Systems Analysis and Design**, Chen, P., ed. (*Proc. First International Conference on Entity-Relationship Approach,* Los Ángeles, diciembre de 1979), North-Holland, 1980.

*ER, Conferencia* [1981] **Entity-Relationship Approach to Information Modeling and Analysis**, Chen, P., eds. (*Proc. Second International Conference on Entity-Relationship Approach,* Washington, octubre de 1981), Elsevier Science, 1981.

*ER, Conferencia* [1983] **Entity-Relationship Approach to Software Engineering**, Davis, C., Jajodia, S., Ng, P. y Yeh, R., eds. (*Proc. Third International Conference on Entity-Relationship Approach,* Anaheim, CA, octubre de 1983), North-Holland, 1983.

*ER, Conferencia* [1985] *Proc. Fourth International Conference on Entity-Relationship Approach,* Liu, J., ed., Chicago, octubre de 1985, IEEE CS.

*ER, Conferencia* [1986] *Proc. Fifth International Conference on Entity-Relationship Approach,* Spaccapietra, S., ed., Dijon, Francia, noviembre de 1986, Express-Tirages.

*ER, Conferencia* [1987] *Proc. Sixth International Conference on Entity-Relationship Approach,* March, S., ed., Nueva York, noviembre de 1987.

*ER, Conferencia* [1988] *Proc. Seventh International Conference on Entity-Relationship Approach,* Batini, C., ed., Roma, noviembre de 1988.

*ER, Conferencia* [1989] *Proc. Eighth International Conference on Entity-Relationship Approach,* Lochovsky, F., ed., Toronto, octubre de 1989.

*ER, Conferencia* [1990] *Proc. Ninth International Conference on Entity-Relationship Approach,* Kangassalo, H., ed., Lausana, Suiza, septiembre de 1990.

*ER, Conferencia* [1991] *Proc. Tenth International Conference on Entity-Relationship Approach,* Teorey, T., ed., San Mateo, CA, octubre de 1991.

*ER, Conferencia* [1992] *Proc. Eleventh International Conference on Entity-Relationship Approach,* Pernul, G., y Tjoa, A., eds., Karlsruhe, Alemania, octubre de 1992.

*ER, Conferencia* [1993] *Proc. Twelfth International Conference on Entity-Relationship Approach,* Elmasri, R., y Kouramajian, V., eds., Arlington, TX, diciembre de 1993.

*ER, Conferencia* [1994] *Proc. Thirteenth International Conference on Entity-Relationship Approach,* Loucopoulos, P., y Theodoulidis, B., eds., Manchester, Inglaterra, diciembre de 1994.

*ER, Conferencia* [1995] *Proc. Fourteenth International Conference on ER-OO Modeling,* Papazouglou, M., y Tari, Z., eds., Brisbane, Australia, diciembre de 1995.

*ER, Conferencia* [1996] *Proc. Fifteenth International Conference on Conceptual Modeling,* Thalheim, B., ed., Cottbus, Alemania, octubre de 1996.

*ER, Conferencia* [1997] *Proc. Sixteenth International Conference on Conceptual Modeling,* Embley, D., ed., Los Ángeles, octubre de 1997.

*ER, Conferencia* [1998] *Proc. Seventeenth International Conference on Conceptual Modeling,* Ling, T-K., ed., Singapur, noviembre de 1998.

*ER, Conferencia* [1999] *Proc. Eighteenth Conference on Conceptual Modeling*, Akoka, J., Bouzeghoub, M., Comyn-Wattiau, I., Métais, E., (eds.): París, Francia, LNCS 1728, Springer, 1999.

*ER, Conferencia* [2000] *Proc. Nineteenth Conference on Conceptual Modeling*, Laender, A., Liddle, S., Storey, V., (eds.), Salt Lake City, LNCS 1920, Springer, 2000.

*ER, Conferencia* [2001] *Proc. Twentieth Conference on Conceptual Modeling*, Kunii, H., Jajodia, S., Solveberg, A., (eds.), Yokohama, Japón, LNCS 2224, Springer, 2001.

*ER, Conferencia* [2002] *Proc. 21st Conference on Conceptual Modeling*, Spaccapietra, S., March S., Kambayashi, Y., (eds.), Tampere, Finlandia, LNCS 2503, Springer, 2002.

*ER, Conferencia* [2003] *Proc. 22nd Conference on Conceptual Modeling*, Song, I-Y., Liddle, S., Ling, T-W., Scheuermann, P., (eds.), Tampere, Finlandia, LNCS 2813, Springer, 2003.

*ER, Conferencia* [2004] *Proc. 23rd Conference on Conceptual Modeling*, Atzeni, P., Chu,W., Lu, H., Zhou, S. y Ling, T-W., (eds.), Shanghai, China, LNCS 3288, Springer, 2004.

*ER, Conferencia* [2005] *Proc. 24th Conference on Conceptual Modeling*, Delacambre, L.M.L., Kop, C., Mayr, H., Mylopoulos, J. y Pastor, O., (eds.), Klagenfurt, Austria, LNCS 3716, Springer, 2005.

Eswaran, K. y Chamberlin, D. [1975] "Functional Specifications of a Subsystem for Database Integrity", en *VLDB* [1975].

Eswaran, K., Gray, J., Lorie, R. y Traiger, I. [1976] "The Notions of Consistency and Predicate Locks in a Data Base System", **CACM**, 19:11, noviembre de 1976.

Everett, G., Dissly, C. y Hardgrave, W. [1971] *RFMS User Manual,* TRM-16, Computing Center, Universidad de Texas en Austin, 1981.

Fagin, R. [1977] "Multivalued Dependencies and a New Normal Form for Relational Databases", **TODS**, 2:3, septiembre de 1977.

Fagin, R. [1979] "Normal Forms and Relational Database Operators", en *SIGMOD* [1979].

Fagin, R. [1981] "A Normal Form for Relational Databases That Is Based on Domains and Keys", **TODS**, 6:3, septiembre de 1981.

Fagin, R., Nievergelt, J., Pippenger, N. y Strong, H. [1979] "Extendible Hashing—A Fast Access Method for Dynamic Files", **TODS**, 4:3, septiembre de 1979.

Falcone, S. y Paton, N. [1997]. "Deductive Object-Oriented Database Systems: A Survey", *Proc. 3ª International Workshop Rules in Database Systems (RIDS '97),* Skovde, Suecia, junio de 1997.

Faloutsos, C. [1996] **Searching Multimedia Databases by Content**, Kluwer, 1996.

Faloutsos, G. y Jagadish, H. [1992] "On B-Tree Indices for Skewed Distributions", en *VLDB* [1992].

Faloutsos, C., Barber, R., Flickner, M., Hafner, J., Niblack, W., Perkovic, D. y Equitz,W. [1994] Efficient and Effective Querying by Image Content", en **Journal of Intelligent Information Systems**, 3:4, 1994.

Farag, W. y Teorey, T. [1993] "FunBase: A Functionbased Information Management System", *CIKM,* noviembre de 1993.

Farahmand, F., Navathe, S., Sharp, G. y Enslow, P. [2003] "Managing Vulnerabilities of Information Systems to Security Incidents", *Proc. ACM 5ª International Conference on Electronic Commerce, ICEC 2003,* Pittsburgh, PA, septiembre de 2003, págs. 348–354.

Farahmand, F., Navathe, S., Sharp, G. y Enslow, P., "A Management Perspective on Risk of Security Threats to Information Systems", **Journal of Information Technology & Management**, Vol. 6, págs. 203–225, 2005.

Fayyad, U., Piatesky-Shapiro, G., Smyth, P. y Uthurusamy, R. [1997] **Advances in Knowledge Discovery and Data Mining**, MIT Press, 1997.

Fensel, D. [2000] "The Semantic Web and Its Languages", **IEEE Intelligent Systems**, Vol. 15, Núm. 6, Nov./Dic. 2000, págs. 67–73.

Fensel, D. [2003]: **Ontologies: Silver Bullet for Knowledge Management and Electronic Commerce**, 2ª ed., Springer-Verlag, Berlín, 2003.

Fernández, E., Summers, R. y Wood, C. [1981] **Database Security and Integrity**, Addison-Wesley, 1981.

Ferrier, A. y Stangret, C. [1982] "Heterogeneity in the Distributed Database Management System SIRIUSDELTA", en *VLDB* [1982].

Fishman, D. y otros [1986] "IRIS: An Object-Oriented DBMS", **TOOIS**, 4:2, abril de 1986.

Flynn, J. y Pitts, T. [2000] **Inside ArcINFO 8**, 2ª ed., On Word press, 2000.

Folk, M.J., Zoellick, B. y Riccardi, G. [1998] **File Structures: An Object Oriented Approach with C++**, 3ª ed., Addison-Wesley, 1998.

Ford, D., Blakeley, J. y Bannon, T. [1993] "Open OODB: A Modular Object-Oriented DBMS", en *SIGMOD* [1993].

Ford, D. y Christodoulakis, S. [1991] "Optimizing Random Retrievals from CLV Format Optical Disks", en *VLDB* [1991].

Foreman, G. y Zahorjan, J. [1994] "The Challenges of Mobile Computing", **IEEE Computer**, abril de 1994.

Fowler, M. y Scott, K. [2000] **UML Distilled**, 2ª ed., Addison-Wesley, 2000.

Franaszek, P., Robinson, J. y Thomasian, A. [1992] "Concurrency Control for High Contention Environments", **TODS**, 17:2, junio de 1992.

Franklin, F. y otros [1992] "Crash Recovery in Client-Server EXODUS", en *SIGMOD* [1992].

Fraternali, P. [1999] Tools and Approaches for Data Intensive Web Applications: A Survey, *ACM Computing Surveys,* 31:3, septiembre de 1999.

Frenkel, K. [1991] "The Human Genome Project and Informatics", **CACM**, noviembre de 1991.

Friesen, O., Gauthier-Villars, G., Lefelorre, A. y Vieille, L., "Applications of Deductive Object-Oriented Databases Using DEL", en Ramakrishnan (1995).

Friis-Christensen, A., Tryfona, N. y Jensen, C.S. [2001] "Requirements and Research Issues in Geographic Data Modeling", *Proc. 9ª ACM International Symposium on Advances in Geographic Information Systems,* 2001.

Furtado, A. [1978] "Formal Aspects of the Relational Model", **Information Systems**, 3:2, 1978.

Gadia, S. [1988] "A Homogeneous Relational Model and Query Language for Temporal Databases", **TODS**, 13:4, diciembre de 1988.

Gait, J. [1988] "The Optical File Cabinet: A Random-Access File System for Write-Once Optical Disks", **IEEE Computer**, 21:6, junio de 1988.

Gallaire, H. y Minker, J., eds. [1978] **Logic and Databases**, Plenum Press, 1978.

Gallaire, H., Minker, J. y Nicolas, J. [1984] "Logic and Databases: A Deductive Approach", **ACM Computing Surveys**, 16:2, junio de 1984.

Gallaire, H., Minker, J. y Nicolas, J., eds. [1981], **Advances in Database Theory**, Vol. 1, Plenum Press, 1981.

Gamal-Eldin, M., Thomas, G. y Elmasri, R. [1988] "Integrating Relational Databases with Support for Updates", *Proc. International Symposium on Databases in Parallel and Distributed Systems,* IEEE CS, diciembre de 1988.

Gane, C. y Sarson, T. [1977] **Structured Systems Analysis: Tools and Techniques**, Improved Systems Technologies, 1977.

Gangopadhyay, A. y Adam, N. [1997]. **Database Issues in Geographic Information Systems**, Kluwer Academic Publishers, 1997.

García-Molina, H. [1982] "Elections in Distributed Computing Systems", **IEEE Transactions on Computers**, 31:1, enero de 1982.

García-Molina, H. [1983] "Using Semantic Knowledge for Transaction Processing in a Distributed Database", **TODS**, 8:2, junio de 1983.

García-Molina, H., Ullman, J., Widom, J. [2000] **Database System Implementation**, Prentice-Hall, 2000.

García-Molina, H., Ullman, J., Widom, J. [2002] **Database Systems: The Complete Book**, Prentice-Hall, 2002.

Gehani, N., Jagdish, H. y Shmueli, O. [1992] "Composite Event Specification in Active Databases: Model and Implementation", en *VLDB* [1992].

Georgakopoulos, D., Rusinkiewicz, M. y Sheth, A. [1991] "On Serializability of Multidatabase Transactions Through Forced Local Conflicts", en *ICDE* [1991].

Gerritsen, R. [1975] "A Preliminary System for the Design of DBTG Data Structures", **CACM**, 18:10, octubre de 1975.

Ghosh, S. [1984] "An Application of Statistical Databases in Manufacturing Testing", en *ICDE* [1984].

Ghosh, S. [1986] "Statistical Data Reduction for Manufacturing Testing", en *ICDE* [1986].

Gifford, D. [1979] "Weighted Voting for Replicated Data", *Proc. Seventh ACM Symposium on Operating Systems Principles,* 1979.

Gladney, H. [1989] "Data Replicas in Distributed Information Services", **TODS**, 14:1, marzo de 1989.

Gogolla, M. y Hohenstein, U. [1991] "Towards a Semantic View of an Extended Entity-

Relationship Model", **TODS**, 16:3, septiembre de 1991.

Goldberg, A. y Robson, D. [1983] **Smalltalk-80: The Language and Its Implementation**, Addison-Wesley, 1983.

Goldfine, A. y Konig, P. [1988] *A Technical Overview of the Information Resource Dictionary System (IRDS),* 2ª ed., NBS IR 88-3700, National Bureau of Standards.

Gordillo, S. y Balaguer, F. [1998] "Refining an Objectoriented GIS Design Model: Topologies and Field Data", *Proc. 6ª ACM International Symposium on Advances in Geographic Information Systems,* 1998.

Goodchild, M.F. [1992] "Geographical Information Science", **International Journal of Geographical Information Systems**, 1992, págs. 31–45.

Goodchild, M.F. [1992a] "Geographical Data Modeling", *Computers & Geosciences* 18(4): 401–408, 1992.

Gotlieb, L. [1975] "Computing Joins of Relations", en *SIGMOD* [1975].

Graefe, G. [1993] "Query Evaluation Techniques for Large Databases", **ACM Computing Surveys**, 25:2, junio de 1993.

Graefe, G. y DeWitt, D. [1987] "The EXODUS Optimizer Generator", en *SIGMOD* [1987].

Gravano, L. y García-Molina, H. [1997] "Merging Ranks from Heterogeneous Sources", en *VLDB* [1997].

Gray, J. [1978] "Notes on Data Base Operating Systems", en Bayer, Graham y Seegmuller [1978].

Gray, J. [1981] "The Transaction Concept: Virtues and Limitations", en *VLDB* [1981].

Gray, J., Helland, P., O'Neil, P. y Shasha, D. [1993] "The Dangers of Replication and a Solution", *SIGMOD* [1993]

Gray, J., Lorie, R. y Putzolu, G. [1975] "Granularity of Locks and Degrees of Consistency in a Shared Data Base", en Nijssen [1975].

Gray, J., McJones, P. y Blasgen, M. [1981] "The Recovery Manager of the System R Database Manager", **ACM Computing Surveys**, 13:2, junio de 1981.

Gray, J. y Reuter, A. [1993] **Transaction Processing: Concepts and Techniques**, Morgan Kaufmann, 1993.

Griffiths, P. y Wade, B. [1976] "An Authorization Mechanism for a Relational Database System", **TODS**, 1:3, septiembre de 1976.

Grochowski, E. y Hoyt, R.F. [1996] "Future Trends in Hard Disk Drives", **IEEE Transactions on Magnetics**, 32:3, mayo de 1996.

Grosky, W. [1994] "Multimedia Information Systems", en **IEEE Multimedia**, 1:1, primavera de 1994.

Grosky,W. [1997] "Managing Multimedia Information in Database Systems", en **CACM**, 40:12, diciembre de 1997.

Grosky, W., Jain, R. y Mehrotra, R., eds. [1997], **The Handbook of Multimedia Information Management**, Prentice-Hall PTR, 1997.

Gruber, T. [1995] Toward principles for the design of ontologies used for knowledge sharing, **International Journal of Human-Computer Studies**, 43:5–6, Nov./Dic. 1995, págs. 907–928.

Guttman, A. [1984] "R-Trees: A Dynamic Index Structure for Spatial Searching", en *SIGMOD* [1984].

Gwayer, M. [1996] **Oracle Designer/2000 Web Server Generator Technical Overview** (versión 1.3.2), Technical Report, Oracle Corporation, septiembre de 1996.

Halsaal, F. [1996] **Data Communications, Computer Networks and Open Systems**, 4ª ed., Addison-Wesley, 1996.

Haas, P., Naughton, J., Seshadri, S. y Stokes, L. [1995] "Sampling-based Estimation of the Number of Distinct Values of an Attribute", en *VLDB* [1995].

Haas, P. y Swami, A. [1995] "Sampling-based Selectivity Estimation for Joins Using Augmented Frequent Value Statistics", en *ICDE* [1995].

Hachem, N. y Berra, P. [1992] "New Order Preserving Access Methods for Very Large Files Derived from Linear Hashing", **TKDE**, 4:1, febrero de 1992.

Hadzilacos, V. [1983] "An Operational Model for Database System Reliability", en *Proceedings of SIGACTSIGMOD Conference,* marzo de 1983.

Hadzilacos, V. [1986] "A Theory of Reliability in Database Systems", **JACM**, 35:1, 1986.

Haerder, T. y Rothermel, K. [1987] "Concepts for Transaction Recovery in Nested Transactions", en *SIGMOD* [1987].

Haerder, T. y Reuter, A. [1983] "Principles of Transaction Oriented Database Recovery—A Taxonomy", **ACM Computing Surveys**, 15:4, septiembre de 1983, págs. 287–318.

Hall, P. [1976] "Optimization of a Single Relational Expression in a Relational Data Base System", **IBM Journal of Research and Development**, 20:3, mayo de 1976.

Hamilton, G., Catteli, R. y Fisher, M. [1997] **JDBC Database Access with Java—A Tutorial and Annotated Reference**, Addison-Wesley, 1997.

Hammer, M. y McLeod, D. [1975] "Semantic Integrity in a Relational Data Base System", en *VLDB* [1975].

Hammer, M. y McLeod, D. [1981] "Database Description with SDM: A Semantic Data Model", **TODS**, 6:3, septiembre de 1981.

Hammer, M. y Sarin, S. [1978] "Efficient Monitoring of Database Assertions", en *SIGMOD* [1978].

Han, J. y M. Kamber, M. [2001] **Data Mining: Concepts and Techniques**, Morgan Kaufmann, 2001.

Han, J., Pei, J. y Yin, Y. [2000] "Mining Frequent Patterns without Candidate Generation", en *SIGMOD* [2000].

Hanson, E. [1992] "Rule Condition Testing and Action Execution in Ariel", en *SIGMOD* [1992].

Hardgrave, W. [1984] "BOLT: A Retrieval Language for Tree-Structured Database Systems", en TOU [1984].

Hardgrave, W. [1980] "Ambiguity in Processing Boolean Queries on TDMS Tree Structures: A Study of Four Different Philosophies", **TSE**, 6:4, julio de 1980.

Harrington, J. [1987] **Relational Database Management for Microcomputer: Design and Implementation**, Holt, Rinehart y Winston, 1987.

Harris, L. [1978] "The ROBOT System: Natural Language Processing Applied to Data Base Query", *Proc. ACM National Conference,* diciembre de 1978.

Haskin, R. y Lorie, R. [1982] "On Extending the Functions of a Relational Database System", en *SIGMOD* [1982].

Hasse, C. y Weikum, G. [1991] "A Performance Evaluation of Multi-Level Transaction Management", en *VLDB* [1991].

Hayes-Roth, F., Waterman, D. y Lenat, D., eds. [1983] **Building Expert Systems**, Addison-Wesley, 1983.

Hayne, S. y Ram, S. [1990] "Multi-User View Integration System: An Expert System for View Integration", en *ICDE* [1990].

Heiler, S. y Zdonick, S. [1990] "Object Views: Extending the Vision", en *ICDE* [1990].

Heiler, S., Hardhvalal, S., Zdonik, S., Blaustein, B. y Rosenthal, A. [1992] "A Flexible Framework for Transaction Management in Engineering Environment", en Elmagarmid [1992].

Helal, A., Hu, T., Elmasri, R. y Mukherjee, S. [1993] "Adaptive Transaction Scheduling", *CIKM,* noviembre de 1993.

Held, G. y Stonebraker, M. [1978] "B-Trees Reexamined", **CACM**, 21:2, febrero de 1978.

Henriksen, C., Lauzon, J.P. y Morehouse, S. [1994] "Open Geodata Access Through Standards", *StandardView Archive*, 1994, 2:3, págs. 169–174.

Henschen, L. y Naqvi, S. [1984], "On Compiling Queries in Recursive First-Order Databases", **JACM**, 31:1, enero de 1984.

Hernández, H. y Chan, E. [1991] "Constraint-Time-Maintainable BCNF Database Schemes", **TODS**, 16:4, diciembre de 1991.

Herot, C. [1980] "Spatial Management of Data", **TODS**, 5:4, diciembre de 1980.

Hevner, A. y Yao, S. [1979] "Query Processing in Distributed Database Systems", **TSE**, 5:3, mayo de 1979.

Hoffer, J. [1982] "An Empirical Investigation with Individual Differences in Database Models", *Proc. Third International Information Systems Conference,* diciembre de 1982.

Holland, J. [1975] **Adaptation in Natural and Artificial Systems**, University of Michigan Press, 1975.

Holsapple, C. y Whinston, A., eds. [1987] **Decisions Support Systems Theory and Application**, Springer-Verlag, 1987.

Holtzman J.M. y Goodman D.J., eds. [1993] **Wireless Communications: Future Directions**, Kluwer, 1993.

Hsiao, D. y Kamel, M. [1989] "Heterogeneous Databases: Proliferation, Issues, and Solutions", **TKDE**, 1:1, marzo de 1989.

Hsu, A. y Imielinsky, T. [1985] "Integrity Checking for Multiple Updates", en *SIGMOD* [1985].

Hull, R. y King, R. [1987] "Semantic Database Modeling: Survey, Applications, and Research Issues", **ACM Computing Surveys**, 19:3, septiembre de 1987.

Huxhold, W. [1991], **An Introduction to Urban Geographic Information Systems**, Oxford University Press, 1991.

IBM [1978] *QBE Terminal Users Guide*, Form Number SH20-2078-0.

IBM [1992] Systems Application Architecture Common Programming Interface Database Level 2 Reference, Números de documentos SC26-4798-01.

*ICDE* [1984] *Proc. IEEE CS International Conference on Data Engineering,* Shuey, R., ed., Los Ángeles, CA, abril de 1984.

*ICDE* [1986] *Proc. IEEE CS International Conference on Data Engineering,* Wiederhold, G., ed., Los Ángeles, febrero de 1986.

*ICDE* [1987] *Proc. IEEE CS International Conference on Data Engineering,*Wah, B., ed., Los Ángeles, febrero de 1987.

*ICDE* [1988] *Proc. IEEE CS International Conference on Data Engineering,* Carlis, J., ed., Los Ángeles, febrero de 1988.

*ICDE* [1989] *Proc. IEEE CS International Conference on Data Engineering,* Shuey, R., ed., Los Ángeles, febrero de 1989.

*ICDE* [1990] *Proc. IEEE CS International Conference on Data Engineering,* Liu, M., ed., Los Ángeles, febrero de 1990.

*ICDE* [1991] *Proc. IEEE CS International Conference on Data Engineering,* Cercone, N., y Tsuchiya, M., eds., Kobe, Japón, abril de 1991.

*ICDE* [1992] *Proc. IEEE CS International Conference on Data Engineering,* Golshani, F., ed., Phoenix, AZ, febrero de 1992.

*ICDE* [1993] *Proc. IEEE CS International Conference on Data Engineering,* Elmagarmid, A., y Neuhold, E., eds.,Viena, Austria, abril de 1993.

*ICDE* [1994] *Proc. IEEE CS International Conference on Data Engineering.*

*ICDE* [1995] *Proc. IEEE CS International Conference on Data Engineering,* Yu, P.S., y Chen, A.L. A., eds., Taipei, Taiwan, 1995.

*ICDE* [1996] *Proc. IEEE CS International Conference on Data Engineering,* Su, S.Y.W., ed., New Orleans, 1996.

*ICDE* [1997] *Proc. IEEE CS International Conference on Data Engineering,* Gray, A., y Larson, P. A., eds., Birmingham, Inglaterra, 1997.

*ICDE* [1998] *Proc. IEEE CS International Conference on Data Engineering,* Orlando, FL, 1998.

*ICDE* [1999] *Proc. IEEE CS International Conference on Data Engineering,* Sydney, Australia, 1999.

IGES [1983] International Graphics Exchange Specification Version 2, National Bureau of Standards, Departamento de comercio de Estados Unidos, enero de 1983.

Imielinski, T. y Badrinath, B. [1994] "Mobile Wireless Computing: Challenges in Data Management", **CACM**, 37:10, octubre de 1994.

Imielinski, T. y Lipski, W. [1981] "On Representing Incomplete Information in a Relational Database", en *VLDB* [1981].

Indulska, M. y Orlowska, M.E. [2002] "On Aggregation Issues in Spatial Data Management", (ACM International Conference Proceeding Series) *Proc. Thirteenth Australasian Conference on Database Technologies, Melbourne*, 2002, págs. 75–84.

Informix [1998] "Web Integration Option for Informix Dynamic Server", disponible en: http://www.infomix.com.

Inmon, W.H. [1992] **Building the Data Warehouse**, Wiley, 1992.

Internet Engineering Task Force (IETF) [1999] "An Architecture Framework for High Speed Mobile Ad Hoc Network", en *Proc. 45th IETF Meeting*, Oslo, Noruega, julio de 1999, disponible en: *http://www.ietf.org/proceedings/99jul/*.

Ioannidis, Y. y Kang, Y. [1990] "Randomized Algorithms for Optimizing Large Join Queries", en *SIGMOD* [1990].

Ioannidis, Y. y Kang, Y. [1991] "Left-Deep vs. Bushy Trees: An Analysis of Strategy Spaces and Its Implications for Query Optimization", en *SIGMOD* [1991].

Ioannidis, Y. y Wong, E. [1988] "Transforming Non-Linear Recursion to Linear Recursion", en *EDS* [1988].

Iossophidis, J. [1979] "A Translator to Convert the DDL of ERM to the DDL of System 2000", en la Conferencia ER [1979].

Irani, K., Purkayastha, S. y Teorey, T. [1979] "A Designer for DBMS-Processable Logical Database Structures", en *VLDB* [1979].

Jacobson, I., Booch, G. y Rumbaugh, J. [1999] **The Unified Software Development Process**, Addison-Wesley, 1999.

Jacobson, I., Christerson, M., Jonsson, P. y Overgaard, G. [1992] **Object Oriented Software Engineering: A Use Case Driven Approach**, Addison-Wesley, 1992.

Jagadish, H. [1989] "Incorporating Hierarchy in a Relational Model of Data", en *SIGMOD* [1989].

Jagadish, H. [1997] "Content-based Indexing and Retrieval", en Grosky y otros [1997].

Jajodia, S., Ammann, P., McCollum, C.D., *"Surviving Information Warfare Attacks",* **IEEE Computer**, Vol. 32, Tema 4, abril de 1999, págs. 57–63.

Jajodia, S. y Kogan, B. [1990] "Integrating an Objectoriented Data Model with Multilevel Security", *IEEE Symposium on Security and Privacy,* mayo de 1990, págs. 76–85.

Jajodia, S. y Mutchler, D. [1990] "Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database", **TODS**, 15:2, junio de 1990.

Jajodia, S., Ng, P. y Springsteel, F. [1983] "The Problem of Equivalence for Entity-Relationship Diagrams", **TSE**, 9:5, septiembre de 1983.

Jajodia, S. y Sandhu, R. [1991] "Toward a Multilevel Secure Relational Data Model", en *SIGMOD* [1991].

Jardine, D., ed. [1977] **The ANSI/SPARC DBMS Model**, North-Holland, 1977.

Jarke, M. y Koch, J. [1984] "Query Optimization in Database Systems", **ACM Computing Surveys**, 16:2, junio de 1984.

Jensen, C. y Snodgrass, R. [1992] "Temporal Specialization", in *ICDE* [1992].

Jensen, C. y otros [1994] "A Glossary of Temporal Database Concepts", **ACMSIGMOD Record**, 23:1, marzo de 1994.

Jensen, C., Friis-Christensen, A., Bach-Pedersen, T. y otros [2001] "Location-based Services: A Database Perspective", *Proc. ScanGIS Conference*, 2001, págs. 59–68.

Jing, J., Helal, A. y Elmagarmid, A. [1999] "Client-server Computing in Mobile Environments", **ACM Computing Surveys**, 31:2, junio de 1999.

Johnson, T. y Shasha, D. [1993] "The Performance of Current B-Tree Algorithms", **TODS**, 18:1, marzo de 1993.

Joshi, J., Aref, W., Ghafoor, A. y Spafford, E. [2001] "Security Models for Web-Based Applications", **CACM**, 44:2, febrero de 2001, págs. 38–44.

Kaefer,W. y Schoening, H. [1992] "Realizing a Temporal Complex-Object Data Model", en *SIGMOD* [1992].

Kamel, I. y Faloutsos, C. [1993] "On Packing R-trees", *CIKM,* noviembre de 1993.

Kamel, N. y King, R. [1985] "A Model of Data Distribution Based on Texture Analysis", en *SIGMOD* [1985].

Kapp, D. y Leben, J. [1978] **IMS Programming Techniques**, Van Nostrand-Reinhold, 1978.

Kappel, G. y Schrefl, M. [1991] "Object/Behavior Diagrams", en *ICDE* [1991].

Karlapalem, K., Navathe, S.B. y Ammar, M. [1996] "Optimal Redesign Policies to Support Dynamic Processing of Applications on a Distributed Relational Database System", **Information Systems**, 21:4, 1996, págs. 353–367.

Katz, R. [1985] **Information Management for Engineering Design: Surveys in Computer Science**, Springer-Verlag, 1985.

Katz, R. y Wong, E. [1982] "Decompiling CODASYL DML into Relational Queries", **TODS**, 7:1, marzo de 1982.

KDD [1996] *Proc. Second International Conference on Knowledge Discovery in Databases and Data Mining,* Portland, Oregón, agosto de 1996.

Kedem, Z. y Silberschatz, A. [1980] "Non-Two Phase Locking Protocols with Shared and Exclusive Locks", en *VLDB* [1980].

Keller, A. [1982] "Updates to Relational Database Through Views Involving Joins", en Scheuermann [1982].

Kemp, K. [1993]. "Spatial Databases: Sources and Issues", in **Environmental Modeling with GIS**, Oxford University Press, Nueva York, 1993.

Kemper, A., Lockemann, P. y Wallrath, M. [1987] "An Object-Oriented Database System for Engineering Applications", en *SIGMOD* [1987].

Kemper, A., Moerkotte, G. y Steinbrunn, M. [1992] "Optimizing Boolean Expressions in Object Bases", en *VLDB* [1992].

Kemper, A. y Wallrath, M. [1987] "An Analysis of Geometric Modeling in Database Systems", **ACM Computing Surveys**, 19:1, marzo de 1987.

Kent,W. [1978] **Data and Reality**, North-Holland, 1978.

Kent, W. [1979] "Limitations of Record-Based Information Models", **TODS**, 4:1, marzo de 1979.

Kent,W. [1991] "Object-Oriented Database Programming Languages", en *VLDB* [1991].

Kerschberg, L., Ting, P. y Yao, S. [1982] "Query Optimization in Star Computer Networks", **TODS**, 7:4, diciembre de 1982.

Ketabchi, M.A., Mathur, S., Risch, T. y Chen, J. [1990] "Comparative Analysis of RDBMS and OODBMS: A Case Study", *IEEE International Conference on Manufacturing,* 1990.

Khan, L. [2000] Ontology-based Information Selection, Ph.D. Dissertation, University of Southern California, agosto de 2000.

Khoshafian, S. y Baker A. [1996] **Multimedia and Imaging Databases**, Morgan Kaufmann, 1996.

Khoshafian, S., Chan, A., Wong, A. y Wong, H.K.T. [1992] **Developing Client Server Applications**, Morgan Kaufmann, 1992.

Kifer, M. y Lozinskii, E. [1986] "A Framework for an Efficient Implementation of Deductive Databases", *Proc. Sixth Advanced Database Symposium,* Tokio, agosto de 1986.

Kim, P. [1996] "A Taxonomy on the Architecture of Database Gateways for the Web", Working Paper TR-96-U-10, Chungnam National University, Taejon, Corea (disponible en: http://grigg.chungnam.ac.kr/projects/UniWeb).

Kim, W. [1982] "On Optimizing an SQL-like Nested Query", **TODS**, 3:3, septiembre de 1982.

Kim,W. [1989] "A Model of Queries for Object-Oriented Databases", en *VLDB* [1989].

Kim, W. [1990] "Object-Oriented Databases: Definition and Research Directions", **TKDE**, 2:3, septiembre de 1990.

Kim W. [1995] **Modern Database Systems: The Object Model**, **Interoperability**, **and Beyond**, ACM Press, Addison-Wesley, 1995.

Kim, W. y Lochovsky, F., eds. [1989] **Object-oriented Concepts**, **Databases**, **and Applications**, ACM Press, Frontier Series, 1989.

Kim, W., Reiner, D. y Batory, D., eds. [1985] **Query Processing in Database Systems**, Springer-Verlag, 1985.

Kim, W. y otros [1987] "*Features of the ORION Object-Oriented Database System*", Microelectronics and Computer Technology Corporation, Technical Report ACA-ST-308-87, septiembre de 1987.

Kimball, R. [1996] **The Data Warehouse Toolkit**, Wiley, Inc. 1996.

King, J. [1981] "QUIST: A System for Semantic Query Optimization in Relational Databases", en *VLDB* [1981].

Kitsuregawa, M., Nakayama, M. y Takagi, M. [1989] "The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method", en *VLDB* [1989].

Klimbie, J. y Koffeman, K., eds. [1974] **Data Base Management**, North-Holland, 1974.

Klug, A. [1982] "Equivalence of Relational Algebra and Relational Calculus Query Languages

Having Aggregate Functions", **JACM**, 29:3, julio de 1982.

Knuth, D. [1973] **The Art of Computer Programming**, **Vol. 3: Sorting and Searching**, Addison-Wesley, 1973.

Kogelnik, A. [1998] "*Biological Information Management with Application to Human Genome Data*", Ph.D. dissertation, Georgia Institute of Technology and Emory University, 1998.

Kogelnik, A., Lott, M., Brown, M., Navathe, S.,Wallace, D. [1998] "MITOMAP: A human mitochondrial genome database—1998 update", **Nucleic Acids Research**, 26:1, enero de 1998.

Kogelnik, A.,Navathe, S.,Wallace, D. [1997] "GENOME: A system for managing Human Genome Project Data". *Proceedings of Genome Informatics '97, Eighth Workshop on Genome Informatics,* Tokio, Japón, Patrocinador: Human Genome Center, Universidad de Tokio, diciembre de 1997.

Kohler, W. [1981] "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems", **ACMComputing Surveys**, 13:2, junio de 1981.

Konsynski, B., Bracker, L. y Bracker, W. [1982] "A Model for Specification of Office Communications", **IEEE Transactions on Communications**, 30:1, enero de 1982.

Korfhage, R. [1991] "To See, or Not to See: Is that the Query?" in *Proc. ACM SIGIR International Conference,* junio de 1991.

Korth, H. [1983] "Locking Primitives in a Database System", **JACM**, 30:1, enero de 1983.

Korth, H., Levy, E. y Silberschatz, A. [1990] "A Formal Approach to Recovery by Compensating Transactions", en *VLDB* [1990].

Kotz, A., Dittrich, K., Mulle, J. [1988] "Supporting Semantic Rules by a Generalized Event/Trigger Mechanism", en *VLDB* [1988].

Krishnamurthy, R., Litwin,W. y Kent,W. [1991] "Language Features for Interoperability of Databases with Semantic Discrepancies", en *SIGMOD* [1991].

Krishnamurthy, R. y Naqvi, S. [1988] "*Database Updates in Logic Programming, Rev. 1*", MCC Technical Report #ACA-ST-010-88, Rev. 1, septiembre de 1988.

Krishnamurthy, R. y Naqvi, S. [1989] "Non-Deterministic Choice in Datalog", *Proceeedings of the 3rd International Conference on Data and Knowledge Bases,* Jerusalén, junio de 1989.

Krovetz, R. y Croft B. [1992] "Lexical Ambiguity and Information Retrieval" in **TOIS**, 10, abril de 1992.

Kulkarni K., Carey, M., DeMichiel, L., Mattos, N., Hong, W. y Ubell M., "Introducing Reference Types and Cleaning Up SQL3's Object Model", *ISO WG3 Report X3H2-95-456,* noviembre de 1995.

Kumar, A. [1991] "Performance Measurement of Some Main Memory Recovery Algorithms", en *ICDE* [1991].

Kumar, A. y Segev, A. [1993] "Cost and Availability Tradeoffs in Replicated Concurrency Control", **TODS**, 18:1, marzo de 1993.

Kumar, A. y Stonebraker, M. [1987] "Semantics Based Transaction Management Techniques for Replicated Data", en *SIGMOD* [1987].

Kumar, V. y Han, M., eds. [1992] **Recovery Mechanisms in Database Systems**, Prentice-Hall, 1992.

Kumar, V. y Hsu, M. [1998] **Recovery Mechanisms in Database Systems**, Prentice-Hall (PTR), 1998.

Kumar, V. y Song, H.S. [1998] **Database Recovery**, Kluwer Academic, 1998.

Kung, H. y Robinson, J. [1981] "Optimistic Concurrency Control", **TODS**, 6:2, junio de 1981.

Lacroix, M. y Pirotte, A. [1977] "Domain-Oriented Relational Languages", en *VLDB* [1977].

Lacroix, M. y Pirotte, A. [1977a] "ILL: An English Structured Query Language for Relational Data Bases", en Nijssen [1977].

Lamport, L. [1978] "Time, Clocks y the Ordering of Events in a Distributed System",**CACM**, 21:7, julio de 1978.

Lander, E. [2001] "Initial Sequencing and Analysis of the Genome", **Nature**, 409:6822, 2001.

Langerak, R. [1990] "View Updates in Relational Databases with an Independent Scheme", **TODS**, 15:1, marzo de 1990.

Lanka, S. y Mays, E. [1991] "Fully Persistent B1-Trees", en *SIGMOD* [1991].

Larson, J. [1983] "Bridging the Gap Between Network and Relational Database Management Systems", **IEEE Computer**, 16:9, septiembre de 1983.

Larson, J., Navathe, S. y Elmasri, R. [1989] "Attribute Equivalence and its Use in Schema Integration", **TSE**, 15:2, abril de 1989.

Larson, P. [1978] "Dynamic Hashing", **BIT**, 18, 1978.

Larson, P. [1981] "Analysis of Index-Sequential Files with Overflow Chaining", **TODS**, 6:4, diciembre de 1981.

Lassila, O. [1998] "Web Metadata: A Matter of Semantics", **IEEE Internet Computing**, 2:4, julio/agosto de 1998, págs. 30–37.

Laurini, R. y Thompson, D. [1992] **Fundamentals of Spatial Information Systems**, Academic Press, 1992.

Lehman, P. y Yao, S. [1981] "Efficient Locking for Concurrent Operations on B-Trees", **TODS**, 6:4, diciembre de 1981.

Lee, J., Elmasri, R. y Won, J. [1998] "An Integrated Temporal Data Model Incorporating Time Series Concepts", **DKE**, 24, 1998, págs. 257–276.

Lehman, T. y Lindsay, B. [1989] "The Starburst Long Field Manager", en *VLDB* [1989].

Leiss, E. [1982] "Randomizing: A Practical Method for Protecting Statistical Databases Against Compromise", en *VLDB* [1982].

Leiss, E. [1982a] **Principles of Data Security**, Plenum Press, 1982.

Lenzerini, M. y Santucci, C. [1983] "Cardinality Constraints in the Entity Relationship Model", en la Conferencia ER [1983].

Leung, C., Hibler, B. y Mwara, N. [1992] "Picture Retrieval by Content Description", en **Journal of Information Science**, 1992, págs. 111–119.

Levesque, H. [1984] "The Logic of Incomplete Knowledge Bases", en Brodie y otros, Cap. 7 [1984].

Li, W., Seluk Candan, K., Hirata, K. y Hara, Y. [1998] Hierarchical Image Modeling for Object-based Media Retrieval in **DKE**, 27:2, septiembre de 1998, págs. 139–176.

Lien, E. y Weinberger, P. [1978] "Consistency, Concurrency, and Crash Recovery", en *SIGMOD* [1978].

Lieuwen, L. y DeWitt, D. [1992] "A Transformation-Based Approach to Optimizing Loops in Database Programming Languages", en *SIGMOD* [1992].

Lilien, L. y Bhargava, B. [1985] "Database Integrity Block Construct: Concepts and Design Issues", **TSE**, 11:9, septiembre de 1985.

Lin, J. y Dunham, M. H. [1998] "Mining Association Rules", en *ICDE* [1998].

Lindsay, B. y otros [1984] "Computation and Communication in R*: A Distributed Database Manager", **TOCS**, 2:1, enero de 1984.

Lippman R. [1987] "An Introduction to Computing with Neural Nets", **IEEE ASSPMagazine**, abril de 1987.

Lipski, W. [1979] "On Semantic Issues Connected with Incomplete Information", **TODS**, 4:3, septiembre de 1979.

Lipton, R., Naughton, J. y Schneider, D. [1990] "Practical Selectivity Estimation through Adaptive Sampling", en *SIGMOD* [1990].

Liskov, B. y Zilles, S. [1975] "Specification Techniques for Data Abstractions", **TSE**, 1:1, marzo de 1975.

Litwin, W. [1980] "Linear Hashing: A New Tool for File and Table Addressing", en *VLDB* [1980].

Liu, K. y Sunderraman, R. [1988] "On Representing Indefinite and Maybe Information in Relational Databases", en *ICDE* [1988].

Liu, L. y Meersman, R. [1992] "Activity Model: A Declarative Approach for Capturing Communication Behavior in Object-Oriented Databases", en *VLDB* [1992].

Livadas, P. [1989] **File Structures: Theory and Practice**, Prentice-Hall, 1989.

Lockemann, P. y Knutsen,W. [1968] "Recovery of Disk Contents After System Failure", **CACM**, 11:8, agosto de 1968.

Longley, P. et al [2001] **Geographic Information Systems and Science**, John Wiley, 2001.

Lorie, R. [1977] "Physical Integrity in a Large Segmented Database", **TODS**, 2:1, marzo de 1977.

Lorie, R. y Plouffe, W. [1983] "Complex Objects and Their Use in Design Transactions", en *SIGMOD* [1983].

Lozinskii, E. [1986] "A Problem-Oriented Inferential Database System", **TODS**, 11:3, septiembre de 1986.

Lu, H., Mikkilineni, K. y Richardson, J. [1987] "Design and Evaluation of Algorithms to Compute the Transitive Closure of a Database Relation", en *ICDE* [1987].

Lubars, M., Potts, C. y Richter, C. [1993] "A Review of the State of Practice in Requirements Modeling", *IEEE International Symposium on Requirements Engineering,* San Diego, CA, 1993.

Lucyk, B. [1993] **Advanced Topics in DB2**, Addison-Wesley, 1993.

Maguire, D., Goodchild, M. y Rhind, D., eds. [1997] **Geographical Information Systems: Principles and Applications.** Vols. 1 y 2, Longman Scientific and Technical, Nueva York.

Mahajan, S., Donahoo. M.J., Navathe, S.B., Ammar, M. y Malik, S. [1998] "Grouping Techniques for Update Propagation in Intermittently Connected Databases", en *ICDE* [1998].

Maier, D. [1983] **The Theory of Relational Databases**, Computer Science Press, 1983.

Maier, D., Stein, J., Otis, A. y Purdy, A. [1986] "Development of an Object-Oriented DBMS", *OOPSLA,* 1986.

Malley, C. y Zdonick, S. [1986] "A Knowledge-Based Approach to Query Optimization", en *EDS* [1986].

Maier, D. y Warren, D.S. [1988] **Computing with Logic**, Benjamin Cummings, 1988.

Mannila, H., Toivonen, H. y Verkamo, A. [1994] "Efficient Algorithms for Discovering Association Rules", en *KDD-94, AAAI Workshop on Knowledge Discovery in Databases,* Seattle, 1994.

Manola F. [1998] "Towards a Richer Web Object Model", en **ACM SIGMOD Record**, 27:1, marzo de 1998.

March, S. y Severance, D. [1977] "The Determination of Efficient Record Segmentations and Blocking Factors for Shared Files", **TODS**, 2:3, septiembre de 1977.

Mark, L., Roussopoulos, N., Newsome, T. y Laohapipattana, P. [1992] "Incrementally Maintained Network to Relational Mappings", **Software Practice & Experience**, 22:12, diciembre de 1992.

Markowitz, V. y Raz, Y. [1983] "ERROL: An Entity-Relationship, Role Oriented, Query Language", en la *Conferencia ER* [1983].

Martin, J., Chapman, K. y Leben, J. [1989] **DB2-Concepts**, **Design**, **and Programming**, Prentice-Hall, 1989.

Martin, J. y Odell, J. [1992] **Object Oriented Analysis and Design**, Prentice-Hall, 1992.

Maryanski, F. [1980] "Backend Database Machines", **ACM Computing Surveys**, 12:1, marzo de 1980.

Masunaga, Y. [1987] "Multimedia Databases: A Formal Framework", *Proc. IEEE Office Automation Symposium,* abril de 1987.

Mattison, R., **Data Warehousing: Strategies**, **Technologies** y **Techniques**, McGraw-Hill, 1996.

Maune, D.F. [2001] **Digital Elevation Model Technologies and Applications: The DEM Users Manual**, ASPRS, 2001.

McFadden, F. y Hoffer, J. [1988] **Database Management**, 2ª ed., Benjamin/Cummings, 1988.

McFadden, F.R. y Hoffer, J.A. [2005] **Modern Database Management**, 7ª ed., Prentice-Hall, 2005.

McGee,W. [1977] "The Information Management System IMS/VS, Part I: General Structure and Operation", **IBM Systems Journal**, 16:2, junio de 1977.

McLeish, M. [1989] "Further Results on the Security of Partitioned Dynamic Statistical Databases", **TODS**, 14:1, marzo de 1989.

McLeod, D. y Heimbigner, D. [1985] "A Federated Architecture for Information Systems", **TOOIS**, 3:3, julio de 1985.

Mehrotra, S. y otros [1992] "The Concurrency Control Problem in Multidatabases: Characteristics and Solutions", en *SIGMOD* [1992].

Melton, J. [2003] **Advanced SQL: 1999—Understanding Object-Relational and Other Advanced Features**, Morgan Kaufmann, 2003.

Melton, J., Bauer, J. y Kulkarni, K. [1991] "Object ADTs (with improvements for value ADTs)", *ISO WG3 Report X3H2-91-083,* abril de 1991.

Melton, J. y Mattos, N. [1996] *An Overview of SQL3—The Emerging New Generation of the SQL Standard,* Tutorial núm. T5, *VLDB*, Bombay, septiembre de 1996.

Melton, J. y Simon, A. R. [1993] **Understanding the New SQL: A Complete Guide**, Morgan Kaufmann, 1993.

Melton, J. y Simon, A.R. [2002] **SQL: 1999—Understanding Relational Language Components**, Morgan Kaufmann, 2002.

Menasce, D., Popek, G. y Muntz, R. [1980] "A Locking Protocol for Resource Coordination in Distributed Databases", **TODS**, 5:2, junio de 1980.

Mendelzon, A. y Maier, D. [1979] "Generalized Mutual Dependencies and the Decomposition of Database Relations", en *VLDB* [1979].

Mendelzon, A., Mihaila, G. y Milo, T. [1997] "Querying the World Wide Web", **Journal of Digital Libraries**, 1:1, abril de 1997.

Metais, E., Kedad, Z., Comyn-Wattiau, C. y Bouzeghoub, M."Using Linguistic Knowledge in View Integration: Toward a Third Generation of Tools", en **DKE** 23:1, junio de 1998.

Mikkilineni, K. y Su, S. [1988] "An Evaluation of Relational Join Algorithms in a Pipelined Query Processing Environment", **TSE**, 14:6, junio de 1988.

Miller, H.J., (2004) "Tobler's First Law and Spatial Analysis", Annals of the Association of American Geographers, 94:2, 2004, págs. 284–289.

Miller, N. [1987] **File Structures Using PASCAL**, Benjamin Cummings, 1987.

Milojicic, D. y otros [2002] *Peer-to-Peer Computing*, HP Laboratories Technical Report Núm. HPL-2002-57, HP Labs, Palo Alto, disponible en: *http://www.hpl.hp.com/techreports/2002/HPL-2002-57R1.html*

Minoura, T. y Wiederhold, G. [1981] "Resilient Extended True-Copy Token Scheme for a Distributed Database", **TSE**, 8:3, mayo de 1981.

Missikoff, M. y Wiederhold, G. [1984] "Toward a Unified Approach for Expert and Database Systems", en *EDS* [1984].

Mitchell, T. [1997] **Machine Learning**, McGraw Hill, 1997.

Mitschang, B. [1989] "Extending the Relational Algebra to Capture Complex Objects", en *VLDB* [1989].

Mohan, C. [1993] "IBM's Relational Database Products: Features and Technologies", en *SIGMOD* [1993].

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H. y Schwarz, P. [1992] "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", **TODS**, 17:1, marzo de 1992.

Mohan, C. y Levine, F. [1992] "ARIES/IM: An Efficient and High-Concurrency Index Management Method Using Write-Ahead Logging", en *SIGMOD* [1992].

Mohan, C. y Narang, I. [1992] "Algorithms for Creating Indexes for Very Large Tables without Quiescing Updates", en *SIGMOD* [1992].

Mohan, C. y otros [1992] "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", **TODS**, 17:1, marzo de 1992.

Morris, K., Ullman, J. y VanGelden, A. [1986] "Design Overview of the NAIL! System", *Proc. Third International Conference on Logic Programming,* Springer-Verlag, 1986.

Morris, K. y otros [1987] "YAWN! (Yet Another Window on NAIL!), en *ICDE* [1987].

Morris, R. [1968] "Scatter Storage Techniques", **CACM**, 11:1, enero de 1968.

Morsi, M., Navathe, S. y Kim, H. [1992] "An Extensible Object-Oriented Database Testbed", en *ICDE* [1992].

Moss, J. [1982] "Nested Transactions and Reliable Distributed Computing", *Proc. Symposium on Reliability in Distributed Software and Database Systems,* IEEE CS, julio de 1982.

Motro, A. [1987] "Superviews: Virtual Integration of Multiple Databases", **TSE**, 13:7, julio de 1987.

Mukkamala, R. [1989] "Measuring the Effect of Data Distribution and Replication Models on Performance Evaluation of Distributed Systems", en *ICDE* [1989].

Mumick, I., Finkelstein, S., Pirahesh, H. y Ramakrishnan, R. [1990] "Magic Is Relevant", en *SIGMOD* [1990].

Mumick, I., Pirahesh, H. y Ramakrishnan, R. [1990] "The Magic of Duplicates and Aggregates", en *VLDB* [1990].

Muralikrishna, M. [1992] "Improved Unnesting Algorithms for Join and Aggregate SQL Queries", en *VLDB* [1992].

Muralikrishna, M. y DeWitt, D. [1988] "Equi-depth Histograms for Estimating Selectivity Factors for Multi-dimensional Queries", en *SIGMOD* [1988].

Mylopolous, J., Bernstein, P. y Wong, H. [1980] "A Language Facility for Designing Database-Intensive Applications", **TODS**, 5:2, junio de 1980.

Naedele, M., [2003] *Standards for XML and Web Services Security*, **IEEE Computer**,Vol. 36, Núm. 4, págs. 96–98, abril de 2003.

Naish, L. y Thom, J. [1983] "*The MU-PROLOG Deductive Database*", Technical Report 83/10, Departamento de informática, Universidad de Melbourne, 1983.

Navathe, S. [1980] "An Intuitive Approach to Normalize Network-Structured Data", en *VLDB* [1980].

Navathe, S. y Ahmed, R. [1989] "A Temporal Relational Model and Query Language", **Information Sciences**, 47:2, marzo de 1989, págs. 147–175.

Navathe, S., Ceri, S., Wiederhold, G. y Dou, J. [1984] "Vertical Partitioning Algorithms for Database Design", **TODS**, 9:4, diciembre de 1984.

Navathe, S., Elmasri, R. y Larson, J. [1986] "Integrating User Views in Database Design", **IEEE Computer**, 19:1, enero de 1986.

Navathe, S. y Gadgil, S. [1982] "A Methodology for View Integration in Logical Database Design", en *VLDB* [1982].

Navathe, S.B, Karlapalem, K. y Ra, M.Y. [1996] "A Mixed Fragmentation Methodology for the Initial Distributed Database Design", **Journal of Computers and Software Engineering**, 3:4, 1996.

Navathe, S. y Kerschberg, L. [1986] "Role of Data Dictionaries in Database Design", **Information and Management**, 10:1, enero de 1986.

Navathe, S. y Patil, U. [2004] "Genomic and Proteomic Databases and Applications: A Challenge for Database Technology", *DASFAA 2004*, Jeju Island, Corea, LNCS: Vol. 2973, Springer-Verlag, marzo de 2004.

Navathe, S., Sashidhar, T. y Elmasri, R. [1984a] "Relationship Merging in Schema Integration", en *VLDB* [1984].

Navathe, S. y Savasere, A. [1996] "A Practical Schema Integration Facility Using an Object Oriented Approach", en **Multidatabase Systems** (A. Elmagarmid y O. Bukhres, eds.), Prentice-Hall, 1996.

Navathe, S.B., Savasere, A., Anwar, T.M., Beck, H. y Gala, S. [1994] "Object Modeling Using Classification in CANDIDE and Its Application", en Dogac y otros [1994].

Navathe, S. y Schkolnick, M. [1978] "View Representation in Logical Database Design", en *SIGMOD* [1978].

Negri, M., Pelagatti, S. y Sbatella, L. [1991] "Formal Semantics of SQL Queries", **TODS**, 16:3, septiembre de 1991.

Ng, P. [1981] "Further Analysis of the Entity-Relationship Approach to Database Design", **TSE**, 7:1, enero de 1981.

Nicolas, J. [1978] "Mutual Dependencies and Some Results on Undecomposable Relations", en *VLDB* [1978].

Nicolas, J. [1997] "Deductive Object-oriented Databases, Technology, Products, and Applications: Where Are We?" *Proc. Symposium on Digital Media Information Base (DMIB '97),* Nara, Japón, noviembre de 1997.

Nicolas, J., Phipps, G., Derr, M. y Ross, K. [1991] "Glue-NAIL!: A Deductive Database System", en *SIGMOD* [1991].

Nievergelt, J. [1974] "Binary Search Trees and File Organization", **ACM Computing Surveys**, 6:3, septiembre de 1974.

Nievergelt, J., Hinterberger, H. y Seveik, K. [1984]. "The Grid File: An Adaptable Symmetric Multikey File Structure", **TODS**, 9:1, marzo de 1984, págs. 38–71.

Nijssen, G., ed. [1976] **Modelling in Data Base Management Systems**, North-Holland, 1976.

Nijssen, G., ed. [1977] **Architecture and Models in Data Base Management Systems**, North-Holland, 1977.

Nwosu, K., Berra, P. y Thuraisingham, B., eds. [1996], **Design and Implementation of Multimedia Database Management Systems**, Kluwer Academic, 1996.

Obermarck, R. [1982] "Distributed Deadlock Detection Algorithms", **TODS**, 7:2, junio de 1982.

Oh, Y-C. [1999] "*Secure Database Modeling and Design*", Ph.D. dissertation, College of Computing, Georgia Institute of Technology, marzo de 1999.

Ohsuga, S. [1982] "Knowledge Based Systems as a New Interactive Computer System of the Next Generation", en **Computer Science and Technologies**, North-Holland, 1982.

Olken, F., Jagadish, J. [2003] "Data Management for Integrative Biology", **OMICS: A Journal of Integrative Biology**, 7:1, enero de 2003.

Olle, T. [1978] **The CODASYL Approach to Data Base Management**, Wiley, 1978.

Olle, T., Sol, H. y Verrijn-Stuart, A., eds. [1982] **Information System Design Methodology**, North-Holland, 1982.

Omiecinski, E. y Scheuermann, P. [1990] "A Parallel Algorithm for Record Clustering", **TODS**, 15:4, diciembre de 1990.

Omura, J.K., [1990] "Novel applications of cryptography in digital communications", **IEEE Communications Magazine**, 28: 5, mayo de 1990, págs. 21 –29.

O'Neill, P. [1994] **Database: Principles**, **Programming**, **Performance**, Morgan Kaufmann, 1994.

Open GIS Consortium, Inc. [1999] "*OpenGIS® Simple Features Specification For SQL*", Revision 1.1, OpenGIS Project Document 99-049, mayo de 1999.

Open GIS Consortium, Inc. [2003] "*OpenGIS® Geography Markup Language (GML) Implementation Specification*", Versión 3, OGC 02-023r4., 2003.

Oracle [1997a] **Oracle 8 Server Concepts**, Vols. 1 y 2, Versión 8-0, Oracle Corporation, 1997.

Oracle [1997b] **Oracle 8 Server Distributed Database Systems**, Versión 8.0, 1997.

Osborn, S. [1977] **Normal Forms for Relational Databases**, Ph.D. dissertation, Universidad de Waterloo, 1977.

Osborn, S. [1979] "Towards a Universal Relation Interface", en *VLDB* [1979].

Osborn, S. [1989] "The Role of Polymorphism in Schema Evolution in an Object-Oriented Database", **TKDE**, 1:3, septiembre de 1989.

Ozsoyoglu, G., Ozsoyoglu, Z. y Matos, V. [1985] "Extending Relational Algebra and Relational Calculus with Set Valued Attributes and Aggregate Functions", **TODS**, 12:4, diciembre de 1987.

Ozsoyoglu, Z. y Yuan, L. [1987] "A New Normal Form for Nested Relations", **TODS**, 12:1, marzo de 1987.

Ozsu, M.T. y Valduriez, P. [1999] **Principles of Distributed Database Systems**, 2ª ed., Prentice-Hall, 1999.

Papadimitriou, C. [1979] "The Serializability of Concurrent Database Updates", **JACM**, 26:4, octubre de 1979.

Papadimitriou, C. [1986] **The Theory of Database Concurrency Control**, Computer Science Press, 1986.

Papadimitriou, C. y Kanellakis, P. [1979] "On Concurrency Control by Multiple Versions", **TODS**, 9:1, marzo de 1974.

Papazoglou, M. y Valder, W. [1989] **Relational Database Management: A Systems Programming Approach**, Prentice-Hall, 1989.

Paredaens, J. y Van Gucht, D. [1992] "Converting Nested Algebra Expressions into Flat Algebra Expressions", **TODS**, 17:1, marzo de 1992.

Parent, C. y Spaccapietra, S. [1985] "An Algebra for a General Entity-Relationship Model", **TSE**, 11:7, julio de 1985.

Paris, J. [1986] "Voting with Witnesses: A Consistency Scheme for Replicated Files", en *ICDE* [1986].

Park, J., Chen, M. y Yu, P. [1995] "An Effective Hash Based Algorithm for Mining Association Rules", en *SIGMOD* [1995].

Paton, A.W., ed. [1999] **Active Rules in Database Systems**, Springer-Verlag, 1999.

Paton, N.W. y Díaz, O. [1999] Survey of Active Database Systems, **ACMComputing Surveys**.

Patterson, D., Gibson, G. y Katz, R. [1988]. "A Case for Redundant Arrays of Inexpensive Disks (RAID)", en *SIGMOD* [1988].

Paul, H. y otros [1987] "Architecture and Implementation of the Darmstadt Database Kernel System", en *SIGMOD* [1987].

Pazandak, P. y Srivastava, J., "Evaluating Object DBMSs for Multimedia", **IEEE Multimedia**, 4:3, págs. 34–49.

PDES [1991] "*A High-Lead Architecture for Implementing a PDES/STEP Data Sharing Environment*". Número de publicación PT 1017.03.00, PDES Inc., mayo de 1991.

Pearson, P., Francomano, C., Foster, P., Bocchini, C., Li, P. y McKusick, V. [1994] "*The Status of Online Mendelian Inheritance in Man (OMIM) Medio 1994*" **Nucleic Acids Research**, 22:17, 1994.

Peckham, J. y Maryanski, F. [1988] "Semantic Data Models", **ACM Computing Surveys**, 20:3, septiembre de 1988, págs. 153–189.

Peng, T. and Tsou, M. [2003] **Internet GIS: Distributed Geographic Information Services for the Internet and Wireless Network**, John Wiley, 2003.

Pfleeger, C.P. [1997] **Security in Computing**, Prentice-Hall, 1997.

Phipps, G., Derr, M., Ross, K. [1991] "Glue-NAIL!: A Deductive Database System", en *SIGMOD* [1991].

Piatesky-Shapiro, G. y Frauley,W., eds. [1991] **Knowledge Discovery in Databases**, AAAI Press/MIT Press, 1991.

Pistor P. y Anderson, F. [1986] "Designing a Generalized NF2 Model with an SQL-type Language Interface", en *VLDB* [1986], págs. 278–285.

Pitoura, E. y Bhargava, B. [1995] "Maintaining Consistency of Data in Mobile Distributed Environments". En *15ª ICDCS*, mayo de 1995, págs. 404–413.

Pitoura, E., Bukhres, O. y Elmagarmid, A. [1995] "Object Orientation in Multidatabase Systems", **ACM Computing Surveys**, 27:2, junio de 1995.

Pitoura, E. y Samaras, G. [1998] **Data Management for Mobile Computing**, Kluwer, 1998.

Ponniah, P. [2002] **Data Warehousing Fundamentals: A Comprehensive Guide for IT Professionals**, Wiley Interscience, 2002.

Poosala, V., Ioannidis, Y., Haas, P. y Shekita, E. [1996] "Improved Histograms for Selectivity Estimation of Range Predicates", en *SIGMOD* [1996].

Potter, B., Sinclair, J., Till, D. [1991] **An Introduction to Formal Specification and Z**, Prentice-Hall, 1991.

Prasad, S., Madisetti, V., Navathe, S. y otros, [2004], "SyD: A Middleware Testbed for Collaborative Applications over Small Heterogeneous Devices and Data Stores", *Proc. ACM/IFIP/USENIX 5ª International Middleware Conference (MW-04)*, Toronto, Canadá, octubre de 2004.

Price, B. [2004] "ESRI Systems Integration-Technical Brief—arcSDE High Availability overview", ESRI, 2004, Rev. 2. (*http://www.lincoln.ne.gov/city/pworks/gis/pdf/arcsde.pdf*).

Rabitti, F., Bertino, E., Kim, W. y Woelk, D. [1991] "A Model of Authorization for Next-Generation Database Systems", **TODS**, 16:1, marzo de 1991.

Ramakrishnan, R., ed. [1995] **Applications of Logic Databases**, Kluwer Academic, 1995.

Ramakrishnan, R. y Gehrke, J. [2003] **Database Management Systems**, 3ª ed., McGraw-Hill, 2003.

Ramakrishnan, R., Srivastava,D. y Sudarshan, S. [1992] "{CORAL}: {C}ontrol, {R}elations and {L}ogic", en *VLDB* [1992].

Ramakrishnan, R., Srivastava, D., Sudarshan, S. y Sheshadri, P. [1993] "Implementation of the {CORAL} deductive database system", en *SIGMOD* [1993].

Ramakrishnan, R. y Ullman, J. [1995] "Survey of Research in Deductive Database Systems", **Journal of Logic Programming**, 23:2, 1995, págs. 125–149.

Ramamoorthy, C. y Wah, B. [1979] "The Placement of Relations on a Distributed Relational Database", *Proc. First International Conference*

*on Distributed Computing Systems,* IEEE CS, 1979.

Ramesh,V. y Ram, S. [1997] "Integrity Constraint Integration in Heterogeneous Databases an Enhanced Methodology for Schema Integration", **Information Systems**, 22:8, diciembre de 1997, págs. 423–446.

Ratnasamy, S., Francis, P., Handley, M. y otros [2001] "A Scalable Content-Addressable Network". SIGCOMM 2001.

Reed, D. [1983] "Implementing Atomic Actions on Decentralized Data", **TOCS**, 1:1, febrero de 1983.

Reisner, P. [1977] "Use of Psychological Experimentation as an Aid to Development of a Query Language", **TSE**, 3:3, mayo de 1977.

Reisner, P. [1981] "Human Factors Studies of Database Query Languages: A Survey and Assessment", **ACM Computing Surveys**, 13:1, marzo de 1981.

Reiter, R. [1984] "Towards a Logical Reconstruction of Relational Database Theory", en Brodie y otros, capítulo 8 [1984].

Ries, D. y Stonebraker, M. [1977] "Effects of Locking Granularity in a Database Management System", **TODS**, 2:3, septiembre de 1977.

Rissanen, J. [1977] "Independent Components of Relations", **TODS**, 2:4, diciembre de 1977.

Rivest, R. y otros [1978] "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", **CACM**, 21:2, febrero de 1978, págs. 120–126.

Robbins, R. [1993] "Genome Informatics: Requirements and Challenges", *Proc. Second International Conference on Bioinformatics, Supercomputing and Complex Genome Analysis,* World Scientific Publishing, 1993.

Roth, M. y Korth, H. [1987] "The Design of Non-1NF Relational Databases into Nested Normal Form", en *SIGMOD* [1987].

Roth, M.A., Korth, H.F. y Silberschatz, A. [1988] Extended Algebra and Calculus for non-1NF relational Databases", **TODS**, 13:4, 1988, págs. 389–417.

Rothnie, J. y otros [1980] "Introduction to a System for Distributed Databases (SDD-1)", **TODS**, 5:1, marzo de 1980.

Roussopoulos, N. [1991] "An Incremental Access Method for View-Cache: Concept, Algorithms, and Cost Analysis", **TODS**, 16:3, septiembre de 1991.

Rozen, S. y Shasha, D. [1991] "A Framework for Automating Physical Database Design", en *VLDB* [1991].

Rudensteiner, E. [1992] "Multiview: A Methodology for Supporting Multiple Views in Object-Oriented Databases", en *VLDB* [1992].

Ruemmler, C. y Wilkes, J. [1994] "An Introduction to Disk Drive Modeling", **IEEE Computer**, 27:3, marzo de 1994, págs. 17–27.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. y Lorensen,W. [1991] **Object Oriented Modeling and Design**, Prentice-Hall, 1991.

Rumbaugh, J., Jacobson, I. y Booch, G. [1999] **The Unified Modeling Language Reference Manual**, Addison-Wesley, 1999.

Rusinkiewicz, M. y otros [1988] "OMNIBASE—A Loosely Coupled: Design and Implementation of a Multidatabase System", **IEEE Distributed Processing Newsletter**, 10:2, noviembre de 1988.

Rustin, R., ed. [1972] **Data Base Systems**, Prentice-Hall, 1972.

Rustin, R., ed. [1974] *Proc. BJNAV2.*

Sacca, D., and Zaniolo, C. [1987] "Implementation of Recursive Queries for a Data Language Based on Pure Horn Clauses", *Proc. Fourth International Conference on Logic Programming,* MIT Press, 1986.

Sadri, F. y Ullman, J. [1982] "Template Dependencies: A Large Class of Dependencies in Relational Databases and Its Complete Axiomatization", **JACM**, 29:2, abril de 1982.

Sagiv, Y. y Yannakakis, M. [1981] "Equivalence among Relational Expressions with the Union and Difference Operators", **JACM**, 27:4, noviembre de 1981.

Sakai, H. [1980] "Entity-Relationship Approach to Conceptual Schema Design", en *SIGMOD* [1980].

Salzberg, B. [1988] **File Structures: An Analytic Approach**, Prentice-Hall, 1988.

Salzberg, B. y otros [1990] "FastSort: A Distributed Single-Input Single-Output External Sort", en *SIGMOD* [1990].

Salton, G. y Buckley, C. [1991] "Global Text Matching for Information Retrieval" in **Science**, 253, agosto de 1991.

Samet, H. [1990] **The Design and Analysis of Spatial Data Structures**, Addison-Wesley, 1990.

Samet, H. [1990a] **Applications of Spatial Data Structures: Computer Graphics**, **Image Processing and GIS**, Addison-Wesley, 1990.

Sammut, C. y Sammut, R. [1983] "The Implementation of UNSW-PROLOG", **The Australian Computer Journal**, mayo de 1983.

Sarasua, W. y O'Neill, W. [1999]. "GIS in Transportation", en Taylor and Francis [1999].

Sarawagi, S., Thomas, S., Agrawal, R. [1998] "Integrating Association Rules Mining with Relational Database systems: Alternatives and Implications", en *SIGMOD* [1998].

Savasere, A., Omiecinski, E. y Navathe, S. [1995] "An Efficient Algorithm for Mining Association Rules", en *VLDB* [1995].

Savasere, A., Omiecinski, E. y Navathe, S. [1998] "Mining for Strong Negative Association in a Large Database of Customer Transactions", en *ICDE* [1998].

Schatz, B. [1995] "Information Analysis in the Net: The Interspace of the Twenty-First Century", *Keynote Plenary Lecture at American Society for Information Science (ASIS) Annual Meeting,* Chicago, 11 de octubre, 1995.

Schatz, B. [1997] "Information Retrieval in Digital Libraries: Bringing Search to the Net", **Science**, Vol. 275, 17 de enero, 1997.

Schek, H.J. y Scholl. M.H. [1986] "The Relational Model with Relation-valued Attributes", **Information Systems**, 11:2, 1986.

Schek, H.J., Paul, H.B., Scholl, M.H. y Weikum, G. [1990] "The DASDBS Project: Objects, Experiences, and Future Projects", **TKDE**, 2:1, 1990.

Scheuermann, P., Schiffner, G. y Weber, H. [1979] "Abstraction Capabilities and Invariant Properties Modeling within the Entity-Relationship Approach", en la *Conferencia ER* [1979].

Schlimmer, J., Mitchell, T., McDermott, J. [1991] "Justification Based Refinement of Expert Knowledge" en Piatesky-Shapiro and Frawley [1991].

Schlossnagle, G. [2005] **Advanced PHP Programming**, Sams, 2005.

Schmidt, J. y Swenson, J. [1975] "On the Semantics of the Relational Model", en *SIGMOD* [1975].

Sciore, E. [1982] "A Complete Axiomatization for Full Join Dependencies", **JACM**, 29:2, abril de 1982.

Selinger, P. y otros [1979] "Access Path Selection in a Relational Database Management System", en *SIGMOD* [1979].

Senko, M. [1975] "Specification of Stored Data Structures and Desired Output in DIAM II with FORAL", en *VLDB* [1975].

Senko, M. [1980] "A Query Maintenance Language for the Data Independent Accessing Model II", **Information Systems**, 5:4, 1980.

Shapiro, L. [1986] "Join Processing in Database Systems with Large Main Memories", **TODS**, 11:3, 1986.

Shasha, D. [1992] **Database Tuning: A Principled Approach**, Prentice-Hall, 1992.

Shasha, D. y Bonnet, P. [2002] **Database Tuning: Principles**, **Experiments**, **and Troubleshooting Techniques**, Morgan Kaufmann, Edición revisada, 2002

Shasha, D. y Goodman, N. [1988] "Concurrent Search Structure Algorithms", **TODS**, 13:1, marzo de 1988.

Shekhar, S. y Chawla, S. [2003] **Spatial Databases, A Tour**, Prentice-Hall, 2003.

Shekita, E. y Carey, M. [1989] "Performance Enhancement Through Replication in an Object-Oriented DBMS", en *SIGMOD* [1989].

Shenoy, S. y Ozsoyoglu, Z. [1989] "Design and Implementation of a Semantic Query Optimizer", **TKDE**, 1:3, septiembre de 1989.

Sheth, A., Gala, S., Navathe, S. [1993] "On Automatic Reasoning for Schema Integration", en **International Journal of Intelligent Cooperative Information Systems**, 2:1, marzo de 1993.

Sheth, A.P. y Larson, J.A. [1990] "Federated Database Systems for Managing Distributed,

Heterogeneous, and Autonomous Databases", **ACM Computing Surveys**, 22:3, septiembre de 1990, págs. 183–236.

Sheth, A., Larson, J., Cornelio, A. y Navathe, S. [1988] "A Tool for Integrating Conceptual Schemas and User Views", en *ICDE* [1988].

Shipman, D. [1981] "The Functional Data Model and the Data Language DAPLEX", **TODS**, 6:1, marzo de 1981.

Shlaer, S., Mellor, S. [1988] **Object-Oriented System Analysis: Modeling the World in Data**, Yourdon Press, 1988.

Shneiderman, B., ed. [1978] **Databases: Improving Usability and Responsiveness**, Academic Press, 1978.

Sibley, E. y Kerschberg, L. [1977] "Data Architecture and Data Model Considerations", *NCC, AFIPS,* 46, 1977.

Siegel, M. y Madnick, S. [1991] "A Metadata Approach to Resolving Semantic Conflicts", en *VLDB* [1991].

Siegel, M., Sciore, E. y Salveter, S. [1992] "A Method for Automatic Rule Derivation to Support Semantic Query Optimization", **TODS**, 17:4, diciembre de 1992.

*SIGMOD* [1974] *Proc. ACM SIGMOD-SIGFIDET Conference on Data Description, Access y Control,* Rustin, R., ed., mayo de 1974.

*SIGMOD* [1975] *Proc. 1975 ACM SIGMOD International Conference on Management of Data,* King, F., ed., San José, CA, mayo 1975.

*SIGMOD* [1976] *Proc. 1976 ACM SIGMOD International Conference on Management of Data,* Rothnie, J., ed., Washington, junio de 1976.

*SIGMOD* [1977] *Proc. 1977 ACM SIGMOD International Conference on Management of Data,* Smith, D., ed., Toronto, agosto de 1977.

*SIGMOD* [1978] *Proc. 1978 ACM SIGMOD International Conference on Management of Data,* Lowenthal, E. y Dale, N., eds., Austin, TX, mayo/junio de 1978.

*SIGMOD* [1979] *Proc. 1979 ACM SIGMOD International Conference on Management of Data,* Bernstein, P., ed., Boston, MA, mayo/junio de 1979.

*SIGMOD* [1980] *Proc. 1980 ACM SIGMOD International Conference on Management of Data,* Chen, P. y Sprowls, R., eds., Santa Mónica, CA, mayo de 1980.

*SIGMOD* [1981] *Proc. 1981 ACM SIGMOD International Conference on Management of Data,* Lien, Y., ed., Ann Arbor, MI, abril/mayo de 1981.

*SIGMOD* [1982] *Proc. 1982 ACM SIGMOD International Conference on Management of Data,* Schkolnick, M., ed., Orlando, FL, junio de 1982.

*SIGMOD* [1983] *Proc. 1983 ACM SIGMOD International Conference on Management of Data,* DeWitt, D., y Gardarin, G., eds., San José, CA, mayo de 1983.

*SIGMOD* [1984] *Proc. 1984 ACM SIGMOD International Conference on Management of Data,* Yormark, E., ed., Boston, MA, junio de 1984.

*SIGMOD* [1985] *Proc. 1985 ACM SIGMOD International Conference on Management of Data,* Navathe, S., ed., Austin, TX, mayo de 1985.

*SIGMOD* [1986] *Proc. 1986 ACM SIGMOD International Conference on Management of Data,* Zaniolo, C., ed., Washington, mayo de 1986.

*SIGMOD* [1987] *Proc. 1987 ACM SIGMOD International Conference on Management of Data,* Dayal, U., and Traiger, I., eds., San Francisco, CA, mayo de 1987.

*SIGMOD* [1988] *Proc. 1988 ACM SIGMOD International Conference on Management of Data,* Boral, H., and Larson, P., eds., Chicago, junio de 1988.

*SIGMOD* [1989] *Proc. 1989 ACM SIGMOD International Conference on Management of Data,* Clifford, J., Lindsay, B., y Maier, D., eds., Portland, OR, junio de 1989.

*SIGMOD* [1990] *Proc. 1990 ACM SIGMOD International Conference on Management of Data,* Garcia-Molina, H., y Jagadish, H., eds., Atlantic City, NJ, junio de 1990.

*SIGMOD* [1991] *Proc. 1991 ACM SIGMOD International Conference on Management of Data,* Clifford, J., y King, R., eds., Denver, CO, junio de 1991.

*SIGMOD* [1992] *Proc. 1992 ACM SIGMOD International Conference on Management of*

*Data,* Stonebraker, M., ed., San Diego, CA, junio de 1992.

*SIGMOD* [1993] *Proc. 1993 ACM SIGMOD International Conference on Management of Data,* Buneman, P., y Jajodia, S., eds., Washington, junio de 1993.

*SIGMOD* [1994] *Proceedings of 1994 ACMSIGMOD International Conference on Management of Data,* Snodgrass, R. T., y Winslett, M., eds., Minneapolis, MN, junio de 1994.

*SIGMOD* [1995] *Proceedings of 1995 ACMSIGMOD International Conference on Management of Data,* Carey, M., y Schneider, D.A., eds., Minneapolis, MN, junio de 1995.

*SIGMOD* [1996] *Proceedings of 1996 ACMSIGMOD International Conference on Management of Data,* Jagadish, H.V., y Mumick, I.P., eds., Montreal, junio de 1996.

*SIGMOD* [1997] *Proceedings of 1997 ACMSIGMOD International Conference on Management of Data,* Peckham, J., ed., Tucson, AZ, mayo de 1997.

*SIGMOD* [1998] *Proceedings of 1998 ACMSIGMOD International Conference on Management of Data,* Haas, L., y Tiwary, A., eds., Seattle, WA, junio de 1998.

*SIGMOD* [1999] *Proceedings of 1999 ACMSIGMOD International Conference on Management of Data,* Faloutsos, C., ed., Filadelfia, PA, mayo de 1999.

*SIGMOD* [2000] *Proceedings of 2000 ACMSIGMOD International Conference on Management of Data,* Chen, W., Naughton J. y Bernstein, P., ed., Dallas, TX, mayo de 2000.

*SIGMOD* [2001] *Proceedings of 2001 ACMSIGMOD International Conference on Management of Data,* Aref, W., ed., Santa Bárbara, CA, mayo de 2001.

*SIGMOD* [2002] *Proceedings of 2002 ACMSIGMOD International Conference on Management of Data,* Franklin, M., Moon, B. y Ailamaki, A., eds., Madison, WI, junio de 2002.

*SIGMOD* [2003] *Proceedings of 2003 ACM SIGMOD International Conference on Management of Data,* Halevy, Y., Zachary, G. y Doan, A., eds., San Diego, CA, junio de 2003.

*SIGMOD* [2004] *Proceedings of 2004 ACMSIGMOD International Conference on Management of Data,* Weikum, G., Christian König, A. y DeBloch, S., eds., París, Francia, junio de 2004.

*SIGMOD* [2005] *Proceedings of 2005 ACMSIGMOD International Conference on Management of Data,* Widom, J., ed., Baltimore,MD, junio de 2005.

Silberschatz, A., Stonebraker, M. y Ullman, J. [1990] "Database Systems: Achievements and Opportunities", en **ACM SIGMOD Record**, 19:4, diciembre de 1990.

Silberschatz, A., Korth, H. y Sudarshan, S. [2001] Database System Concepts, 5ª ed., McGraw-Hill, 2006.

Sion, R., Atallah, M. y Prabhakar, S. [2004] "Protecting Rights Proofs for Relational Data Using Watermarking", **TKDE**, 16:12, 2004, págs. 1509–1525.

Sklar, D. [2005] **Learning PHP5**, O'Reilly Media, Inc., 2005.

Smith, G. [1990] "The Semantic Data Model for Security: Representing the Security Semantics of an Application", en *ICDE* [1990].

Smith, J. y Chang, P. [1975] "Optimizing the Performance of a Relational Algebra Interface", **CACM**, 18:10, octubre de 1975.

Smith, J. y Smith, D. [1977] "Database Abstractions: Aggregation and Generalization", **TODS**, 2:2, junio de 1977.

Smith, J. y otros [1981] "MULTIBASE: Integrating Distributed Heterogeneous Database Systems", *NCC, AFIPS,* 50, 1981.

Smith, K. y Winslett, M. [1992] "Entity Modeling in the MLS Relational Model", en *VLDB* [1992].

Smith, P. y Barnes, G. [1987] **Files and Databases: An Introduction**, Addison-Wesley, 1987.

Snodgrass, R. [1987] "The Temporal Query Language TQuel", **TODS**, 12:2, junio de 1987.

Snodgrass, R., ed. [1995] **The TSQL2 Temporal Query Language**, Kluwer, 1995.

Snodgrass, R. y Ahn, I. [1985] "A Taxonomy of Time in Databases", en *SIGMOD* [1985].

Soutou, G. [1998] "Analysis of Constraints for N-ary Relationships", en ER98.

Spaccapietra, S. y Jain, R., eds. [1995] *Proc.Visual Database Workshop,* Lausanne, Suiza, octubre de 1995.

Spooner D., Michael, A. y Donald, B. [1986] "Modeling CAD Data with Data Abstraction and Object Oriented Technique", en *ICDE* [1986].

Srikant, R. y Agrawal, R. [1995] "Mining Generalized Association Rules", en *VLDB* [1995].

Srinivas, M. y Patnaik, L. [1994] "Genetic Algorithms: A Survey", **IEEE Computer**, junio de 1994.

Srinivasan, V. y Carey, M. [1991] "Performance of BTree Concurrency Control Algorithms", en *SIGMOD* [1991].

Srivastava, D., Ramakrishnan, R., Sudarshan, S. y Sheshadri, P. [1993] "Coral++: Adding Object-orientation to a Logic Database Language", en *VLDB* [1993].

Stachour, P. y Thuraisingham, B. [1990] "The Design and Implementation of INGRES", **TKDE**, 2:2, junio de 1990.

Stallings, W. [1997] Data and Computer Communications, 5ª ed., Prentice-Hall, 1997.

Stallings, W., [2000] **Network Security Essentials**, Applications and Standards, Prentice-Hall, 2000.

Stevens, P. y Pooley, R. [2000] **Using UML: Software Engineering with Objects and Components**, Edición actualizada, Addison-Wesley, 2000.

Stoesser, G. y otros [2003], "The EMBL Nucleotide Sequence Database: Major New Developments", **Nucleic Acids Research**, 31:1, enero de 2003, págs. 17–22.

Stoica, I., Morris, R., Karger, D. y otros [2001] "Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications", SIGCOMM 2001.

Stonebraker, M. [1975] "Implementation of Integrity Constraints and Views by Query Modification", en *SIGMOD* [1975].

Stonebraker, M. [1993] "The Miro DBMS" in *SIGMOD* [1993].

Stonebraker, M., ed. [1994] **Readings in Database Systems**, 2ª ed., Morgan Kaufmann, 1994.

Stonebraker, M., Hanson, E. y Hong, C. [1987] "The Design of the POSTGRES Rules System", en *ICDE* [1987].

Stonebraker, M., con Moore, D. [1996], **Object-Relational DBMSs: The Next Great Wave**, Morgan Kaufmann, 1996.

Stonebraker, M. y Rowe, L. [1986] "The Design of POSTGRES", en *SIGMOD* [1986].

Stonebraker, M., Wong, E., Kreps, P. y Held, G. [1976] "The Design and Implementation of INGRES", **TODS**, 1:3, septiembre de 1976.

Su, S. [1985] "A Semantic Association Model for Corporate and Scientific-Statistical Databases", **Information Science**, 29, 1985.

Su, S. [1988] **Database Computers**, McGraw-Hill, 1988.

Su, S., Krishnamurthy, V. y Lam, H. [1988] "An Object-Oriented Semantic Association Model (OSAM*)", en **AI in Industrial Engineering and Manufacturing: Theoretical Issues and Applications**, American Institute of Industrial Engineers, 1988.

Subrahmanian, V. [1998] **Principles of Multimedia Databases Systems**, Morgan Kaufmann, 1998.

Subrahmanian V.S. y Jajodia, S., eds. [1996] **Multimedia Database Systems: Issues and Research Directions**, Springer-Verlag. 1996.

Sunderraman, R. [2005] **ORACLE 9i Programming: A Primer**, Addison-Wesley Longman, 2005.

Swami, A. y Gupta, A. [1989] "Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques", en *SIGMOD* [1989].

Tanenbaum, A. [2003] **Computer Networks**, 4ª ed., Prentice-Hall PTR, 2003.

Tan, P., Steinbach, M. y Kumar, V. [2006] *Introduction to Data Mining*, Addison-Wesley, 2006.

Tansel, A. y otros, eds. [1993] **Temporal Databases: Theory**, **Design** y **Implementation**, Benjamin Cummings, 1993.

Teorey, T. [1994] **Database Modeling and Design: The Fundamental Principles**, 2ª ed., Morgan Kaufmann, 1994.

Teorey, T., Yang, D. y Fry, J. [1986] "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model", **ACM Computing Surveys**, 18:2, junio de 1986.

Thomas, J. y Gould, J. [1975] "A Psychological Study of Query by Example", *NCC AFIPS,* 44, 1975.

Thomas, R. [1979] "A Majority Consensus Approach to Concurrency Control for Multiple Copy Data Bases", **TODS**, 4:2, junio de 1979.

Thomasian, A. [1991] "Performance Limits of Two-Phase Locking", en *ICDE* [1991].

Thuraisingham, B., Clifton, C., Gupta, A., Bertino, E. y Ferrari, E. [2001] "Directions for Web and E-commerce Applications Security", *Proc. Tenth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 2001, págs. 200–204.

Todd, S. [1976] "The Peterlee Relational Test Vehicle—A System Overview", **IBM Systems Journal**, 15:4, diciembre de 1976.

Toivonen, H., "Sampling Large Databases for Association Rules", en *VLDB* [1996].

Tou, J., ed. [1984] **Information Systems COINS-IV**, Plenum Press, 1984.

Tsangaris, M. y Naughton, J. [1992] "On the Performance of Object Clustering Techniques", en *SIGMOD* [1992].

Tsichritzis, D. [1982] "Forms Management", **CACM**, 25:7, julio de 1982.

Tsichritzis, D. y Klug, A., eds. [1978] **The ANSI/X3 /SPARC DBMS Framework**, AFIPS Press, 1978.

Tsichritzis, D. y Lochovsky, F. [1976] "Hierarchical Database Management: A Survey", **ACM Computing Surveys**, 8:1, marzo de 1976.

Tsichritzis, D. y Lochovsky, F. [1982] **Data Models**, Prentice-Hall, 1982.

Tsotras, V. y Gopinath, B. [1992] "Optimal Versioning of Object Classes", en *ICDE* [1992].

Tsou, D.M. y Fischer, P.C. [1982] "Decomposition of a Relation Scheme into Boyce Codd Normal Form", *SIGACT News,* 14:3, 1982, págs. 23–29.

Ullman, J. [1982] **Principles of Database Systems**, 2ª ed., Computer Science Press, 1982.

Ullman, J. [1985] "Implementation of Logical Query Languages for Databases", **TODS**, 10:3, septiembre de 1985.

Ullman, J. [1988] **Principles of Database and Knowledge-Base Systems**, Vol. 1, Computer Science Press, 1988.

Ullman, J. [1989] **Principles of Database and Knowledge-Base Systems**, Vol. 2, Computer Science Press, 1989.

Ullman, J.D. y Widom, J. [1997] **A First Course in Database Systems**, Prentice-Hall, 1997.

U.S. Congress [1988] "Office of Technology Report, Appendix D: Databases, Repositories y Informatics", en **Mapping Our Genes: Genome Projects: How Big**, **How Fast?** John Hopkins University Press, 1988.

U.S. Department of Commerce [1993]. **TIGER/ Line Files**, Bureau of Census, Washington, 1993.

Uschold, M. y Gruninger, M. [1996] "Ontologies: Principles, Methods and Applications", *Knowledge Engineering Review*, 11:2, junio de 1996.

Vaidya, J. y Clifton, C., "Privacy-Preserving Data Mining: Why, How y What For?" **IEEE Security & Privacy**, noviembre/diciembre de 2004, págs. 19–27.

Valduriez, P. y Gardarin, G. [1989] **Analysis and Comparison of Relational Database Systems**, Addison-Wesley, 1989.

Vassiliou,Y. [1980] "Functional Dependencies and Incomplete Information", en *VLDB* [1980].

Verheijen, G. y VanBekkum, J. [1982] "NIAM: An Information Analysis Method", en Olle y otros [1982].

Verhofstadt, J. [1978] "Recovery Techniques for Database Systems", **ACM Computing Surveys**, 10:2, junio de 1978.

Vielle, L. [1986] "Recursive Axioms in Deductive Databases: The Query-Subquery Approach", en *EDS* [1986].

Vielle, L. [1987] "Database Complete Proof Production Based on SLD-resolution", en *Proc. Fourth International Conference on Logic Programming,* 1987.

Vielle, L. [1988] "From QSQ Towards QoSaQ: Global Optimization of Recursive Queries", en *EDS* [1988].

Vielle, L. [1998] "VALIDITY: Knowledge Independence for Electronic Mediation", envited paper, in *Practical Applications of Prolog/ Practical*

*Applications of Constraint Technology (PAP/ PACT '98),* Londres, marzo de 1998, disponible en *lvieille@computer.org*.

Vin, H., Zellweger, P., Swinehart,D. y Venkat Rangan, P. [1991] "Multimedia Conferencing in the Etherphone Environment", **IEEE Computer**, Special Issue on Multimedia Information Systems, 24:10, octubre de 1991.

*VLDB* [1975] *Proc. First International Conference on Very Large Data Bases,* Kerr, D., ed., Framingham, MA, septiembre de 1975.

*VLDB* [1976] **Systems for Large Databases**, Lockemann, P. y Neuhold, E., eds., en *Proc. Second International Conference on Very Large Data Bases,* Bruselas, Bélgica, julio de 1976, North-Holland, 1976.

*VLDB* [1977] *Proc. Third International Conference on Very Large Data Bases,*Merten, A., ed., Tokio, Japón, octubre de 1977.

*VLDB* [1978] *Proc. Fourth International Conference on Very Large Data Bases,* Bubenko, J., y Yao, S., eds., Berlín Occidental, Alemania, septiembre de 1978.

*VLDB* [1979] *Proc. Fifth International Conference on Very Large Data Bases,* Furtado, A., y Morgan, H., eds., Río de Janeiro, Brasil, octubre de 1979.

*VLDB* [1980] *Proc. Sixth International Conference on Very Large Data Bases,* Lochovsky, F., y Taylor, R., eds., Montreal, Canadá, octubre de 1980.

*VLDB* [1981] *Proc. Seventh International Conference on Very Large Data Bases,* Zaniolo, C., y Delobel, C., eds., Cannes, Francia, septiembre de 1981.

*VLDB* [1982] *Proc. Eighth International Conference on Very Large Data Bases,*McLeod, D., y Villasenor, Y., eds., Ciudad de México, septiembre de 1982.

*VLDB* [1983] *Proc. Ninth International Conference on Very Large Data Bases,* Schkolnick, M., y Thanos, C., eds., Florencia, Italia, octubre/ noviembre de 1983.

*VLDB* [1984] *Proc. Tenth International Conference on Very Large Data Bases,* Dayal, U., Schlageter, G., y Seng, L., eds., Singapur, agosto de 1984.

*VLDB* [1985] *Proc. Eleventh International Conference on Very Large Data Bases,* Pirotte, A., y Vassiliou, Y., eds., Estocolmo, Suecia, agosto de 1985.

*VLDB* [1986] *Proc. Twelfth International Conference on Very Large Data Bases,* Chu, W., Gardarin, G., y Ohsuga, S., eds., Kyoto, Japón, agosto de 1986.

*VLDB* [1987] *Proc. Thirteenth International Conference on Very Large Data Bases,* Stocker, P., Kent,W., y Hammersley, P., eds., Brighton, Inglaterra, septiembre de 1987.

*VLDB* [1988] *Proc. Fourteenth International Conference on Very Large Data Bases,* Bancilhon, F., y DeWitt, D., eds., Los Ángeles, agosto/septiembre de 1988.

*VLDB* [1989] *Proc. Fifteenth International Conference on Very Large Data Bases,* Apers, P., y Wiederhold, G., eds., Amsterdam, agosto de 1989.

*VLDB* [1990] *Proc. Sixteenth International Conference on Very Large Data Bases,* McLeod, D., Sacks-Davis, R., y Schek, H., eds., Brisbane, Australia, agosto de 1990.

*VLDB* [1991] *Proc. Seventeenth International Conference on Very Large Data Bases,* Lohman, G., Sernadas, A., y Camps, R., eds., Barcelona, Cataluña, España, septiembre de 1991.

*VLDB* [1992] *Proc. Eighteenth International Conference on Very Large Data Bases,* Yuan, L., ed., Vancouver, Canadá, agosto de 1992.

*VLDB* [1993] *Proc. Nineteenth International Conference on Very Large Data Bases,* Agrawal, R., Baker, S., y Bell, D.A., eds., Dublín, Irlanda, agosto de 1993.

*VLDB* [1994] *Proc. 20th International Conference on Very Large Data Bases,* Bocca, J., Jarke, M., y Zaniolo, C., eds., Santiago, Chile, septiembre de 1994.

*VLDB* [1995] *Proc. 21st International Conference on Very Large Data Bases,* Dayal, U., Gray, P.M.D., y Nishio, S., eds., Zurich, Suiza, septiembre de 1995.

*VLDB* [1996] *Proc. 22nd International Conference on Very Large Data Bases,* Vijayaraman, T.M., Buchman, A.P., Mohan, C. y Sarda, N.L., eds., Bombay, India, septiembre de 1996.

*VLDB* [1997] *Proc. 23rd International Conference on Very Large Data Bases,* Jarke, M., Carey, M.J., Dittrich, K.R., Lochovsky, F.H., y Loucopoulos, P., eds., Zurich, Suiza, septiembre de 1997.

*VLDB* [1998] *Proc. 24th International Conference on Very Large Data Bases,* Gupta, A., Shmueli, O., y Widom, J., eds., Nueva York, septiembre de 1998.

*VLDB* [1999] *Proc. 25th International Conference on Very Large Data Bases,* Zdonik, S.B., Valduriez, P., y Orlowska, M., eds., Edinburgo, Escocia, septiembre de 1999.

*VLDB* [2000] *Proc. 26th International Conference on Very Large Data Bases,* Abbadi, A., Brodie, M., Chakravarthy, S., Dayal, U., Kamel, N., Schlageter, G. y Whang, K-Y., eds., El Cairo, Egipto, septiembre de 2000.

*VLDB* [2001] *Proc. 27th International Conference on Very Large Data Bases,* Apers, P., Atzeni, P., Ceri, S., Paraboschi, S., Ramamohanarao, K. y Snodgrass, R., eds., Roma, Italia, septiembre de 2001.

*VLDB* [2002] *Proc. 28th International Conference on Very Large Data Bases,* Bernstein, P., Ionnidis, Y. y Ramakrishnan, R., eds., Hong Kong, China, agosto de 2002.

*VLDB* [2003] *Proc. 29th International Conference on Very Large Data Bases,* Freytag, J., Lockemann, P., Abiteboul, S., Carey, M., Selinger, P. y Heuer, A., eds., Berlín, Alemania, septiembre de 2003.

*VLDB* [2004] *Proc. 30th International Conference on Very Large Data Bases,* Nascimento, M., Özsu, T., Kossmann, D., Miller, R., Blakeley, J., y Schiefer, K., eds., Toronto, Canadá, septiembre de 2004.

*VLDB* [2005] *Proc. 25th International Conference on Very Large Data Bases,* Böhm, K., Jensen, C., Haas, L., Kersten, M., Larson, P-A. y Ooi, B., eds., Trondheim, Noruega, agosto-septiembre de 2005.

Vorhaus, A. y Mills, R. [1967] "The Time-Shared Data Management System: A New Approach to Data Management", System Development Corporation, Report SP-2634, 1967.

Wallace, D. [1995] "1994 William Allan Award Address: Mitochondrial DNA Variation in Human Evolution, Degenerative Disease y Aging". **American Journal of Human Genetics**, 57:201–223, 1995.

Walton, C., Dale, A. y Jenevein, R. [1991] "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins", en *VLDB* [1991].

Wang, K. [1990] "Polynomial Time Designs Toward Both BCNF and Efficient Data Manipulation", en *SIGMOD* [1990].

Wang, Y. y Madnick, S. [1989] "The Inter-Database Instance Identity Problem in Integrating Autonomous Systems", en *ICDE* [1989].

Wang, Y. y Rowe, L. [1991] "Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture", en *SIGMOD* [1991].

Warren, D. [1992] "Memoing for Logic Programs", **CACM**, 35:3, ACM, marzo de 1992.

Weddell, G. [1992] "Reasoning About Functional Dependencies Generalized for Semantic Data Models", **TODS**, 17:1, marzo de 1992.

Weikum, G. [1991] "Principles and Realization Strategies of Multilevel Transaction Management", **TODS**, 16:1, marzo de 1991.

Weiss, S. y Indurkhya, N. [1998] **Predictive Data Mining: A Practical Guide**, Morgan Kaufmann, 1998.

Whang, K. [1985] "*Query Optimization in Office By Example*", IBM Research Report RC 11571, diciembre de 1985.

Whang, K., Malhotra, A., Sockut, G. y Burns, L. [1990] "Supporting Universal Quantification in a Two-Dimensional Database Query Language", en *ICDE* [1990].

Whang, K. y Navathe, S. [1987] "An Extended Disjunctive Normal Form Approach for Processing Recursive Logic Queries in Loosely Coupled Environments", en *VLDB* [1987].

Whang, K. y Navathe, S. [1992] "Integrating Expert Systems with Database Management Systems—an Extended Disjunctive Normal Form Approach", **Information Sciences**, 64, marzo de 1992.

Whang, K., Wiederhold, G. y Sagalowicz, D. [1982] "Physical Design of Network Model Databases Using the Property of Separability", en *VLDB* [1982].

Widom, J., "Research Problems in Data Warehousing", *CIKM,* noviembre de 1995.

Widom, J. y Ceri, S. [1996] **Active Database Systems**, Morgan Kaufmann, 1996.

Widom, J. y Finkelstein, S. [1990] "Set Oriented Production Rules in Relational Database Systems", en *SIGMOD* [1990].

Wiederhold, G. [1983] **Database Design**, 2ª ed., McGraw-Hill, 1983.

Wiederhold, G. [1984] "Knowledge and Database Management", **IEEE Software**, enero de 1984.

Wiederhold, G. [1995] "Digital Libraries, Value, and Productivity", **CACM**, abril de 1995.

Wiederhold, G., Beetem, A. y Short, G. [1982] "A Database Approach to Communication in VLSI Design", **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, 1:2, abril de 1982.

Wiederhold, G. y Elmasri, R. [1979] "The Structural Model for Database Design", en la Conferencia ER [1979].

Wilkinson, K., Lyngbaek, P. y Hasan, W. [1990] "The IRIS Architecture and Implementation", **TKDE**, 2:1, marzo de 1990.

Willshire, M. [1991] "How Spacey Can They Get? Space Overhead for Storage and Indexing with Object-Oriented Databases", en *ICDE* [1991].

Wilson, B. y Navathe, S. [1986] "An Analytical Framework for Limited Redesign of Distributed Databases", *Proc. Sixth Advanced Database Symposium,* Tokio, agosto de 1986.

Wiorkowski, G. y Kull, D. [1992] **DB2-Design and Development Guide**, 3ª ed., Addison-Wesley, 1992.

Wirth, N. [1972] **Algorithms + Data Structures = Programs**, Prentice-Hall, 1972.

Wolfson, O. Chamberlain, S., Kalpakis, K. y Yesha, Y. [2001] "Modeling Moving Objects for Location Based Services", NSF Workshop on Infrastructure for Mobile and Wireless Systems, en **LNCS 2538**, págs. 46–58.

Wong, E. [1983] "Dynamic Rematerialization-Processing Distributed Queries Using Redundant Data", **TSE**, 9:3, mayo de 1983.

Wong, E. y Youssefi, K. [1976] "Decomposition—A Strategy for Query Processing", **TODS**, 1:3, septiembre de 1976.

Wong, H. [1984] "Micro and Macro Statistical/ Scientific Database Management", en *ICDE* [1984].

Wood, J. y Silver, D. [1989] **Joint Application Design: How to Design Quality Systems in 40% Less Time**, Wiley, 1989.

Wu, X. y Ichikawa, T. [1992] "KDA: A Knowledgebased Database Assistant with a Query Guiding Facility", **TKDE** 4:5, octubre de 1992.

Worboys, M. y Duckham, M. [2004] *GIS – A Computing Perspective,* 2ª ed., CRC Press, 2004.

Xie, W. [2005], "*Supporting Distributed Transaction Processing Over Mobile and Heterogeneous Platforms*", Ph.D. Dissertation, Georgia Institute of Technology, 2005.

Xie, W., Navathe, S. y Prasad, S. [2003], "Supporting QoSAware Transaction in the Middleware for a System of Mobile Devices (SyD)", en *Proc. 1er International Workshop on Mobile Distributed Computing in ICDCS '03*, Providence, RI, mayo de 2003. XML (2005) : *http://www.w3.org/XML/.*

Yannakakis, Y. [1984] "Serializability by Locking", **JACM**, 31:2, 1984.

Yao, S. [1979] "Optimization of Query Evaluation Algorithms", **TODS**, 4:2, junio de 1979.

Yao, S., ed. [1985] **Principles of Database Design**, Vol. 1: **Logical Organizations**, Prentice-Hall, 1985.

Yee, W., Donahoo, M. y Navathe, S. [2001] "Scaling Replica Maintenance in Intermittently Synchronized Databases", en *CIKM* [2001].

Yee, W., Navathe, S., Omiecinski, E. y Jermaine, C. y otros [2002] Efficient Data Allocation over Multiple Channels at Broadcast Servers, *IEEE Transactions on Computers*, Special Issue on Mobility and Databases, 51:10, 2002.

Yoshitaka, A. y Ichikawa, K. [1999] "A Survey on Content-Based Retrieval for Multimedia Databases", **TKDE**, 11:1, enero de 1999.

Youssefi, K. y Wong, E. [1979] "Query Processing in a Relational Database Management System", en *VLDB* [1979].

Zadeh, L. [1983] "The Role of Fuzzy Logic in the Management of Uncertainty in Expert Systems", **Fuzzy Sets and Systems**, 11, North-Holland, 1983.

Zaniolo, C. [1976] "*Analysis and Design of Relational Schemata for Database Systems*", Ph.D. dissertation, Universidad de California, Los Ángeles, 1976.

Zaniolo, C. [1988] "*Design and Implementation of a Logic Based Language for Data Intensive Applications*", MCC Technical Report #ACA-ST-199-88, junio de 1988.

Zaniolo, C. y otros [1986] "Object-Oriented Database Systems and Knowledge Systems", en *EDS* [1984].

Zaniolo, C. y otros [1997] **Advanced Database Systems**, Morgan Kaufmann, 1997.

Zave, P. [1997] "Classification of Research Efforts in Requirements Engineering", **ACM Computing Surveys**, 29:4, diciembre de 1997.

Zeiler, Michael. [1999] **Modeling Our World—The ESRI Guide to Geodatabase Design**, 1999.

Zhang, T., Ramakrishnan, R. y Livny, M. [1996] "Birch: An Efficient Data Clustering Method for Very Large Databases", en *SIGMOD* [1996].

Zhou, X. y Pu, P. [2002] Visual and Multimedia Information Management, *Proc. Sixth Working Conf. on Visual Database Systems*, Zhou, X., Pu, P. (eds.), Brisbane, Australia, *IFIP Conference Proceedings 216*, Kluwer, 2002.

Zicari, R. [1991] "A Framework for Schema Updates in an Object-Oriented Database System", en *ICDE* [1991].

Zloof, M. [1975] "Query by Example", *NCC, AFIPS,* 44, 1975.

Zloof,M. [1982] "Office By Example: A Business Language That Unifies Data, Word Processing, and Electronic Mail", **IBM Systems Journal**, 21:3, 1982.

Zobel, J., Moffat, A. y Sacks-Davis, R. [1992] "An Efficient Indexing Technique for Full-Text Database Systems", en *VLDB* [1992].

Zook, W. y otros [1977] **INGRES Reference Manual**, Departamento de EECS, Universidad de California en Berkeley, 1977.

Zvieli, A. [1986] "A Fuzzy Relational Calculus", en *EDS* [1986].

# Índice


1 FN (primera forma normal), 128, 300–303, 330
2 FN (segunda forma normal)
  conservación de la dependencia, 319
  definición, 304
  panorámica de, 305–308
  probar con 3NF, 308
2PL, 551
3 FN (tercera forma normal)
  definiciones generales de, 305
  descomposición de la conservación de la dependencia en, 323–328
  descomposición de relación en, 318–319
  panorámica de, 304–308
  resumen de, 331
4 FN (cuarta forma normal), 332–337
5 FN (quinta forma normal), 337–338

**A**

abajo arriba, metodología de diseño, 281, 317–318, 355
abstracción, 108
  funcionamiento, 12
  identificación como, 108
abstractos, tipos de datos
  concepto de, 599
  encapsulación relacionada con, 604–606
acceso
  bases de datos biológicas, 890
  control de acceso basado en el roles, 692–693
  control de acceso XML, 693
  control, definición, 683
  DAC (control de acceso discrecional), 685–689
  estructura auxiliar, 392
  MAC (control de acceso obligatorio), 689–692
  método, 405
  modelo de matriz, 686
  modo, SQL, 538
  no autorizado, restringido, 17
  políticas de comercio electrónico, 693–694
  protección, 684–685
  rutas de acceso, definición, 29–30
  rutas de acceso, diseño físico de bases de datos, 364–365
  rutas de acceso, tipos de, 45
acción de activación referencial
  definición, 211
  en el comando DELETE, 237
  en el comando UPDATE, 237–238
ACEDB, bases de datos, 895
ACID, propiedades, 526–527
activación, comando de, 711
actores en escena
  diagramas de caso de uso, 368
  tipos de, 13–15
actualizaciones
  anomalías, 286–288
  bases de datos bitemporales, 721–723
  conversión de bloqueo, 548
  de las vistas, 241–243
  definición, 137
  descomponer, 766–768
  designar bases de datos, 595
  diferidas, 572, 577–580
  estados de bases de datos y, 29–30
  estructuras de almacenamiento, 17–18
  fichero, 403–405
  incrementales, 241
  inmediatas en recuperación, 572, 582–583
  operaciones encapsuladas, 606–607
  panorámica de las, 139–140
  pérdida de las, 521–522
  preventiva, 719
  problema de lectura irrepetible, 522
  problema de suma incorrecta, 522
  QBE, 913–915
  recuperación basada en las, 577–583
  retroactivas, 719
  rol de las, 8
  simultáneas, 719
  SQL dinámico, 259–260
  temporales, 522
  tiempo válido para, 718–719
  transacción, 359–361
adición (+), operador de, 221
Administración de la relación con el cliente (CRM), 23
Administración de recursos de información (IRM), 346–348
Administrador de bases de datos. *Véase* DBA (administrador de bases de datos)
ADN (ácido desoxirribonucleico), 890–891
agregación
  cualificada, 75
  modelo EER, 109–110

relaciones UML como, 74
temporal, 727
agrupamiento
minería de datos, 827, 839–841
objetos complejos, 614
OQL, 644–645
QBE, 913–915
álgebra relacional, 145–187
CARTESIAN PRODUCT, operación, 152–155
convertir consultas SQL en, 465–466
CROSS PRODUCT, operación, 152–155
DIVISION, operación, 159–160
ejemplos de consulta, 167–169
EQUIJOIN, operación, 156–159
expresión, 145
funciones agregadas, 163–164
INTERSECTION, operación, 151–152
JOIN, operación, 155–159
MINUS, operación, 151–152
NATURAL JOIN, operación, 156–159
notación Datalog usando, 739
notación de árbol de consulta, 160–163
operaciones de cierre recursivo, 164–165
OUTER JOIN, operación, 165–166
OUTER UNION, operación, 167
panorámica, 145–146
PROJECT, operación, 146–149
proyección generalizada, operación, 162–163
RENAME, operación, 149–151
repaso/ejercicios, 180–186
SELECT, operación, 146–148
UNION, operación, 151–152, 158–162
algoritmos, 317–344
búsqueda binaria, 408
búsqueda de dispersión lineal, 417
cuestiones/ejercicios, 341–344
de muestreo, 829–830
de ordenación, 466–467
de síntesis, 330
dependencias de concatenación y 5 FN, 337–338
dependencias de inclusión, 338–339
dependencias de plantilla, 339–340
dependencias funcionales basadas en la aritmética, 339
dependencias multivalor y 4 FN, 335–337
dependencias multivalor, 332–334
determinar X bajo F, 296
diseño de esquema de base de datos relacional, 323
dispersión interna, 410–412
ER-a-relacional, 189
forma normal de dominio de clave, 340–341
genéticos (GAs), 843
panorámica, 317–318
propiedades de las descomposiciones relacionales, 318–323
resumen, 341
alias, definición, 216–217
ALL, palabra clave, 219, 224
ALLFusion Component Modeler, 345
almacén de metadatos, 860
almacén distribuido, 860
almacén para multimedia, 873
almacenamiento
administrador de datos, 36–38
atributos almacenados, 57
comando reorganizar para, 405
conectado a la red (NAS), 424
control de la redundancia, 15–16
definición, 39
documento XML, 813
en la dispersión externa, 412–414
en la fase de diseño físico, 364–365
en las bases de datos multimedia, 873
en masa, 391
en registros desordenados, 406
en registros ordenados, 408
medio de, 390
minimizar el espacio derrochado, 16, 284–286
monitorización de estadísticas para el, 507
no volátil, 392
*offline*, 392
*online*, 392, 423–424
persistente, 17, 272–274, 392
principal, 390–391
procedimientos almacenados, 19, 272–274
procesamiento eficaz de consultas, 17–18, 490
rol del, 8
SANs (redes de área de almacenamiento), 423–424
SDL, 34
secundario, 390–391, 393–398
sistemas de tipos extendidos, 673
terciario, 391–392
tipos de ruta de acceso para, 45
utilidad de reorganización, 39
valor NULL en el, 288–289
volátil, 392
almacenes de datos
a nivel de empresa, 854
características, 852–854
crear, 858–860
definición, 852
funcionalidad típica de los, 861–861
minería de datos frente a, 824
modelado de datos para los, 854–858
problemas en, 862–863
repaso/ejercicios, 864
terminología, 851–852
vistas frente a, 861–862
virtuales, 854

ALTER, comando, 212–213
analista de sistemas, 15
analizador de consultas, 463–464
anclas de bloque, 431
AND, operador, 170, 224
animaciones, datos multimedia, 873
ANSI/SPARC, arquitectura. *Véase* arquitectura de tres esquemas
ANY, operador, 224
API. *Véase* interfaz de programación de aplicaciones
aplicaciones
  científicas, 22
  de marketing, minería de datos, 844
  de presentación para multimedia, 873
  espaciales, 22
  financieras, minería de datos, 844
  genéticas, tecnología de bases de datos, 890
  GIS, 886–887
  minería de datos, 844
  para el cuidado de la salud, minería de datos, 844
  series cronológicas, 22
aplicaciones de bases de datos
  activas, 714–715
  ciclo vital de las, 349
  científicas, 22
  definición, 51, 252
  ejemplo de, 54–55
  multimedia, 876
  tradicionales, 3
apóstrofe (‘), 221
aprendizaje sin supervisión
  agrupamiento, 839
  redes neuronales, 843
aprendizaje supervisado, 836, 843
Apriori, algoritmo, 828–829
árbol
  de búsqueda, 442–444
  de consulta equivalente, 481
  de consulta final, 483–489
  de consulta inicial, 483–488
  de enlace B, 565
  panorámica de, 442
  repaso/ejercicios, 457–461
árboles B, 444–446, 453
  definición, 392, 418
  estructura de datos de árbol de los, 442
  panorámica de los, 444–446
  repaso/ejercicios, 457–461
  variaciones de los, 453
árboles B+, 446–453
  búsquedas con, 449
  estructura de datos de los, 442
  inserción de registros en los, 449–451
  operación SELECT, 468, 491
  panorámica de los, 446–449
  repaso/ejercicios, 457–461
  variaciones de los, 452
árboles de consulta
  convertir en planes de ejecución de consultas, 488
  definición, 463
  notación para los, 160–162
  optimización heurística de, 483–488
  panorámica de los, 481–483
ArcINFO, software, 886–887
argumentos, notación Prolog/Datalog, 730–733
ARIES, algoritmo de recuperación, 584–587
Armstrong, reglas de inferencia de,
  definición, 296
  para las dependencias multivalor, 334
arquitectura
  centralizada, 40
  cliente/servidor de dos capas, 42–43
  cliente-servidor de tres niveles, 770–772
  de almacén federado, 860
  de disco compartida, 750–757
  de memoria compartida, 750–757
  de tres esquemas, 31–32
arrays
  constructores de tipo, 601, 655
  PHP, 792–793
arriba abajo, síntesis conceptual, 99
AS, cualificador, 221, 228–229
ASC, palabra clave, 222
aserciones
  aserción declarativas, 238–239
  definición, 136
  restricciones como, 238–239
asignación
  continua, 402
  enlazada, 402
ASN.1, 893
asociaciones
  agregación y, 74
  bidireccionales, 74
  binarias, 74
  cualificadas, 75
  modelo EER, 109–110
  panorámica de las, 74
  reflexivas, 74
  terminología UML para las, 74
  unidireccionales, 74
aspecto dinámico, 28
asterisco (*)
  consultas SQL, 221
  notación UML, 74
  operaciones OUTER JOIN, 167
  SQL, funciones agregadas, 230–231
  SQL, recuperar variables de atributo de tuplas, 217

atomicidad, propiedad
como propiedad ACID, 526
definición, 13
transacciones multibase de datos, 588
transacciones SQL, 538
atributos
agrupamiento, 231–234
agrupamiento para formar esquema de relación, 283–286
almacenados, 57
clave, 58–59
complejos, 57
conjuntos de valores de, 59–60
conservación, 319
de clave múltiple, 58–59
de imagen, 198
de tipos de relación, 62, 66
definición, 28
denominación, 70
derivados, 57
diseño de indexación, 505–506
enlace, 74
específicos, 93
locales, 93
monovalor, 56
no aplicable, 222
no primo, 300
notación para los diagramas ER, 71–72
ocultos de objetos, 607
primos, 301, 306
relaciones y, 63
renombrado, 149–151, 228–229
simples, 56, 675
solicitados, 170
tipos de, 56–57
valores NULL, 57
atributos atómicos
definición, 56, 124
en las tuplas, 128
primera forma normal y, 300–303
atributos compuestos
como atributos clave, 58
definición, 56
desaprobación, 301–303
esquema XML, 944–945
mapeado de esquema EER a esquema ODB, 648
atributos multivalor
definición, 56–57
independientes, 332
mapeado, 194, 648
auditoría, base de datos, 685
autonomía, en DBMS federadas, 753, 761
autorización
AUTHORIZATION, cláusula, 205, 255
control de acceso a comercio electrónico, 693
restricción de acceso no autorizado, 17
AVERAGE, operador, 478
AVG, función, 230–231
axiomas, 735

## B

barra doble (//), 819
barra inclinada (/), 819
base de datos universal (UDB), 32
bases de datos
auditoría, 685
back-ends, 23
bitemporales, 718, 721–722
cargadas, 30
colección de elementos de datos con nombre, 519
confidencialidad, 682
copia de seguridad, 588
de tiempo válido, 718–719
diseñadores, 14
diseño. *Véase* diseño físico de bases de datos
disponibilidad, 682
espaciales, 706, 727–729
esquema. *Véase* esquemas
estados, 29–30, 526
estructura, 28
grandes, 346, 391
integridad, 682
lógicas, 706
personales, 347–348
programación, 252. *Véase también* SQL (Lenguaje de consulta estructurado)
refinación. *Véase* refinación
rellenar, 30
servidores, 38, 253
supervivencia, 700
terminología, 660–661
bases de datos activas
aplicaciones para, 3, 714–715
definición, 706
modelo generalizado para, 707–710
panorámica de, 707
problemas de diseño e implementación, 710–712
reglas activas a nivel de sentencia en STARBURST, 712–714
reglas activas para, 19
SQL, creación de *triggers* con, 243
*triggers* en SQL-99, 715
bases de datos de objetos, 597–619
C++, vinculación con, 645–647
conceptos de orientación a objetos, 599–600
configuración, 616
constructores de tipos, 604
diseño conceptual, 647–649

diseño de base de datos relacional frente a, 647–648
encapsulación de operaciones, 606–607
encapsulación de persistencia, 607–610
estándares. *Véase* ODMG (Grupo de modelos de datos de objetos)
estructura de objetos, 601–604
herencia múltiple, 616
herencia selectiva, 616
identidad de objeto, 601
jerarquías de tipo y herencia, 610–612
lenguaje de consulta de objetos. *Véase* OQL (lenguaje de consulta de objetos)
lenguaje de definición de objetos, 633–638
mapeado de esquema EER a, 648–649
objetos complejos, 613–615
polimorfismo, 615
repaso/ejercicios, 617–619, 649–651
restricciones en extensiones, 612–613
versiones, 616–617
base de datos del genoma (GDB), 893–894
características de los datos biológicos, 890–892
ciencia biológica y genética, 889–890
proyecto del genoma humano y bases de datos biológicas, 892–896
bases de datos móviles, 866–871
arquitectura, 866–867
características, 867–870
problemas de administración de datos, 870–871
bases de datos multimedia, 872–878
definición, 706
naturaleza de las, 873
panorámica de, 729–730
problemas de administración, 874
problemas de investigación abierta, 874–877
bases de datos temporales
calendarios, 717–718
construcciones de consulta, 725–727
datos de series de tiempo, 727
definición, 705
dimensiones de tiempo, 717
panorámica de, 715–717
representación del tiempo, 717–718
versionado de atributos en, 723–725
versionado de tuplas en, 718–720
bases de datos, tecnologías emergentes, 865–900
bases de datos del genoma, 889–896
bases de datos móviles, 866–872
bases de datos multimedia, 872–877
Sistemas de información geográfica, 878–889
BCNF. *Véase* Boyce-Codd, foma normal (BCNF)
Bell-LaPadula, modelo, 690
BETWEEN, operador, 221
BFILE, 672
BFIM (imagen antes), 529, 573
biblioteca de funciones, 253, 264, 268
bioinformática, 890
BIRCH, algoritmo, 841
bit, 393
bit pin-unpin, 573
bit sucio, 573
bitmap, indexación, 856–857
BLOB (objeto binario grande), 208, 400, 613, 659, 672
bloquear_elemento(X), operación
bloquear conversión, 548
bloqueo en dos fases, 551
ordenar marca de tiempo, 556–557
panorámica, 546–547
bloqueo
binario, 545–547
bloqueo en dos fases, 549–551
de certificación, 558–559
compartido/exclusivo, 547–549
conversión de, 548
definición, 545
detección del interbloqueo, 554
granularidad múltiple, 560–663
inanición, 554
modo múltiple, 547
protocolo de prevención de interbloqueos, 552
tiempo limitado, 554
bloqueo en dos fases (2PL)
actualizaciones diferidas, 579
detección de interbloqueos, 554
inanición, 554
multiversión, 558–559
prevención del interbloqueo, 552–554
tiempo limitado, 554
tipos de bloqueos y tablas de bloqueo del sistema, 546–548
bloques
asignación de archivo, 402
compartidos/exclusivos, 547–548
definición, 393
factor de bloqueo de registro, 402
tiempo de transferencia, 397
Bluetooth, red, 866
body, HTML, 786
bolsa de tuplas, 149, 213
booleano, tipo de datos, 208, 222
3 FN y, 308
4 FN y, 335
Boyce-Codd, forma normal (BCNF)
algoritmos de normalización y, 330
definición, 299
descomposición de concatenación no aditiva en, 323–327
descomposición y, 318
BPWin, 345, 381

BS (estaciones base), 866
búferes
  almacenamiento en caché de bloques de disco, 572
  bloque de disco, 572
  procesamiento de transacciones y, 518
búsquedas
  binarias, 408, 468, 491
  dispersión externa para ficheros de disco, 412–414
  en registros desordenados, 405
  operaciones de fichero para, 403–406
  técnicas de dispersión para, 409
búsquedas lineales
  definición, 403
  en registros desordenados, 405
  implementación de SELECT con, 468, 491
byte, 393

## C

C, lenguaje de programación
  desajuste de impedancia y, 253
  PHP, utilizando, 788
  SQL, incrustado y, 254–257
  SQL/CLI, utilizando, 266–268
C++, lenguaje de programación
  almacenamiento persistente y, 17
  estructura de base de datos y, 9–10
  SQL con, 204
  vinculación, 645–647
cabeza de lectura/escritura, 394
cadena de caracteres, tipos de datos, 207–208, 219
cadenas en PHP, 790–791
cálculo relacional
  cuantificador existencial en el, 171–173
  cuantificador universal en el, 171, 175–176
  de dominio, 177–179
  de tupla, 169–171, 204
  definición, 146, 169
  expresiones en tuplas, 171
  expresiones seguras en el, 176
  fórmulas en tuplas, 171
  panorámica, 145–146
  relaciones de rango, 170–171
  repaso/ejercicios, 180–187
  variables de tupla, 170–171
calendarios, 717, 727
CALL, sentencia, 273
campos
  agrupados, 433
  claves, 407
  conexión, 418
  de indexación, 430, 439
  en la definición de un registro, 400
  en registros ordenados, 407
  opcionales, 400
  repetidos, 400
  tipos de, 400
canales
  de almacenamiento, 697
  de temporización, 697
  ocultos, 696–697
canónicas, formas, 297–298
capa de aplicación, arquitectura cliente-servidor de 3 niveles, 43–44, 771
capa de lógica de negocio, arquitectura cliente-servidor de 3 niveles, 43–44, 771
capa de presentación en la arquitectura cliente-servidor de 3 niveles, 43–44, 771
capa de servicios de administración de datos, arquitectura de tres capas, 43–44
capas intermedias, 43–44
capas raster, GIS, 879
caracteres separadores, 401
cartesiano, producto, 126, 152–155, 230
cartuchos, 671, 705
CASE (ingeniería de software asistida por computador)
  DDL, utilizando, 364
  diseño de bases de datos, utilizando, 39
  notación, 901–903
  panorámica de, 378–381
  recopilación de requisitos y análisis, utilizando, 353
catálogo
  DBMS, 9
  SQL, 205
categoría
  cuándo usar, 104
  definición, 105
  mapeado, 199, 648
  modelado mediante, 100–101
categoría parcial, 101
categoría total, 101
CD-ROM, 391, 398
celdas, movilidad geográfica, 866
CHECK, cláusula, 209–210, 212
ciclo vital, 348–349
ciencias biológicas, 889–890
cierre recursivo, 164–165
cifrado de clave pública, 697–699
cilindros, disco, 393–394
cinta lineal digital (DLT), 398
cinta lineal superdigital (SDLT), 398
cinta magnética
  definición, 391
  panorámica de, 393–398
  uso en el almacenamiento de bases de datos, 391
clase base, 106
clase, propiedades, 108
  dependencias de inclusión, 338–339
  en el modelo de objeto ODMG, 631

especificar comportamiento de objetos, 606–607
instanciación, 108
restricciones de relación, 338–339
seguridad, 690
subclases, 90–91
superclases, 90–91
clase hoja, 106
clasificación
atributo de, 690
de los sistemas de administración de bases de datos, 44–46
definición, 108
indexación de imágenes, 875
minería de datos, 826–827, 836–839
proceso de, 108
cláusulas, 147
AUTHORIZATION, 205, 255
CHECK, 209, 212
clave externa, 211
Datalog, notación, 733
DEFAULT, 209–210
FOR UPDATE OF, 258–259
GROUP BY/HAVING, 231–234
INTO, 257, 263
ordenación de las cláusulas SQL, 234–235
ORDER BY, 222
SELECT/FROM/WHERE, 214, 229–230
SET, 237–238
UNIQUE, 226–228
WITH CHECK OPTION, 243
clausura, 293
clave. *Véase también* clave externa
aparente, relaciones multinivel, 690
candidata, 131, 292, 300
cifrado de clave pública, 697–699
compuesta, 59, 454
definición, 300
esquema XML, 944
parcial, 68, 302
privada, 698
secundaria, 434
clave externa (*foreign key*)
cláusulas, 211
definición, 133–134
en el diseño de base de datos relacional, 647
errores, 207
método de mapeo, 192–194
notación en la herramienta Rational Rose, 375–376
restricción de la dependencia de inclusión y, 338–339
valores NULL en la, 328
clave, restricciones de
definición, 58
en el modelo de datos relacionales, 130–131, 133
en los tipos de entidad débil, 67–68
especificar en SQL, 210–211
violación en las operaciones, 137–139
claves principales
definición, 131, 430
en la normalización, 301–305
en los esquemas de relación, 300
especificación de restricciones en SQL, 209–211
restricción de la integridad de entidad y, 133–134
cliente, definición, 41
cliente/servidor, arquitectura
panorámica de, 40–43
uso de NAS, 424
CLOB (objeto grande de caracteres), 613, 659, 672
CLOSE CURSOR, comando, 258
close, comando, 404
coalescencia, regla, 334
COBOL, 9–10, 46, 204
Codd, Ted, 123, 299–300
código de programa procedimental, 19
colección
de basura, 584
literales, 623
objetos, 622, 624–629
operadores, sintaxis OQL, 642–645
persistente, 609
transitoria, 613
tipo, 602, 664
colisión, 412
columnas
añadir/eliminar/cambiar, 212–213
enlazadas, 267
comando de escritura, discos, 394
comandos. *Véase también* funciones; operaciones
ALTER, 212–213
CLOSE CURSOR, 258
close, 404
CONNECT TO, 255
CREATE ROLE, 692
CREATE SCHEMA, 686
CREATE TABLE, 205–207, 210–211
CREATE VIEW, 240–241
CURSOR, 258
DELETE, 237, 709
DESTROY ROLE, 692
DISCONNECT, 256
DROP VIEW, 241
DROP, 212
evolución del esquema, 212–213
EXECUTE, 260
FETCH, 314
ficheros, 404–405
incrustar SQL, 252
incrustar SQL, en Java, 260–262
INSERT, 235–237

PREPARE, 260
PROCESS RULES, 711
REVOKE, 687
SQL, 243–245
UPDATE, 237–238
comercio electrónico (e-commerce), 22, 693–694
comillas dobles (“”), 791
COMMIT_TRANSACTION, operación, 524
compatibilidad, 17
compatibilidad parcial, 167
componente heurístico, 380
computador cliente, 38
concatenación de bucle anidado (fuerza bruta)
ejemplo de función de coste para JOIN, 493
factor de espacio en búfer, 474
panorámica de, 471–474
usando concatenación externa, 479
concatenaciones
algoritmo de concatenación de dispersión, 475
algoritmo de concatenación de dispersión híbrida, 477
anidadas, 230
atributos, 156, 328
consultas de relación múltiple y, 495
de dispersión, 471–474, 475–476
de dos vías, 470
de intersección temporal, 722
dependencias, 337–338
ejemplo de función de coste para, 493–495
espaciales, 727
espacio de búfer y, 474
especificar en QBE, 913
factor de selección, 474
implementación, 470–474
intersección temporal, 722
multivía, 470
refinación de consulta y, 512
reglas heurísticas en, 480
selectividad, 493
concatenaciones de mezcla
coste y, 494
factor de selección y, 474
operaciones SET usando, 477
panorámica de, 471
concatenaciones de bucle simple
concatenaciones externas, 479
factor de selección de concatenación y, 474
función de coste y, 493
panorámica de, 471
concatenaciones internas
definición, 157
operación OUTER JOIN y, 166
tuplas colgantes y, 328–330
uso en SQL, 229–230
concatenar (| |), operador, 208, 221
condición de selección simple, 403
condición disyuntiva, 470
condiciones
cálculo relacional de dominio, 177
conclusión de plantilla, 339
de calificador, 819
definición, 171
ejemplo, 712–714
especificar restricciones, 238–239
evaluación, 711
excepciones, 261
JOIN. *Véase* concatenaciones
marcadores en diagramas de secuencia, 369
operación sobre ficheros, 403–404
reglas de activación, 707
Conectividad de base de datos abierta. *Véase* ODBC (Conectividad de base de datos abierta)
Conectividad de bases de datos de Java. *Véase* JDBC (Conectividad de bases de datos de Java)
conexiones
entre servidor cliente/bases de datos, 254
lógicas, 171
confianza, minería de datos, 827
confidencialidad, 682
confirmación automática, 261
confirmación en dos fases, 588, 768
conflictos
de identificación entre esquemas, 356–357
de operaciones en la planificación, 527
de planificaciones equivalentes, 532
de planificaciones serializables, 532–536
de tipo, 357
conjunto de entidades, 58–59
conjunto de referencias, 604
conjuntos
de dependencias funcionales, 296, 297–298
de lectura, transacciones, 520
de relaciones, 61–62
de valores, 59–60, 228–229
explícitos en SQL, 228–229
CONNECT TO, comando, 255
conocimiento inductivo, 826
conservación de la dependencia, 299, 317, 319, 323–328
constructor
de conjunto, 609
de objeto, operación, 607
de tipo atómico, 602–603
de tipo bolsa (*bag*), 602
de tipo lista, 602
de tipo tupla, 602
de tipos, 601, 604, 655
consultas, 463–501
álgebra relacional, 167–170, 465–466
almacenamiento para las, 17–18

anidadas, 223–225, 512
anidadas correlacionadas, 225–226, 466, 512
anidadas no correlacionadas, 466
árboles, 160–162
bases de datos biológicas, 892
bloques, 465
cálculo relacional de dominio, 177–179
cálculo relacional de tupla, 169–171
compiladas, 489
computación móvil, 871
con nombre, 642
cuantificador existencial, 171-173
cuantificador universal, 160, 171, 175–176
Datalog no recursivas, 739–742
de relación múltiples, 495
descomponer, 766
definición, 5
en bases de datos espaciales, 727
en bases de datos estadísticas, 694
en bases de datos multimedia, 727–730, 874
en bases de datos temporales, 725–727
en los sistemas de bases de datos distribuidos, 754, 762–768
en los sistemas relacionales de objetos, 787
externas, 223–225
FDBS, 761
grafo, 463, 480–481
heurísticas. *Véase* reglas heurísticas
interactivas, 36–37
interpretadas, 489
JOINS, 155, 470–475, 479, 493–495
lenguaje de, 34
modificar, 241–243
operaciones agregadas, 478
optimización basada en el coste, 489–498
optimizar, 37
OQL, 639–642
Oracle, 498
ordenación externa, 466
ordenar resultado de, 221
panorámica de, 463–465
PHP, 798–799
PROJECT, operación, 477
recursivas, 217
refinación, 507, 510–512
repaso/ejercicios, 499–501
rol de, 8
SELECT, 468–470, 491–493
semántica de la optimización, 499
SET, 477
SQL, 234–235, 259–260, 465–466
simples, OQL, 639
temporales, 858
XML, 819–821
CONTAINS, operador, 226
contexto predeterminado, 261
contexto, bases de datos biológicas, 890–892
contradicción, prueba por, 294
contraseñas, 17
control de acceso obligatorio, 689–693
control de acceso basado en roles (RBAC), 681, 692–693
control de acceso discrecional (DAC)
  basado en privilegios, 685–689
  control de acceso obligatorio y, 689, 692
  control de acceso basado en roles frente a, 692–693
  definición, 681
control de la concurrencia, 545–569
  actualizaciones diferidas en el sistema multiusuario, 578–580
  aislamiento, 526
  bases de datos distribuidas, 768–769
  bloqueo en dos fases. *Véase* bloqueo en dos fases (2PL)
  bloqueo por nivel de granularidad múltiple, 561–563
  bloqueos en los índices, uso, 563–565
  con marcas de tiempo, 555–557
  definición, 13
  fallo de transacción y, 522
  granularidad de los elementos de datos, 560–563
  multiversión, 557–559
  necesario para, 520
  optimista, 559–560
  paginación en la sombra, 583–584
  problema del registro fantasma y, 565–566
  problemas de eliminación, 565
  problemas de inserción, 565
  problemas de transacción interactiva, 566
  repaso/ejercicios, 566–568
  sistema monousuario frente a sistema multiusuario, 518
  validación, 559–560
control del flujo, 683, 696–697
controlador, 268–268
controlador de disco, 396
convenciones de denominación
  atributos en SQL, 228–229
  atributos, 216–217
  conflictos, 357
  convenciones de denominación de construcción de esquema, 70
  definición de nombre, 124
  EER, modelo convenciones de denominación, 111
  especificar persistencia de objetos, 607–610
  RENAME, operación, 149–150
  restricciones, 212
conversión
  bloqueo, 548
  coste de, 362

herramientas, 39
rutinas, 366
corchetes ([...]), 244–245
corchetes angulares (< ... >), 244–245
coste de aprendizaje, 362
COUNT, función, 230–231
COUNT, operador, 478, 643
CREATE ASSERTION, sentencia, 238–239
CREATE ROLE, comando, 692
CREATE RULE, sentencia, 712–714
CREATE SCHEMA, comando, 686
CREATE TABLE, comando, 205–207, 210–211
CREATE TRIGGER, sentencia, 238–239, 708
CREATE VIEW, comando, 240–241
CREATE, sentencia, 205
CRM (Administración de la relación con el cliente), 23
crop, función, 668
CROSS JOIN, operación, 230
CROSS PRODUCT, operación, 152–155
cuantificador existencial, 171–173
cuantificador universal, 171
definición, 160
fórmulas de cálculo relacional usando, 171
transformaciones, 174
cuarta forma normal (4 FN), 332–337
cubos, 412, 414–417
cuentas
de propietario, 686
de sistema, 684
de usuario, 684
superusuario, 684
CURSOR, comando, 258
cursores, 253, 258–259

## D

DAC. *Véase* control de acceso discrecional (DAC)
Data Blades (API), Interfaz de programación de aplicaciones, 662–663, 667–670
Datalog, programas
consultas no recursivas, 740–742
definición, 730
Horn, cláusulas, 733–735
interpretaciones de reglas, 735–737
notación, 733
operaciones relacionales, 739
Prolog/Datalog, 730–733
seguridad de los, 737–739
datos. *Véase también* granularidad de los elementos de datos
abstracción, 10–12, 28
acceso, 43–44
aislamiento, 10–12
almacenamiento. *Véase* almacenamiento
almacenes, 3
autodescriptivos, 784
blade, 705
cifrado de, 683
complejidad/volumen de, 363
contenido/estructura, diseño de, 350
datos de eventos válidos, 727
definición, 4
dependencias, 130
distribución de, 757
elementos, 6-7, 519
estructurados, 784–788
fragmentación, 754–759
GIS, 879
independencia, 32–33, 346
localización de, 753
minimizar la incoherencia de los, 16
modelos, 761
no estructurados, 784–788
normalización de los, 298–299
punteros, 444
registros. *Véase* registros
relaciones. *Véase* relaciones
replicación, 757–759
semiestructurados, 784–788
virtuales, 12
DBA (administrador de bases de datos)
almacenamiento de datos y, 862–863
definición, 13
esquemas de autorización, 205, 686
interfaces para, 36
papel organizativo de, 346
seguridad y, 684
DBMS (sistema de administración de bases de datos). *Véase también* sistemas de bases de datos
arquitectura de tres esquemas, 31–33
caché, 572
categorías de modelos de datos, 28–32
centralizado, 45
clasificación, 44–46
de propósito especial, 45
de propósito general, 45
definición, 5–6
desventajas de los, 23–24
diseño/implementación, 350–352
distribuido (DDBMS), 44, 750
entorno del sistema, 36–44
factores al elegir, 362–363
GIS, 889
heredado, 44, 124, 660
heterógeneo, 44, 761–762, 774
independencia de los datos, 32–33
interfaces, 34–36
lenguajes, 33–35
monousuario, 44, 518, 578–579, 582

multiusuario, 518, 579–580
relacionales. *Véase* modelo de datos relacional
SQL, programación. *Véase* SQL (Lenguaje de consulta estructurado)
tendencias actuales en, 660–661
variables de comunicación, 256
DBMS de objetos relacionales (ORDBMS)
definición, 653
ejemplo extendido de. *Véase* Informix Universal Server
implementación de sistema de tipos extendido, 673
modelo relacional anidado, 674–676
panorámica de, 653
repaso, 676
SQL. *Véase* SQL (Lenguaje de consulta estructurado)
versión Oracle 8.X, 671–673
DDL (lenguaje de definición de datos)
definición, 33
especificación, 364
para los RDBMS, 133
restricciones, 134
SQL como, 204
declaraciones, XML, 805
declare, sección, 255
define, palabra clave, OQL, 642
definición parcial, 107
degradaciones en la conversión de bloque, 548
DELETE, comando, 709, 712–714
dependencia parcial, 304, 305–306
dependencia transitiva, 304–305
dependencias
algoritmos y. *Véase* algoritmos
concatenación, 337–338
de existencia, 65
de inclusión, 338–339
de plantilla, 339–340
dependencias funcionales, 281–316
Boyce-Codd, forma normal, 308–311
conjuntos de, 296–298
definición, 291–292
definición claves/atributos, 300
definiciones generales de las formas normales segunda/tercera, 305–308
generación de tuplas falsas, 289–290
información anómala/redundante en tuplas, 286–288
normalización de la relación y, 298–299
primera/segunda/tercera/cuarta, formas normales, 300–308
RDBMS, uso, 130
reglas de inferencia para las, 293–296
repaso/ejercicios, 311–316
restricciones, 137
resumen de las directrices de diseño, 290–291
valores NULL en las tuplas, 288–289
dependencias multivalor (MVD), 332–337
cuarta forma normal y, 335–337
definición formal/reglas para, 332–334
dependencias de concatenación y, 337–338
en los RDBMSs, 130
panorámica, 332
derechos de propiedad intelectual, 700
DES (Estándar de cifrado de datos), 697
desajuste de impedancia, problema, 17, 253
desbloquear(X), operación, 547
desbloquear_elemento(X), operación, 546–547
DESC, palabra clave, 222
descifrado
cifrado de clave pública, 698
uso en arquitectura cliente/servidor, 44
descomposición multivía, 337
descomposición binaria, 323
descomposiciones relacionales
algoritmos para crear, 323–331
consulta y actualización, 766–768
propiedades de la, 318–323
Descubrimiento de conocimiento en bases de datos (KDD), 824–826
DESHACER (ESCRIBIR_OP), procedimiento, 582
Designer 2000, 345
desnormalización, 300, 506–507,
DESTROY ROLE, comando, 692
Developer 2000, 347, 361
DFD (diagrama de flujo de datos), 53, 352–353
diagramas
de actividad, 371
de caso de uso, 368
de clase, 51, 72–75, 105–107, 367
de colaboración, 370
de componentes, 368
de flujo de datos (DFD), 53
de implantación, 368
de secuencia, 53, 75, 369–370
de transacción del estado, 371
estructuración de requisitos, uso, 352–353
herramientas CASE, 379
modelado de bases de datos, tipos de, 51
notación de dependencia funcional, 293
restricción de integridad referencial, 134
UML. *Véase también* UML (Lenguaje de modelado universal)
diccionario, definición, 112
Digital Rdb, 244
Dirección de bloque lógica (LBA), 394
direccionamiento abierto, 412
directorio
actual, 583
caché, 572
sombra, 583–584

Disco de vídeo digital (DVD), 391, 398
DISCONNECT, comando, 256
discos, 389–428
acceso usando RAID, 418–423
almacenamiento conectado a la red, 424
bloques, 398–400, 572
colocar registros de fichero en, 400–403
cuestiones/ejercicios, 425–428
de cabeza fija, 396
de cabeza móvil, 396
de doble cara, 393
de una cara, 393
dispositivos de almacenamiento de cinta magnética, 397–398
dispositivos de almacenamiento en memoria, 390
dispositivos de disco hardware, 393–397
fallo, 522
fichero *heap* (desordenado), 405–406
ficheros de registros ordenados, 406–409
ficheros ordenados, 406–409
operaciones sobre ficheros, 403–406
organizaciones principales de ficheros, 417–418
panorámica, 389–390
parámetros, 905–907
redes de área de almacenamiento, 423–424
resumen, 424–425
subficheros, 476
técnicas de dispersión, 409–417
WORM, 391
discos/cintas magnéticos
definición, 391
panorámica de, 393–398
uso en bases de datos, 391
discriminador, UML, 75
discriminación de atributos, 198
diseñadores de sistemas, 15
diseño arriba abajo, método, 281, 355–356
diseño conceptual de bases de datos.
con la herramienta Rational Rose, 374–378
definición, 28, 350–352
en arquitectura de tres capas, 31
en el modelo ER, 68–72
para esquemas, 353–359
para las transacciones, 359–361
diseño de base de datos relacional, 189–202
algoritmo de mapeado de ER-a-relacional para el, 189–196
diseño de bases de datos de objetos frente al, 647
factores de diseño físico en el, 503–507
mapeado EER, estructuras de modelo para relaciones, 196–200
repaso/ejercicios, 200–202
diseño de bases de datos. *Véase también* diseño práctico de bases de datos
algoritmos. *Véase* algoritmos
almacenes de datos, 858–859
bases de datos activas, 710–712
bases de datos biológicas, 890
bases de datos de objetos, 647–648
bases de datos distribuidas, 753–754
bases de datos multimedia, 874
bases de datos objeto-relación, 674
bases de datos relacionales, 189–202
computación móvil, 871
dependencias funcionales en el. *Véase* dependencias funcionales
directrices, 284–289
diseñadores de bases de datos, 15
fases, 8
herramientas automáticas para, 39, 374–382
metodología, 345
modelo de datos relacional. *Véase* modelo de datos relacional
rutas de acceso, 364–365
síntesis frente a análisis en, 282
diseño de esquema centralizado, 354–355
diseño físico de bases de datos
decisiones, 503–505
para la indexación, 505–506
para la normalización, 506–507
refinación, 507–512
diseño lógico. *Véase también* diseño conceptual de bases de datos
convertir en modelo de objeto, 377
descripción, 54
independencia de datos lógicos en el, 32–33
mapeo de modelo de datos, 364
teoría, 112
tercera fase del diseño de la base de datos, 8
diseño práctico de bases de datos, 345–385
ciclos vital de las bases de datos, 348–349
contexto organizativo de, 346–348
diseño conceptual, 353–361
diseño físico, 364–365
ejemplo de modelado, 372–374
elegir DBMS, 362–364
estándar UML para el, 366–372
herramientas automáticas para, 378–381
implementación, 349–352, 365–366
panorámica, 345–346
Rational Rose usada en, 374–378
refinación, 365–366
repaso/ejercicios, 382–384
requisitos de la fase de recopilación y análisis, 352
sistemas de información, 348–349
diseño sin pérdida, 320, 323
dispersión
estática, 414

extensible, 415
externa, 412–414
interna, 410–412
lineal, 416–417
múltiple, 412
partida, 454–455
disponibilidad, 752
dispositivo apuntador, 35, 402
dispositivos de acceso secuencial, 397–398
distancia, función de, 730
DISTINCT, palabra clave, 219–220
división (/), operador, 221
DIVISION, operación, 159–160
DKNF (dominios), 340–341
DLT (Cinta lineal digital), 398
DML (lenguaje de manipulación de datos), 34, 46, 204
de bajo nivel, 34
no procedimental, 34
procedimental, 34, 169
doble búfer, 398
doc, tipos de datos, 669
Documento de definición de datos (DTD), 806–807
documentos
almacenamiento en bases de datos multimedia, 729
como forma de almacenamiento de datos, 110
XML híbridos, 805
here, PHP, 791
documentos XML
almacenamiento, 813
convertir gráficos en árboles, 816
centrados en documentos, 805
centrados en los datos, 805
extraer, 814–816, 818–819
panorámica de los, 805–807
tipos de, 805–807
dólar ($), signo, 400, 790
DOM (Modelo de objeto de documento), 806
dominio
como conjunto de valor, 59–60
como elementos de esquema, 207
conflictos, 357
de conocimiento, 107–108
de discurso, 107
de movilidad geográfica, 866
especificar en UML, 73
estructurado, 73
modelo de datos relacional, 124
relacional, 177–179
restricciones, 130, 137–139, 209–210
DRAM (RAM dinámica), 390–391
DROP VIEW, comando, 241
DROP, comando, 212
DSS (sistemas de soporte a las decisiones), 852
DTD (Documento de definición de datos), 807–807
durabilidad, ACID, 526
duración de eventos, 717
DVD (Disco de vídeo digital), 391, 398

**E**

ECA (Evento-condición-acción), modelo, 707
EcoCyc, 894–895
e-commerce (comercio electrónico), 22, 693–694
economías de escala, 20
EEPROM, 391
EER (ER mejorado), modelo, 90–120
agregación, 109–110
asociación, 109–110
clasificación, 108
conceptos, 104–105
diseño, 104–105
ejemplo, 102–103
en la fase de diseño conceptual, 350–352, 354
especialización/generalización, 91–94, 98–99, 107–112
herramienta de diseño Rational Rose para, 375–376
identificación, 108
instanciación, 108
mapeado a esquema ODB, 648–649
modelado de tipos de entidad, 100–101
notaciones diagramáticas alternativas, 901–903
panorámica, 90–91
repaso/ejercicios, 112–120
restricciones, 94–98
semántica web, 110–112
subclases/superclases/herencia, 90–91
EIS (sistemas de información ejecutiva), 852
elemento
actual, 624
bloqueado para lectura, 547
bloqueado contra escritura, 547
bloqueado en exclusiva, 547
de datos con nombre, 519
XML, 805, 807, 943–944
eliminar duplicados, 149
Embarcadero Technologies, 345
encadenamiento, 412
encapsulación, 599, 604–610, 655, 657–658
encriptación
AES (Estándares de cifrado avanzado), 698
arquitectura cliente/servidor, 44
clave pública, 698–699
definición, 683, 698
DES (Estándar de cifrado de datos), 697
estándares XML para la, 693
firmas digitales, 699
panorámica de la, 697–699
END_TRANSACTION, operación, 524
enlaces hipermedia, 875

entidad, tipos de
asociar atributos a, 901–903
atributos clave de los, 58–59
conjuntos de entidades y, 58–60
conjuntos de valores y, 59–60
convenciones de denominación, 70
mapeo, 189–190
notación para los, 69–80
tipos de entidad débiles, 191–192
tipos de relación, 61–67
visualización ER, 901–903
entidad-atributo-relación, sistemas, 876
Entidad-relación, mapeado. *Véase* diseño de base de datos relacional
Entidad-relación, modelo. *Véase* ER (Entidad-relación), modelo
entornos
desarrollo de aplicaciones, 39
registros de entorno, 265–266
SQL, 205
entramados, 98–99, 198
Environmental Systems Research Institute, Inc. (ESRI), 886–887
EQUIJOIN, operación, 156–159, 229–230, 470
equivalencia de tipo, 658
equivalencia, 174, 296
ER (Entidad-relación), mapeado. *Véase* diseño de base de datos relacional
ER (Entidad-relación), modelo
atributos clave de los tipos de entidad, 58–59
binario frente a relaciones ternarias, 75–78
conjuntos de relación, 61–62
conjuntos de valores de atributos, 59–60
convenciones de denominación, 70
diagramas, 69–71
diseño de la base de datos EMPRESA, 60–61, 68
ejemplo, 54–55, 72–75
en la fase de diseño conceptual, 350–352
entidades y atributos, 55–57
grado de relación, 62
herramienta de diseño, 345, 375–376
instancias de relaciones, 62
notaciones alternativas, 72, 901–903
panorámica, 51–52
problemas de diseño, 52–54, 70–72
repaso/ejercicios, 79–87
restricciones en las relaciones ternarias, 78
resumen, 78–79
tipos de entidad débil, 67–68
tipos de entidad y conjuntos de entidades, 58–59
tipos de relación como atributos, 63
tipos de relación, 61–62
tipos de relación, nombres de rol, 63
tipos de relación, relaciones recursivas, 63
tipos de relación, restricciones en, 65–66
ERP (Planificación de recursos empresariales), 23
errores del sistema, 522
errores locales, 523
ERWin, herramienta de diseño, 345, 381
escala en los sistemas de bases de datos, 20
escape, carácter de, 221
escritura ciega, 538
escritura forzosa, ficheros de registro, 526
escritura restringida, 537
espacio de nombre, XML, 943
espacio, utilización del, 365
especialización. *Véase también* generalización
cuándo usar, 99
definición de clase usando, 91–93
definición de término EER, 104–107
definición, 109
en modelado usando categorías, 103
jerarquías/entramados, 98–99
mapeado, 196–198
notación UML de, 105–107
opciones de diseño para, 104–105
restricciones, 94–98
total, 97–98
especialización definida por atributo
definición, 96
modelo EER, 104
reglas inserción/eliminación para la, 98
especificación, 110
espejo, 420
esquema. *Véase también* dependencias funcionales; diseño práctico de bases de datos; diseño de base de datos relacional
base de datos ER, 53, 69–71
bases de datos biológicas, 890
conceptual global, 355
convenciones de denominación para el, 70
copo de nieve (*snowflake*), 855
de componentes, FDBS, 762
de relación multinivel, 690
definición, 112
denominación de estructuras, 70
descripción de tipos de entidad, 58
diagramas, 29
en estrella, 855
en la arquitectura de tres capas, 31–32
especialización para el, 99
estructuras, 29
evolución, 31, 212–213, 364
extensión del, 31
externo, 32, 346, 421, 762
generalización para, 99
independencia de datos y, 32–33
intensidad del, 31

interno, 31
modelado KR, 107
modelo de datos relacional, 124
relación, 283–284
relacional universal, 292, 318–319
restricciones, 129–131
sentencias de cambio, SQL, 212–213
SQL, 205–207
XML, 807–813
ESRI (Environmental Systems Research Institute, Inc.), 886–887
Estaciones base (BS), 866
estado
actual, 30, 126
consistente, 526
incorrecto, 132
inicial, 30
parcialmente confirmado en las transacciones, 524
terminado, transacciones, 524
vacío, 30
válido, 30, 132
de igualdad de los objetos, 603
estados legales de relación, 292
Estándar de cifrado avanzado (AES), 698
Estándar de cifrado de datos (DES), 697
Estándar de procesamiento de información federal (FIPS), 886
estándares
GIS, 882–886, 887
interoperabilidad y, 621–622
ODMG-93, 598
portabilidad y, 621
SQL, 654
estrategia binaria equilibrada, 358
estrategia mixta, 355, 358
estructura de acceso auxiliar, 392
estructuralmente equivalente, SQL, 658
etiquetas, HTML, 785–788
evaluaciones materializadas, 488
Evento-condición-acción (ECA), modelo, 707
eventos, 717–718
bases de datos de tiempo, 717
ejemplo, 712–714
excepciones
especificar firmas de operación, 631
Java, 261
ODMG, 624
EXCEPT, operación, 219. *Véase también* MINUS
EXECUTE, comando, 260
EXISTS, función, 226–228
explorar interfaces, 35
exportar esquema, FDBS, 762
expresiones
álgebra relacional, 145
cálculo relacional, 171, 176
declarativas, 169, 272
no seguras, 176
ruta de acceso, 639–642
seguras, 176, 179
extends, palabra clave, 631, 649
extensiones, 609, 631–632

**F**

factory, objetos, 632–633
fallos catastróficos, 523, 588
fallos en la transacción, 522–524
FALSE, valores
álgebra relacional, 148
cálculo relacional de dominio, 177
cálculo relacional, 171
tipo de datos booleanos, 208
fantasma, problema en el control de la concurrencia, 565
fase de análisis, algoritmo ARIES, 584–587
fase de escritura, control de la concurrencia optimista, 560
fase de expansión, bloqueo en dos fases, 550
fase de lectura, control optimista de la concurrencia, 560
fase de mezcla, algoritmo de ordenación, 466–467
fase de reducción, bloqueo en dos fases, 550
FDBS (Sistemas de bases de datos federados), 44, 761
fecha, tipos de datos, 208
fenotipos, 894
FETCH, comando, 258, 263
ficheros
almacenamiento. *Véase* almacenamiento
completamente invertidos, 457
de desbordamiento, 409
de resultado, 474
definición, 400
desordenados, 405–406
dinámicos, 405, 412–417
directo, 406
estáticos, 405
factor de carga, 417
*heap* (desordenado), 392, 405–406
maestros, 409
mixtos, 400
operaciones, 403–405
operaciones de un conjunto a la vez, 34
operaciones de un registro a la vez, 34, 45–46, 403
ordenados, 392, 406–409
otras organizaciones de ficheros principales, 417–418
principales, 409
relativos, 406
secuenciales indexados, 441
técnicas de dispersión. *Véase* técnicas de dispersión
FIFO (primero en entrar, primero en salir), 573
fila
actual, 263

definición, 205
filas actuales en iteradores, 313
ordenar en sentencias CREATE TABLE, 207
filtrado en la operación con ficheros, 403–404
find, comando, 404
FIPS (Estándar de procesamiento de información federal), 886
firmas, 11, 600, 606
firmas digitales, 699
flujo explícito, 696
flujos implícitos, 696
FLWR, expresión en XQuery, 820
FOR UPDATE OF, cláusula, 258–259
FOR UPDATE, palabra clave, 258–259
formas normales
Boyce-Codd, 308–311
cuarta (4 FN), 332–337
dominio-clave (DKNF), 340–341
insuficiencia de, 318
panorámica de, 298–300
primera (1 FN), 300–303
quinta (5 FN), 337–338
segunda (2 FN), 304, 305–308
temporal, 722
tercera (3 FN), 304–308
formatos
definición, 125
disco, 393
HTML, cabecera de documento, 786
fórmulas
cálculo relacional de dominio, 177
cálculo relacional, 171
cuantificadores en las, 171
negación/no negación, 174
forzar/no forzar, técnica de recuperación, 573–575
fragmentación
datos, 754–759
híbrida, 756
horizontal, 751, 755
mixta, 756
panorámica de, 754–755
transparencia, 751
vertical, 751, 755–756
FROM, cláusula, 214, 229–230
FS (Host fijos), 866
FULL OUTER JOIN, 230
funciones. *Véase también* comandos; operaciones
agregadas, 146, 163–164, 230–231
AVG, 230–231
biblioteca de, 253, 264
cadenas PHP, 790
constructor, 633
COUNT, 230–231
de subtipo local, 611
definición, 11
en QBE, 913–915
EXISTS, 226–228
integradas en SQL, 657
interfaces de llamada, 264–266
MAX, 230–231
NOT EXISTS, 226–228
PHP, 793–794
preservación del orden, 413
regresión, 842
sobrecarga, 658–659
SUM, 230–231
UNIQUE, 226–228
funciones de agregación
definición, 146, 163–164
implementación, 478
OQL, 642
QBE, 913–915
SQL, 230–231, 465–466

## G

GALILEO, 361
GAs (algoritmos genéticos), 843
GC (ontología del gen), consorcio, 896
GDB (base de datos del genoma), 893–894
GenBank, 893, 895
Gene Expression Omnibus (GEO), 896
generador de código, consulta, 463–464
genética molecular, 890
genoma, 892–893
GEO (Gene Expression Omnibus), 896
Geodatabase, 886–887
get, comando, 404
GIS (Sistemas de información geográfica), 878–889
aplicaciones y software, 886–887
características de los datos en, 879–881
componentes, 879–880
definición, 878
estándares y operaciones, 882–886
mejoras en el DBMS para, 882–883
modelos de datos conceptuales para, 881–882
móvil, 888
panorámica de, 878–879
recursos para, 896–900
temporal, 888
trabajo futuro en, 887–889
usos de, 3
glosario, 112
GORDAS, 361
grado de relación, 62, 75–78, 125
gráficos acíclicos, 45
gráficos de versión, 617
gráficos, multimedia, 873
consulta, 174, 481–483

dependencia del predicado, 741–742
gráficos acíclicos, 45
usando el lenguaje QBE, 179
GRANT OPTION, 687–689
granularidad de los elementos de datos, 519, 545, 560–563
GROUP BY, cláusula, 231–234, 465–466, 644
Grupo de modelos de datos de objetos. *Véase* ODMG
guión de subrayado (_), 220, 790
GUI (interfaces de usuario gráficas), 18, 35, 43, 845

**H**

hardware
base de datos distribuida, 754
coste de adquisición del, 362
dispositivo de disco, 393–397
GIS, 879
*hashing*. *Véase* dispersión
HAVING, cláusula, 231–234, 465–466, 645
herencia
datos, 665–667
de comportamiento, 631
de tipos, 91
definición, 91, 600
diseño de base de datos relacional, 647–649
diseño de bases de datos de objetos, 647–649
en el diagrama UML, 105
en ODL, 633
mapeado EER a esquema ODB, 648
modelo de objeto ODMG, 632
múltiple, 98–99, 197, 616
para las interfaces integradas, 624
selectiva, 616
sintaxis SQL para la, 655, 658–659
tipo de jerarquía y, 610–612
herramientas comerciales, minería de datos, 844–847
herramientas de diseño de bases de datos automáticas, 378–381
heterogeneidad semántica, 761–762
HGMDB, bases de dato, 895
HGP (Proyecto del genoma humano), 892
HML, etiquetas de inicio, 785
homónimo, 357
Horn, cláusulas, 733–735
Host fijos (FS), 866
HTML (Lenguaje de marcado de hipertexto)
datos web usando, 22
definición, 22
etiquetas, 785–788
páginas estáticas, 80
panorámica de, 783–784
programas PHP, 788–790
recopilar datos de, 797–798
huecos entre bloques, 393

**I**

IBM
creación SQL, 244
lenguaje QBE, 177–179
modelo de datos relacional, 123
identidad única, 601
identificación
definición, 108
manual, 729
minería de datos, 825–826
relación, 67–68, 375
tipo de entidad, 67–68
identificador de autorización, 205, 685
identificador de objeto. *Véase* OID
imagen antes (BFIM), 529, 573
imagen después (AFIM), 573
imágenes
almacenamiento y recuperación de, 22
datos multimedia, 873, 875–876
de mapas de bits (bitmaps), 613
raster, 881
implementación
bases de datos activas, 710–712
diseño de bases de datos, 349–352, 365–366
vistas, 241–243
IN, operador, 224
inanición, 554
índices, 429–461
claves múltiples y, 453–454
control de la concurrencia en los, 563–565
definición, 29
densos, 431, 434, 478
diseño, 505–506
en el almacenamiento de datos, 856–857
en Oracle 8, 673
escasos, 431
espaciales, 727
imágenes, 875–876
lógicos, 456
multinivel, 438–453
no densos, 478
ordenados, 454, 644
papel en el almacenamiento, 17
principales, 409, 430–433
refinación, 508–509
repaso/ejercicios, 457–461
secundarios, 434–437
tipos de, 430
vídeo, 730
índices agrupados
definición, 430
diseño físico para, 505
implementando operación SELECT, 468, 491
implementando operadores de agregación, 478

panorámica de, 433–434
propiedades, 437
índices multinivel, 438–453
árboles B, 442, 444–446, 453
árboles B+, 442, 446–453
árboles de búsqueda, 442–444
multinivel dinámicos, 441
panorámica de, 438–441
repaso/ejercicios, 457–461
índices ordenados de un nivel, 430–438
agrupados, 433–434
principales, 430–433
repaso/ejercicios, 457–461
secundarios, 434–437
tipos de, 430
índices principales o primarios
definición, 430
ficheros ordenados y, 409
implementación de SELECT, operación, 468, 491
panorámica de, 430–433
índices secundarios
definición, 430
ficheros completamente invertidos, crear con, 457
panorámica de, 434–437
propiedades de los, 437
SELECT, usando, 468, 491
inferencias
control de, 683
definición, 107
permitir uso de reglas, 19
regla para la dependencia de la inclusión, 338
reglas para las dependencias funcionales, 293–296
reglas para las dependencias funcionales/multivalor, 334
información
almacén de, 39, 347
ocultar, 604–606
Informix Universal Server, 661–670
herencia, 665–667
índices, 667
modelo de datos relacional extendido, 662
orígenes de datos externos, 667
panorámica de, 661–662
rutinas definidas por el usuario, 664–665
soporte API Data Blades, 667–670
tipos de datos extensibles, 662–664
ingeniería
concurrente, 616
de software asistida por computador. *Véase* CASE
inversa, 375
inmutable, definición, 601
INSERT, palabra clave, 709, 712
INSERT, privilegio, 689
insertar, operación
anomalías, 286
control de la concurrencia y, 565
definición, 137–138, 404
en las bases de datos de tiempo válido, 719
en los registros ordenados, 406–409
QBE, 915–915
reglas para especialización/generalización, 98
SQL, 235–237
instancias
de relación, 62
definición, 29
en los diagramas ER, 69–71
instanciación, 108
variables de, 599
Instituto nacional americano de normalización (ANSI), 204
integridad de entidad, 133–134, 691
integridad interinstancia, 691
intención, definición, 30, 58
interbloqueo, 552–554
interfaces
amigables para el usuario, 35
basadas en formularios, 35
basadas en menús, 35
de lenguaje natural, 35
de usuario basados en la Web, 35
de usuario gráficas (GUIs), 18, 35, 43, 845
definición, 11
herramientas de diseño, 380
instanciables, 631
interactivas, 36–37, 252
llamada a función, 264–266
multiusuario, 18
no instanciables, 631
tipos de, 35–36
Interfaz de nivel de llamadas (SQL/CLI), 243, 264–268
interfaz de objeto, 624–629
interfaz de programación de aplicaciones (API)
definición, 253
en arquitectura cliente/servidor, 42
en programación de bases de datos, 264
herramienta de minería de datos, 845
Internet. *Véase también* e-commerce; Web
semántica web, 875
interoperabilidad, 621–622, 660
interpretaciones de reglas, 735–737
INTERSECTION, operación, 103, 151–152, 219–220, 477
intervalos, 208
INTO, cláusula, 257, 263
IRM (administración de recursos de información), 346–348
IS NOT, operador de comparación, 223
IS, operador de comparación, 223

ISAM (Método de acceso secuencial indexado), 429, 441
ISDBS, 872
ISO, 204, 888
iterador posicional, 262
iteradores con nombre, 262

**J**

JAD, 352
Java
  almacenamiento persistente, 17
  con SQL, 204
  incrustación de comando, 260–262
  llamadas a función, 268–272
  problema por desajuste de la impedancia, 253
JDBC (Conectividad de bases de datos de Java)
  acceso a SQL mediante, 243
  incrustación de comandos en, 260–262
  JDBC frente a, 264
  llamadas a funciones en, 268–272
  uso en arquitectura cliente/servidor, 42
jerarquías
  de clasificación, 826
  de tipos, 610–613
  DML de alto nivel, 34
  ejemplos de DBMS basados en, 44
  en bases de datos antiguas, 20
  en el control de acceso basado en roles, 692
  especialización, 98–99
  generalización, 98–99
  modelo de datos jerárquico XML, 803–805
  modelo, 29, 45–46
  reglas de asociación entre, 834
JOIN, operación
  conservación de la dependencia usando, 319
  extensiones a, 165–166
  panorámica de, 155–159
  SQL, 229–230
  valores NULL/tuplas colgantes en, 328–330
*jukeboxes* de cintas, 391

**K**

KDD (Descubrimiento de conocimiento en bases de datos), 824–826

**L**

LAN (red de área local), 39, 754
LARGE-TEXT, tipos de datos, 669
latencia, 397
LBA (dirección de bloque lógica), 394
lectura irrepetible, 522
lectura sucia, 539
LEFT OUTER JOIN, operación, 230
lenguaje de consulta de objetos. *Véase* OQL
lenguaje de consulta estructurado. *Véase* SQL (Lenguaje de consulta estructurado)
lenguaje de definición de datos. *Véase* DDL (lenguaje de definición de datos)
lenguaje de definición de objetos (ODL), 604, 633–639
lenguaje de definición de vistas (VDL), 37
lenguaje de manipulación de datos. *Véase* DML (lenguaje de manipulación de datos)
lenguaje de marcado de hipertexto. *Véase* HTML (Lenguaje de marcado de hipertexto)
lenguaje de marcado extensible. *Véase* XML (Lenguaje de marcado extensible)
lenguajes de especificación de formularios, 35
lenguaje de programación
  almacenamiento persistente y, 17
  conjuntos de valores y, 59–60
  de propósito general, 606
  estructura de base de datos y, 9–10
  híbridos, 599
  lenguajes de bases de datos y, 33–35
  OO puro, 599
  orientado a objetos (OOPL), 599, 607
  rol en el entorno de bases de datos, 37
  uso con SQL, 204
LIKE, operador de comparación, 220
lista, tipo colección, 664
literales, 622–624, 733
llamadas a función dinámicas, 264–268
llaves ({ ... }), 244–245
locate, comando, 404
lógica de operación, 732
longitud variable, registros de, 400
LSN (número de secuencia de registro), 585–587

**M**

MAC (control de acceso aleatorio). *Véase* control de acceso aleatorio (MAC)
MANET, 867–868
mantenimiento
  coste del mantenimiento de un DBMS, 362
  funciones DBMS, 5
  principal, ciclo vital, 348–349
  secundario, ciclo vital, 348
  personal de, 15
mapeado
  de categorías, 199
  definición, 32
  EER, modelo, 649–649
  ER-a-relacional, 189–194
  modelo de datos, 364
  subclases compartidas, 197
  usando herramientas de diseño automáticas, 378–381
  usando Rational Rose, 374–378

máquinas cliente, 40–43
marca de tiempo
de transacción, 553
de escritura, 555, 557
de lectura, 555, 557
definición, 545
ordenar, 555–556
panorámica de, 555–555
tipo de datos, 208–209
transacción de base de datos de tiempo, 720
marcado, HTML, 785
marcador de borrado, 406
marcadores de interacción, 369
MAX, función, 230–231
MAX, operador, 478, 643
mecanismos de razonamiento, 107
medidas de control, frente a amenazas, 683, 700
memoria
caché, 390–391
compartida, 750
coste de ejecutar consulta, 489
EEPROM, 391
flash, 391
jerarquía, 390–391
principal, 390
RAM (Memoria de acceso aleatorio), 390, 419
mendeliana, genética, 890
menos utilizado recientemente (LRU), reemplazar página, 573
menús, interfaces controladas por, 18
mercados de datos, 854
meta-clase, 108
metadatos
administración, 347
definición, 5
en la estructura de bases de datos, 9
Método de acceso de almacenamiento virtual (VSAM), 457
Método de acceso secuencial indexado (ISAM), 429, 441
método de copia primaria, control de la concurrencia distribuido, 768–769
método de votación, control de la concurrencia distribuida, 770
métodos, 11
MGL, protocolo, 562–563
*middleware*, capa, 44
MIN, operador, 478, 643
minería de datos, 823–849
agrupamiento, 839–841
algoritmos genéticos, 843
almacenes de datos frente a, 960
aplicaciones de la, 22, 844
clasificación, 836–839
descubrimiento de patrones en series temporales, 842
descubrimiento de patrones secuenciales, 841
herramientas comerciales, 844–847
redes neuronales, 843
reglas de asociación, 826–984
regresión, 842–843
repaso/ejercicios, 847–849
visión general de la tecnología, 824–826
minimundo, 4
MINUS, operación, 151–152. *Véase también* EXCEPT, operación
modelo de datos raster, GIS, 881
modelo de datos relacional, 123–144
características relacionales, 126–129
delete, operación, 138–139
dependencias
dependencias funcionales. *Véase* funcional
en la clasificación DBMS, 44–45
funciones agregadas en el, 230–231
insert, operación, 137–138
notación, 129
operaciones de actualización, 139–140
panorámica, 123–146
repaso/ejercicios, 140–144
restricciones, 129–137
SQL frente al, 213
terminología, 124–126
modelo de datos vectorial, GIS, 882
Modelo de objeto de documento (DOM), 806
modelo de red, GIS, 882
modelo relacional anidado, 674–676
modelo relacional plano, 128, 674–675
modelo teórico, interpretación de reglas, 735–737
modelos de datos
categorías de, 28–29
como criterios DBMS, 44
como tipo de abstracción de datos, 11–12
de bajo nivel, 28
definición, 28
diseño lógico, 351
en GIS, 888
en la arquitectura de tres esquemas, 31–32
en las bases de datos biológicas, 890
funcionales, 63
herramientas de diseño de bases de datos automáticas para los, 378–381
mapeado, 351
para almacenes de datos, 854–857
Rational Rose para, 374–378
representativos, 28
modelos mínimos, 737
modificación de operación
anomalías, 286–288
definición, 137
modify, comando, 404

proceso de modificación, 358
modificar privilegios, 686
módulos
búfer, 18
cliente, 27, 660
cliente/servidor, 27
como componente DBMS, 15, 36–37
Data Blade, 662
módulos cliente, 27, 660
módulos de búfer
en almacenamiento en disco, 394
en consultas, 18
en entorno de bases de datos, 37
en transferencia de bloque, 398
módulos de componentes, 36–38
MOLAP (OLAP multidimensional), 861
monitorización
ciclo vital, 349
usando condiciones en, 238–239
utilidad para monitorizar rendimiento, 39
motor de inferencia, 730
MTTF (Tiempo medio entre fallos), 420
MU (unidades móviles), 866
multiconjunto de tuplas, 149, 213, 219–220
multiplicidad, 74
multiprogramación, 518
multiusuario
interfaces, 18
procesamiento de transacciones, 13
restricción por acceso no autorizado, 17
sistemas de bases de datos, 44, 757
MVD (dependencia multivalor) trivial, 332
MySQL, 896

**N**

NAS (almacenamiento conectado a la red), 424
NATURAL JOIN, operación
descripción, 156–159
uso en descomposición, 320–322
uso en SQL, 229–230
valores NULL/tuplas colgantes en, 328–330
NCLOB, 672
NEST, operación, 676
NEW, palabra clave, 709, 713–714
NFNF, 674–676.
nivel externo, arquitectura de tres esquemas, 32
no espera (NW), algoritmos, 553
No-1NF, 674–676
nodos
árbol de búsqueda, 442–444
árboles B, 444–446
árboles B+, 446–453
base de datos distribuida, 754
constantes, 174, 482
descendientes, 442
gráficos de dependencia de predicado, 741–742
hijo, 442
internos, 442, 447
niveles, 442
padre, 442
panorámica de, 442
nodos hoja
de árboles de consulta, 160, 482
definición, 97
insertar registros en árboles B+, 446–447, 449–451
no-forzar, método, 573–574
nombres de rol, relaciones recursivas y, 63
nombres de tipos, 610–611
nombres, objetos, 622
normalización
algoritmos, 330–331
definición, 298–300
desnormalización, 300, 506–507
en las herramientas de diseño de bases de datos automáticas, 378–381
no-robar, método, 573–574
NOT EXISTS, función, 226–228
NOT, operador, 171, 223
notación de flecha, 624
modelo de datos relacionales, 129
SQL, 244–245
UML. *Véase* UML (Lenguaje de modelado universal)
notación de punto, 656
aplicación de operaciones a objetos con la, 607, 624
expresiones de ruta utilizando la, 639
sintaxis SQL, 656
NO-UNDO/REDO, algoritmo de recuperación, 571, 577, 580, 582
NULL, valores
comando INSERT y, 235–237
definición, 57
en las tuplas, 128, 288–289
funciones agregadas y, 231
problemas con los, 328–330
reglas SQL para los, 222
restricciones en los, 130–131, 209
número de secuencia de registro (LSN), 585–587
NW (no espera), algoritmos, 553

**O**

objeto agregado de nivel superior, 109
objetos
atómicos, 622
binarios grandes (BLOB), 208, 400, 613, 659, 672
compuesto, 110
diagramas de objetos, 368
dispersos, 599
en la clasificación de DBMS, 44–45

especificar comportamiento, 606–607
especificar persistencia, 607–610
estados, 601–604
excepción, 108
grandes de caracteres (CLOB), 613, 659, 672
identidad, 601, 655
interfaces integradas para colección, 624–629
iteradores, 624
moleculares, 110
operaciones de modificador, 607
panorámica de, 599–600
persistentes, 599, 607–610
tiempos de vida, 623
transitorios, 599, 607
vistas, 672
objetos complejos, 613–615
ejemplo de, 602–604
estructura de datos de los, 599
estructurados, 614–615
no estructurados, 613–614, 659
Oracle 8 utilizando tablas anidadas, 672
ODBC (Conectividad de base de datos abierta)
en la arquitectura cliente/servidor, 42
en SQL, 243
en SQL/CLI, 264–268
herramienta de minería de datos, 844–847
ODL (Lenguaje de definición de objetos), 604, 633–639
ODMG (Grupo de modelos de datos de objetos), 622–633
clases, 631
claves, 632–633
definición, 598
extensiones, 632
herencia, 631
interfaces integradas para objetos de colección, 624–629
objetos atómicos, 629–631
objetos factory, 632–633
objetos y literales, 622–624
panorámica de, 622
relaciones binarias, 600
ODBMS (Sistema de administración de bases de datos de objetos), 597
OGC, 883–884
OID (identificador de objeto)
definición, 599, 622
diseño de bases de datos de objetos, 647
identidad de objeto y, 601
sintaxis SQL, 655–656
sistemas GIS, 880
OLAP (procesamiento analítico en línea)
almacenamiento de datos usando, 852
con SQL, 243
definición, 851
usos de, 3
OLAP (ROLAP) relacional, 861
OLAP multidimensional (MOLAP), 861
OLD, palabra clave, 709, 713–714
OLTP (Procesamiento de transacciones en línea), 13, 45, 140, 852
OMDG, 29
OMIM, 894, 895
Ontología del gen (GC), consorcio, 896
ontología, 107–108, 110, 876, 896
OO (orientación a objetos), 598
OOA (análisis orientado a objetos), 352–353, 901–903
OODB (base de datos orientada a objetos)
almacenamiento persistente en, 17
arquitectura cliente/servidor de dos capas, 42–43
conceptos, 598–600
historia de, 22
lenguajes, 253
modelado utilizando la herramienta Rational Rose, 374–378
sistema objeto-relacional, 45
opciones de relación múltiple, 196–197
OPEN CURSOR, comando, 257
open, comando, 404
operación destructor, 607
operaciones
actualizar, 139–140
aritméticas, 167, 220–221
básicas, 28
binarias, 146, 151–152
CARTESIAN PRODUCT, 152–155
conmutativas, 148, 152
costes, 362
CROSS PRODUCT, 152–155
definición, 11
definidas por el usuario, 28
DIVISION, operación, 159–160
insert. *Véase* insertar, operación
INTERSECTION, 103, 219–220
JOIN, 155–159, 165–166, 229–230, 319, 328–330
LEFT OUTER JOIN, 230
MINUS, 151–152
modificar, 137, 288, 358, 404
NATURAL JOIN, 156–159, 229–230, 320–322, 328–330
OUTER JOIN, 165–166, 229–230, 328–330, 479
OUTER UNION, 167
PROJECT, 148–149, 477, 481
REDO (rehacer), 580, 585–587
RENAME, 149–151
RIGHT OUTER JOIN, 230
SELECT, 146–148
UNION, 151–152
operaciones relacionales binarias
asignación o mapeo, 192–194

CARTESIAN PRODUCT, operación, 152–155
convenciones de denominación para las, 70
cuándo elegir las, 75–78
definición, 62, 146
INTERSECTION, operación, 151–152
MINUS, operación, 151–152
restricciones, 65–66
UNION, operación, 151–152
operaciones unarias
PROJECT, 146
SELECT, 146–148
operador de elemento, OQL, 642
operadores
álgebra relacional, 148
aritméticos, 167, 220–221
cálculo, 171
consulta, 223–225
lógicos, 223
nombres de, 615
SQL, 214
operadores de comparación
álgebra relacional, 148
búsquedas binarias, 408
como condición de registro, 404
concatenaciones, 230
consultas anidadas, 223–226
ficheros ordenados, 406
IS, 223
LIKE, 220
operaciones en archivos, 404
SQL, 214, 230
optimización de consultas basada en el coste, 489–498
componentes para la, 489
consultas de relación múltiple y ordenación JOIN, 495
definición, 489
ejemplo de, 495–498
información de catálogo utilizada, 490–491
para JOIN, 493–495
para SELECT, 491–493
optimización de semántica de consulta, 499
optimización, minería de datos, 826
optimizadores de consulta, 463–464, 510–512
OQL (lenguaje de consulta de objetos), 638–645
agrupamiento, operador, 644–645
consultas con nombre, 642
consultas sencillas, 639
expresiones de colección ordenada, 644
expresiones de ruta, 639–642
extraer elementos simples, 642
operadores de colección, 642
puntos de entrada, 639
resultados de la consulta, 639–642
variables iteradoras, 639
OR, operador, 171, 223
Oracle, 671–673
bases de datos distribuidas en, 772–775
formularios, 35
herramienta de diseño, 345
Open Gateways, 774
optimización de consultas en, 498
sintaxis de concatenación en, 230
SQLJ en, 260–264
ordenación de marca de tiempo (TO), 556–557
ordenación externa, 406, 466–467
ordenación, definición, 75
ORDER BY, cláusula, 221
organización de ficheros principal, 392
orientación a objetos (OO), 598
orientación de extracción, 259
OUTER JOIN, operaciones
en SQL, 229–230
implementación, 479
panorámica de, 165–166
valores NULL/tuplas colgantes en, 328–330
OUTER UNION, operación, 167
*overflow*. *Véase* fichero de desbordamiento

**P**

paginación en la sombra, 583–584
páginas HTML estáticas, 80
páginas web dinámicas, 783, 788
Paradox DBMS, 909
parámetros de disco, 905–907
parámetros de sentencia, 267
partición
algoritmo de concatenación de dispersión, 471–475
en unidades móviles, 868
índices y tablas, 673
uniforme, 475
participación, 62, 65
participación total, 65, 67
patrón de subcadena, coincidencia de, 220–221
patrones secuenciales, 824, 826, 841
PEAR DB, librería, 795
pérdida de la última actualización, 521–522
permisos, control de acceso basado en el roles, 692–693
PGP, 693
PHP, lenguaje de *scripting*
arrays, 792–793
conexión a una base de datos, 796–797
consultas de recuperación, 798–799
ejemplo, 788–790
estructuras de programación, 792
funciones, 793–794
panorámica de, 783
repaso/ejercicios, 799–801
tipos de datos, 792

variables de servidor, 794–795
variables, 790–792
pistas, disco, 393–394
planes de ejecución, 463–464, 488–489
planificación de transacciones
definición, 530
equivalencia de vista, 537
no serie, 530–533
recuperabilidad, 528–529
resultado equivalente, 532
serialización de vista, 537
serie, 530–532
transacciones débito-crédito, 538
usos de la, 536–537
planificaciones completas, 527
planificaciones estrictas, 529
planificaciones irrecuperables, 528
planificaciones no serie, 530–532
plantillas, QBE, 909–913
plataforma
basada en la infraestructura, datos móviles, 866
definición, 363
sin infraestructura, datos móviles, 867
poli-instanciación, 690, 692
polimorfismo, 600, 615
Popkin Software, 345
porcentaje (%), signo, 221, 401
portabilidad, 363–364, 621
POSTGRES DBMS, 661
PowerBuilder (Sybase), 39
precompiladores, 37, 253, 255
predicados
bloqueo, 566
cálculo de fórmulas, 733
definición, 129
definidos por hechos, 737
definidos por reglas, 737
gráficos de dependencia, 741–742
incorporados, 733
notación Datalog, 730–733, 737–739
sistema Prolog, 731
predicción en la minería de datos, 825
PREPARE, comando, 260
preprocesadores, 253, 255
de hipertexto, 783
primer nivel en índices multinivel, 439
primera forma normal (1 FN), 128, 300–303, 330
primero en entrar, primero en salir (FIFO), 573
privilegios
asignación de, 686
definición, 17
ejemplo de, 687–689
especificar usando vistas, 686
otorgar, 243, 687
propagación de los, 689
revocar, 243, 687
tipos de, 685–686
problema de la actualización temporal (lectura sucia), 522
problemas legales
derechos de propiedad intelectual, 700
seguridad de la base de datos, 681–683
procedimiento almacenado, 19, 272–273
procesamiento
analítico en línea. *Véase* OLAP (procesamiento analítico en línea)
basado en el flujo (*pipelining*), 480
consultas, 17–18
de transacciones, 13
de transacciones en línea (OLTP), 13, 45, 140, 851
diseño orientado a procesos, 350–351
especificación, 362
ficheros, 9
interpolado, 518
procesamiento paralelo
acceso paralelo a disco usando RAID, 419–421
almacenamiento de datos, 861
de transacciones concurrentes, 518
definición, 398
tecnología distribuida frente a, 750
PROCESS RULES, comando, 711
profundidad global, 415
profundidad local, 415
programación
almacenar objeto de programa, 17
de bases de datos dinámicas, 264
estática de base de datos, 264
estructuras, PHP, 792
independencia del funcionamiento del programa, 11
métodos, 8–9
programadores de aplicaciones, 15
programas cliente, 254
programas de aplicación
definición de datos en los, 9–10
definición, 5, 252
diseño y pruebas de los, 51
ejemplo de, 6–8
herramientas de desarrollo, 347
independencia del funcionamiento del programa para los, 11
reglas en la arquitectura de tres capas, 43–44
restricciones basadas en la aplicación en los, 129
PROJECT, operación, 148–149, 477, 481
Prolog, programas, 730–732
propagación horizontal, 689
propagación vertical, 689
propagar privilegios, 687, 689
propiedad
aislamiento, 13, 526, 539

atomicidad, 13
clase, 108
conservación de la consistencia, 526
de las formas normales, 298
de preservación de la consistencia, ACID, 526
de seguridad simple, 690
descomposición, 318
durabilidad, 526
objeto atómico, 629
transacciones, 13, 526–526
propiedad de concatenación no aditiva
conservación de la dependencia y, 319–328
de descomposición, 318–319
definición, 299
dependencias de concatenación y, 337–338
uso en descomposición en 4NF, 335–337
uso en descomposición en BCNF, 325
protección de la privacidad, 695, 699, 700
protección del sistema, 5
protocolo MGL, 562–563
Proyecto del genoma humano (HGP), 892
PSM, 243, 272–273, 654
puntero indirecto, 601
punto de confirmación, transacciones, 526, 580
puntos de control, recuperar
actualizaciones diferidas en sistema multiusuario, 579
difuso, 575
en el algoritmo ARIES, 585
panorámica de los, 575
puntos de entrada, 623, 639
puntos de tiempo, 717

**Q**

QBE (Consulta mediante ejemplo), lenguaje
agregación, 913–915
agrupamiento, 913–915
cálculo de dominio usando, 177–179
modificación de base de datos en, 913–915
panorámica de, 909
recuperaciones en, 909–913
quinta forma normal (5 FN), 337–338

**R**

R, árboles, 728
RAID, 419–423
raíz, estructura de datos en árbol, 442
RAM (Memoria de acceso aleatorio), 390, 419
RAM dinámica (DRAM), 390
Rational Rose, herramienta de diseño, 374–378
ratón, 35
razón de cardinalidad
definición, 126
en relaciones ternarias, 78
para las relaciones binarias, 65–66
RDBMS (sistema de administración de bases de datos relacional)
álgebra relacional en los, 173
arquitectura cliente/servidor de dos capas en los, 42
DDL para, 132, 134
definición, 22
evolución y tendencias actuales en los, 20–21, 123, 660–661
Informix Universal Server para, 662
modelo de datos relacional. *Véase* modelo de datos relacional
refinación, 507–512
representación de consultas, 160
SQL-99, estándar. *Véase* SQL-99, estándar
RDBMS comercial, 22
RDU_M, 579–580
RDU_S, 578–579
READ COMMITTED, SQL, 539
READ ONLY, SQL, 539
READ UNCOMMITTED, SQL, 539
READ WRITE, SQL, 539
READ, operación, 523
recuperación
definición, 137
operaciones, 403–405
PHP, 798–799
problemas con bases de datos multimedia, 874–876
QBE, 909–913
transacciones, 359–362
recuperar, 571–594
actualización diferida, 577–580
actualización inmediata, 581–583
algoritmos, clasificar para, 571–573
anular transacción, 575–577
ARIES, algoritmo, 584–587
causas de los fallos, 522–523
computación móvil, 871
estados de transacción y, 523–524
estrategia típica para, 571
fallos catastróficos, 588
método forzar/no forzar, 573–574
método robar/no robar, 573–574
sistemas multibase de datos, 587–588
paginación en la sombra, 583–584
planificaciones basadas en, 527–529
punto de confirmación utilizado en, 525
puntos de control, 575
registros de sistema usados en, 524–525
repaso/ejercicios, 589–593
sistemas de bases de datos distribuidos, 754, 770
sistemas de copia de seguridad y, 18, 38
subsistemas de recuperación, 18
recursividad lineal, 659–660

redes
base de datos distribuida, 754
irregulares trianguladas (TIN), 882
LAN (red de área local), 39, 754
neuronales, 843
particionamiento de, 768
unidades móviles, particiones frecuentes de, 868
WAN (red de área expandida), 754
REHACER, operación
actualizaciones diferidas, 578–580
en el algoritmo ARIES, 584–587
redundancia, control de la
como ventaja del sistema de bases de datos, 16–17
en dependencias funcionales, 297–298
en dependencias multivalor, 332–334
usando el algoritmo 11.4, 326–328
usando RAID, 420–421
REFCOUNT, tabla de sistema, 669
referencia cruzada, método de mapeo, 193
referencia, privilegio de, 655–656, 686
referencias a campo lógicas, 418
referencias inversas, 600, 604
referir relación, 133, 216–217
refinación
consultas, 510–512
directrices para la, 511–512
diseño de base de datos, 348, 350–351, 365–366, 509–510
índices, 508–509
panorámica de, 507–508
repaso/ejercicios, 512
registrar
controladores JDBC, 268
estadísticas de monitorización para la refinación, 507
punto de confirmación y, 525
sistema, 524–525
registro actual, 403
registro de sistema
boceto para la recuperación, 571
control del acceso usando, 684
copia de seguridad del, 588
panorámica de, 524–525
punto de confirmación y el, 525
registro desordenado, 405–406
registro, uno a la vez
DML, 34, 45–46
operaciones con ficheros, 404–405
registros
ancla, 431
bloquear, 402–403
búsqueda por, 403–405
de conexión, 265
de longitud fija, 400–402
de sentencia, 266
del registro, 524, 584–585
descripción, 266
desordenados, 405–406
entorno, 265
extendidos, 402
ficheros como secuencias de, 400
*hashing*. *Véase hashing*
longitud fija, 400
longitud variable, 400
mezclados, 417
modelos de datos basados en, 29
no extendidos, 402–403
ordenados, 406–409
tamaño, estimación para coste, 490
tipos de, 399
regla
de aumento, 294–295, 334
base de datos activa, 711–712
base de datos deductiva, 730–733
cálculo relacional, 171
comerciales o de negocio, 19, 129, 714–715
de complementación, 334
de regresión, 842–843
de deducción, 19, 706
de herencia, 19
dependencia de inclusión, 338
dependencia funcional, 293–296
dependencia funcional/multivalor, 334
desactivadas, 711
especialización, 98
generalización, 98
interpretaciones de las, 735–737
procesamiento en SQL, 673
reflexiva, 294–296, 334
regla 80-20, 360–361
restricción de datos, 19
seudotransitiva, 294–296
transitivas, 294–296, 334
reglas activas. *Véase también triggers*
aplicaciones potenciales, 714–715
definición, 705
dificultades con, 711–712
reglas de asociación, minería de datos, 826–843
algoritmo de árbol de patrón frecuente, 830–833
algoritmo de muestreo, 829–830
algoritmo de particionado, 833
Apriori, algoritmo, 828–829
asociaciones multidimensionales, 834–835
asociaciones negativas, 835–836
definición, 824, 826
entre jerarquías, 834
factores de complicación en, 835

reglas heurísticas, 480–489
árboles de consulta y notación de gráficos de consulta, 481–483
árboles de consulta, conversión, 488–489
definición, 465
optimización de consultas, 480–481
regresión lineal, 842
relaciones. *Véase también* esquema
agregación en, 109–110
anidadas, 243, 301–302
asociación en, 109–110
base. *Véase* dependencias funcionales
binarias, 75–78
campos de conexión en el fichero, 418
características, 126–129
complejas, 18
convenciones de denominación para, 70
de grado más alto. *Véase* relaciones ternarias
de rango, 170–171
definición, 29, 107
diseño, 60–61, 69–72
en modelos de red, 45
estado de, 292
extensión, 125, 368
implícitas, 62
incluidas, 368
intensidad, 125
jerarquías y entramados de especialización, 98–99
mapear, 192–194
matemáticas, 123
normalizar, 298–303
recursivas, 63, 164
referencia, 133, 216–217
renombrado, 149–151
restricción de integridad referencial, 133–134
superclase y subclase, 90–91, 106
ternarias, 75–78
tipos de entidades débiles, 67–68
todo clave, 334
universales, 318
virtuales, 207
relaciones binarias
mapeado de esquema EER a esquema ODB, 648–649
mapeado en el diseño de bases de datos de objetos, 648
RENAME, operación, 149–151
rendimiento
bases de datos distribuidas, 753
bases de datos multimedia, 874
del dispositivo, 507
reconstrucción de índices para, 508–509
refinación, 507
utilidad de monitorización, 39
REPEATABLE READ, nivel de aislamiento, SQL, 539
replicación
esquema de, 757
parcial, 757
simétrica, 773
replicar
base de datos distribuida, 754, 773
computación móvil, 870
ejemplo, 757–759
para dependencias multivalor, 334
transparencia, 751
representación diagramática, 354
requisitos de recopilación y análisis de datos
ciclo vital principal, 348
diseño de bases de datos, paso, 52
en diseño/implementación, 350–351
fase de diseño, 8
primera fase del diseño de la base de datos, 8
técnicas para especificar requisitos, 351–353
reset, comando, 404
restricción de cardinalidad mínima, 65
restricción de especialización parcial, 97–98
restricción disjunta, 97–98, 107
restricción de unicidad, atributos, 58, 504
restricciones
acceso no autorizado, 17
atributos, 58–59
basadas en el modelo, herencia, 129
cambiar, 32, 212–213
como aserciones/triggers, 238–239
como elemento de esquema, 209
conflictos, 356–358
CREATE TABLE, comando, 205–207
de transición, 137
definición de término EER, 104–105
denominación, 212
elecciones de diseño para la, 104–105
eliminar, 212–213
en el control de acceso basado en el roles, 692
en el modelado usando categorías, 103
en el modelo de datos relacional, 129–131, 134–137
en las formas normales de dominio de clave, 340–341
especialización/generalización, 94–98
especificación de lenguaje, 137
explícitas, 130
FDBS, 761
GIS, 881, 884–886
implícitas, 130
jerarquías/entramados, 98–99
mapeo, 196–199
notación de diagrama ER para las, 69–71
notación UML de la, 105–107
proyección generalizada, 162–163
relación ternaria, 78–78

SQL, 209–211
superclase generalizada, 93
tipo de relación, 65–67
UPDATE, comando, 237–238
usada en la definición de entidades, 93–94
violación en las operaciones, 137–140
visualización en los diagramas de esquema, 29
restricciones de integridad
en el comando INSERT, 235–237
en el comando UPDATE, 237–238
implementar, 18–19, 714–715
referenciales, 137–140, 338–339
reglas para relaciones multinivel, 691
utilizando GIS, 881
utilizando relación de modelo de datos, 132
restricciones de integridad referencial
definición, 133
en SQL, 210–211
restricción de la dependencia de inclusión y, 338–339
violación en operaciones, 137–140
retardo rotacional (rr), parámetro de disco, 397
REVOKE, comando, 687
RIGHT OUTER JOIN, operación, 230
ROLAP (OLAP relacional), 861
roles
sintaxis SQL, 660
sistemas GIS, 881
rr (retardo rotacional), parámetro del disco, 905
RSA, algoritmo de cifrado de clave pública, 698–699
ruta de acceso secundaria, 430
ruta de acceso. *Véase* acceso
rutinas externas, SQL, 660

**S**

same_as, operación, 624
SCSI, 396
SDBE, 872
SDL, 33
SDLT, 398
SDTS, 886
secciones críticas, 546
SECSI, herramienta de diseño, 381
secuencia de conjuntos de elementos, 842
segunda forma normal (2 FN)
conservación de la dependencia, 319
definición, 303
panorámica de, 305–367
probar con 3 FN, 308
segundo nivel, índices multinivel, 439
seguridad de la base de datos estadística, 683, 694–695
seguridad. *Véase también* acceso
auditorías, 685
autorización y, 17
base de datos distribuida, 754
base de datos estadística, 694–695
cifrado de clave pública, 698–699
computación móvil, 871
confidencialidad y, 690
control de acceso basado en roles, 692–693
control de acceso discrecional, 685–689
control de acceso XML, 693
control del flujo, 696–697
DBA y, 684
firmas digitales, 699
privacidad y, 699
protección del acceso, 684–685
repaso/ejercicios, 701–703
retos de la, 700–701
roles en SQL, 660
tipos de, 681–684
selección, condiciones de
cálculo relacional de dominio, 178
cálculo relacional, 171
consultas SQL, 213–216
operaciones con ficheros, 403–405
selección (recuperación o lectura), privilegio de, 686
SELECT, operación
AS en, 221
definición, 214
funciones de coste y, 491–493
implementación, 490–493
panorámica de, 146–148
uso de heurísticas en, 480
select-from-where, bloque, 214–216, 465–466
selección-proyección-concatenación, consultas, 216, 223
semántica
de propiedad, 614
definición, 19, 539
dependencia funcional y, 292
para atributos en las relaciones, 283–284
restricciones de integridad, 134–137, 881
restricciones, 129
web, 110, 875
sentencia
CALL, 273
CREATE, 205
CREATE ASSERTION, 204, 238–239
CREATE TRIGGER, 238–239
de final de transacción, 518
SEQUEL, 204
SERIALIZABLE, nivel de aislamiento, SQL, 539
serialización de planificaciones, 530–538
conflicto de serialización, 533–536
definición, 530
equivalencia por vista, 537
garantizar. *Véase* bloqueo en dos fases (2PL)
planificaciones de resultado equivalente, 532–533
planificaciones no serie, 530–532

planificaciones serie, 530–532
serialización de vista, 537
transacciones débito-crédito, 538
usos de la, 536–537
series de tiempo (cronológicas), tipos de datos
aplicaciones, 22
Data Blades API, 668–670
definición, 717
minería de datos, 826
panorámica de, 727
patrones en las, 842
servidores
computación móvil, 866
de aplicaciones, arquitectura de 3 capas, 43–44
de consultas, arquitectura de dos capas, 42
de ficheros, 41–42
en la programación de bases de datos, 254
módulo servidor, 27
nivel en la arquitectura cliente/servidor, 42–43
PHP, 794–795
rol en la arquitectura de tres capas, 43–44
tipos de, 40–43
de e-mail, 40–42
de impresión, 40–42
especializados, 40–42
SQL, 42
web, 41–42
sesiones de inicio, 685
SET DIFFERENCE, operación, 477
SET, cláusula, comando UPDATE, 237–238
SET, operación, 477
set-at-a-time, DML, 34
set-oriented, DML, 34
*shadowing* (en la sombra), 420, 573
SIMULA, lenguaje de programación, 599
sinónimos, conflictos con el nombre, 357
sintaxis de cifrado y especificación de procesamiento, XML, 693
sintaxis de firma, XML, 693
Sistema de administración de bases de datos de objetos (ODMS), 597
sistema de administración de bases de datos relacional. *Véase* RDBMS
sistema de administración de bases de datos. *Véase* DBMS
sistema operativo (SO), 36–38
sistema, cuentas del, 684
sistemas basados en el conocimiento, 706
sistemas de bases de datos
abstracción de datos en los, 11–12
control de la concurrencia en los, 13
coste, 362
cuestiones/ejercicios, 24–25
DBMS, método, 15–19, 23–24
definición, 5
diseño. *Véase* diseño práctico de bases de datos
ejemplo de, 6–8
historia de los, 20–23
independencia programa-datos en, 10–11
modelado usando el modelo ER. *Véase* ER (entidad-relación), modelo
panorámica de los, 4–5
sistemas de almacenamiento de datos, 39
sistemas de archivos frente a, 8–9
trabajadores en, 13–15
trabajadores entre bambalinas en, 15
sistemas de bases de datos deductivos
Datalog, consultas no recursivas, 740–742
Datalog, notación, 733
Datalog, programas y su seguridad, 737–739
definición, 706
Horn, cláusulas, 733–735
interpretaciones de reglas, 735–737
operaciones relacionales, uso de, 739
panorámica de, 730–731
Prolog/Datalog, notación, 731–733
repaso/ejercicios, 742–746
sistemas de bases de datos distribuidos
arquitectura cliente-servidor de tres niveles, 770–771
computación móvil como variación de los, 870–871
control de la concurrencia, 768–770
en Oracle, 772–775
fragmentación de los datos, 754–759
funciones, 753–754
panorámica de los, 749–750
paralelos frente a tecnología distribuida, 750
procesamiento de consultas en los, 762–768
recuperación en los, 770
repaso/ejercicios, 775–778
replicación de los datos, 757–759
tipos de, 759
ventajas de los, 751–753
sistemas de bases de datos expertos, 706
sistemas de bases de datos, arquitectura, 27–48
centralizados, 40–43
ciente/servidor básico, 40–42
clasificación de DBMS, 44–46
cliente/servidor dos capas, 42–43
cuestiones/ejercicios, 48
entornos de aplicación, 38–39
esquemas, 28–30
facilidades de comunicación, 39
herramientas, 39
independencia de los datos, 32–33
instancias, 28–30
interfaces, 35–36
lenguajes de bases de datos, 33–35
modelos de datos, 28–30

módulos componente, 36–38
panorámica, 27
para aplicaciones web, 43–44
resumen, 47
tres esquemas, 31–33
utilidades, 38
sistemas de copia de seguridad
definición, 18, 38
en almacenamiento de cinta magnética, 398
panorámica de los, 588
sitios, 769
sistemas de diccionario de datos, 39, 347
sistemas de información ejecutiva (EIS), 852
Sistemas de información geográfica. *Véase* GIS (Sistemas de información geográfica)
sistemas de red
en la clasificación DBMS, 44
modelo, 45
rol en la historia de las bases de datos, 20
software de comunicaciones en los, 39
sistemas decisión-soporte (DSS), 852
sistemas relacionales extendidos, 45
sistemas relacionales mejorados. *Véase* DBMS de objetos relacionales (ORDBMS)
sitio primario, control de la concurrencia distribuido, 768
sitio resultado, 764
SMALLTALK, lenguaje de programación, 599
*snapshot* (captura), 30
SNP, 773
SO (sistema operativo), 36–38
sobrecarga del operador (polimorfismo), 600, 615
software
coste, 362
de comunicación, 39
de terceros, 364
GIS, 878, 886
solapamiento, 96, 106
soporte, 827–828
SQL (Lenguaje de consulta estructurado)
componentes, 654
convertir consultas a álgebra relacional, 145–146, 465–466
desajuste de impedancia y, 253
dinámico, 259–260
encapsulación de operaciones en, 657–658
incrustado en Java (SQLJ), 260–264
incrustado, 252, 255–259
Informix Universal Server y, 661
Interfaz de nivel de llamadas (SQL/CLI), 264–268
llamadas a función para Java (JDBC), 268–272
Módulos almacenados persistentes (PSM) y, 272–274
panorámica, 251–252
privilegios, 686
procedimientos almacenados (SQL/PSM) y, 272–273
QBE frente a, 909, 913–915
repaso/ejercicios, 274–277
rutinas, 660
soporte de transacciones en, 538–540
SQL/CLI, 264–268
versiones, 204
SQL dinámico, 243, 251, 259–260
SQL incrustado, 243, 251–252, 255–259, 654
SQL*Forms, 35
SQL/Bindings, 654
SQL/CLI (Interfaz de nivel de llamadas), 243, 264–268
SQL/PSM (Módulos almacenados persistentes), 243, 272–274, 654
SQL/Temporal, 654
SQL/Transaction, 654
SQL-3, 660
SQL-99, estándar, 203–249, 654
agrupamiento, 232–234
alias, 216–217
aserciones, 238–239
características adicionales, 243
catálogo, 205
comparaciones conjunto/multiconjunto, 223–225
comparaciones, 222–225
conjuntos explícitos y atributos, renombrar, 228–229
consultas anidadas correlacionadas, 225–226
consultas anidadas, 223–225
consultas, 234–235
CREATE TABLE, comando, 205–207
DELETE, comando, 237
dominios, 207–209
esquema, 205
EXISTS, función, 226–228
funciones agregadas, 230–231
INSERT, comando, 235–237
nombres, 216–217
operadores aritméticos, 220–221
panorámica, 203–204
patrones de subcadena, 220–221
recursividad lineal en, 659
repaso/ejercicios, 244–249
restricciones, 137, 209–211
SELECT-FROM-WHERE, consultas, 214–216
sentencias de modificación del esquema, 212–213
soporte objeto-relacional en, 654–659
tablas como conjuntos, 218–220
tablas concatenadas, 229–230
tablas virtuales, 239–243
tipos de datos de atributo, 207–209
triggers, 238–239, 715
tuplas, 223–225
UNIQUE, función, 226–228
UPDATE, comando, 237–238
variables de tupla, 216–217

vistas, 239–243
WHERE, uso de cláusula y asterisco, 217–218
SQLCODE, 256
SQLJ, 260–264
SQLSTATE, 256
SSL, 693
STARBURST, sistema, 712–714
subárboles, nodo, 442
subclases
categoría, 100–101
compartidas, 98
definidas por condición, 96
definidas por el usuario, 97
definidas por subclases, 96, 98
en la especialización, 93–94
en la especialización, definición de conjuntos, 91–93
en la especialización, jerarquías/entramados, 98–99
en la especialización, mapeado, 196–199
en la especialización, restricciones, 96–104
en la generalización, 93–94
en la generalización, jerarquías/entramados, 98–99
en la generalización, mapeado, 196–199
en la generalización, restricciones, 96–104
modelo EER, definición, 104
panorámica de, 90–91
tipo UNION, 100–101
subconjuntos, 126, 130
subtipos, 611–612
SUM, función, 230–231
SUM, operador, 478, 643
suma incorrecta, problema, 522
superclases
especialización usando las, 91–93, 98–99
generalización usando las, 94–99
mapeado de estructuras del modelo EER a relaciones, 196–199
modelo EER y, 90–91, 104
múltiples, 100–101
notación UML para las, 105
superclave, 130–131, 300
supertipos, 611
superusuario, cuentas de, 684
Sybase (PowerBuilder), 39

**T**

tabla de transacciones, 585–587
tablas
anidadas, 672
base, 207, 212–213
comandos, 212–213
como conjuntos, SQL, 218–220
como elementos de esquema, 205
concatenadas, 229–230
consultas anidadas y, 223
de búsqueda, 193
de dimensión, 855
de hechos, 855
de resultado, QBE, 913
definición, 205
*hash*, 409
virtuales, 239–243
tamaño, factor de diseño de bases de datos, 346
TAXIS, 361
taxonomía, 112
técnicas de dispersión
estáticas, 413–414
extensible, 415
externas, 412–414
internas, 410–412
lineales, 416–417
panorámica, 409–410
partida, 454–455
tecnología de la información (TI), 346–348
Telelogic System Architect, 345
tercer nivel, índices multinivel, 440
tercera forma normal (3 FN)
definiciones generales de, 305
descomponer conservación de la dependencia en, 323–328
descomposición de relación en, 318–319
panorámica de, 303–308
resumen de, 331
texto
datos multimedia, 873, 876–876
tipos de datos, 669–670
Thomas, regla de escritura de, 556
TI (tecnología de la información), 346–348
tiempo de ejecución, procesador de base de datos en, 463–464
Tiempo medio entre fallos (MTTF), 420
tiempo. *Véase también* bases de datos temporales
componentes, parámetro de disco, 905–907
de búsqueda, 397, 905
de reescritura, parámetro de disco, 905
de respuesta, 365
de transferencia de bloque, parámetro de disco, 905
definido por el usuario, 717
dimensiones, 717
limitado, interbloqueos, 554–555
período, 717
restricciones, 504
tipos de datos SQL, 208–209
TIN (Redes irregulares trianguladas), 882
tipo conjunto, 45,
tipo de dato opaco, 663
tipo de entidad débil
definición, 67–68
elegir clave de relación y, 75

mapeado, 191–194, 648
notación ER, 69–71
tipo de entidad
distinto, 103
fuerte, 67–68
propietario, 67–68
regular, 67–68, 190–191
tipo de relación parcial, 65
tipos colección, 602, 624–629
tipos de colección multiconjunto, 664
tipos de datos
atributos/dominios, SQL, 207–209
bidimensionales, 667
biológicos, 890
campo de registro, 399
Data Blades API, 667–670
de cadena de bits, 208
de imagen, 668
desajuste de impedancia, problema de, 253
distintos, 663
extensibles, 613–614
fila, 663–664
modelo de datos relacional, 125
numéricos, 207, 791
Oracle 8, 671
PHP, 791
rol de, 6–7
tipos de relación
atributos de los, 66
como atributos, 63
definición, 62
en UML, 74
grado de los, 62
nombres de rol, 63
panorámica de, 61–62
recursivos, 63, 164
restricciones, 65–67
tipos definidos por el usuario. *Véase* UDT
tipos estructurados, 602
TO (ordenación de marcas de tiempo), 555–557
tolerancia a fallos, 871
total, término EER, 105
trabajo cooperativo, información multimedia, 873
transacción, base de datos de tiempo, 718, 720–721
transacciones
anulación, 575–577
búferes DBMS, 519–520
computación móvil, 870–871
control de la concurrencia para las, 520–522
de inicio, 518, 523
de sólo lectura, 518
débito-crédito, 538
definición, 6
diseño de base de datos, 359–361, 503–504
en bases de datos relacionales, 140
errores, 522
estadísticas de monitorización para la refinación, 507
estados y operaciones adicionales, 523–524
fichero, 409
interactivas, 566
mixtas, 360–361
operaciones de lectura y escritura de, 518–520
panorámica de, 518
planificaciones. *Véase* planificación de transacciones
punto de confirmación de las, 525
recuperación de, 522–523
registro del sistema, 524–525
rendimiento, 365
repaso/ejercicios, 540–542
requisitos funcionales para las, 52–54
servidores, 42
sistemas de procesamiento, 346–348
sistemas monousuario, 518
sistemas multiusuario, 13, 518
SQL, 538–540
subsistema de copia de seguridad y recuperación para las, 18
transacciones enlatadas
en SQL, 252
interfaces basadas en formularios y, 35
usuarios finales y, 14
transformaciones, 483–488
transparencia, 751
*triggers*
a nivel de fila, 710,
a nivel de sentencia, 710, 712–714
acción de activación referencial, 211, 237
definición, 19, 136, 706
en SQL-99, 715
granularidad, 660
modelo generalizado para los, 707–710
sintaxis SQL, 660
trivial, definición, 294
TRUE, valores, 208
TSQL2, lenguaje, 727
tuplas
actualizar, operación, 139–140
CARTESIAN PRODUCT, crear, 155
colgantes, 328–330
consultas anidadas utilizando, 223–225
eliminar, operación, 138–139
en el control de acceso obligatorio, 690–691
en operaciones binarias, 151–152
expresiones seguras/no seguras y, 176
falsas, 289–290, 320–322
insertar, operación, 137–138
múltiples, 262–264
notación de modelo relacional, 125, 128

operación JOIN para las, 155–156
operación OUTER UNION para, 167
operación SELECT, 146–148
operaciones OUTER JOIN para, 165–166
ordenar en las relaciones, 126–127
plantilla de generación de tuplas, 339–340
recuperación múltiple, 258–259, 262–264
recuperación sencilla, 254–257
restricción de integridad referencial, 133–134
restricciones, usando CHECK, 212
valores NULL en las, 288–289
variables, 216–217
versionado, 718–723

**U**

UDB (Base de datos universal), 32
UDT (tipos definidos por el usuario),
Informix Universal Server, 663–665
sintaxis SQL, 245, 654, 657–658
UML (Lenguaje de modelado universal), 51, 105–107
UML, diagramas
actividad, 371
caso de uso, 368
colaboración, 370
como diseño de aplicación de base de datos, 367–368
como estándar de especificación de diseño, 366–367
ejemplo, 372–374
gráfica de estado, 371
implantación, 368
objeto, 368
Rational Rose, herramienta de diseño, 374–378
de secuencia, 53, 75, 369–370
UNDER, palabra clave, 663–664
UNDO, entradas de registro de tipo, 573
UNDO (DESHACER), operación, 584–587
UNDO/REDO, algoritmo, 571, 573
unidades móviles (MU), 866
UNION, operación
compatibilidad, 151–152
en SQL, 219–220
implementación, 477
reglas de inferencia, 294–296
UNION, tipos
en modelado, 100–101
mapeado, 199, 648
trabajar con, 104
UNIQUE, cláusula, 211, 226–228
universo de discurso (UoD), 4
UNIX, sistema operativo, 788
UNKNOWN, valor, 208, 222–223
UNNEST, operación, 676
UoD (universo de discurso), 4
UPDATE, comando, 237–238
UPDATE, palabra clave, 709, 712–713
UPDATE, privilegio, 689
Upper CASE, herramientas, 353
usuarios
autorización crear esquema, 205
como criterios DBMS, 45
independientes, 14,
lenguajes de bases de datos, 34
modelos de datos, 28
monousuario frente a sistemas multiusuario, 518
múltiples. *Véase* multiusuarios
requisitos de recopilación de datos, 52–53, 352–353
restringir acceso no autorizado, 17
sofisticados, 14
tipos de, 13–14
usuarios finales, 14, 28–29, 35
casuales, 14, 35
paramétricos, 14, 35
principiantes, 14, 35
utilidad de carga, 38

**V**

validación
consultas, 463–464
control de la concurrencia, 559–560
documento XML, 806
ciclo vital principal, 348
ciclo vital secundario, 348
valor no disponible, 222
valores
atómicos. *Véase* atributos atómicos
ordenados, SELECT, operación, 146
predeterminados, 209–210
variables
compartidas, 255
de comunicación, 255
de dominio, 177
de iteración, 253, 257, 262–264, 639
de programa, 599
PHP, 790–791, 794–795
VARRAY, tipo de datos, 671–672
VDL (lenguaje de definición de vistas), 34
velocidad de transferencia en masa, parámetro de disco, 397, 905
velocidad de transferencia, parámetro de disco, 905
versionado de atributos, 723–725
vídeo, 22, 729–730, 873
vistas
actualizaciones incrementales, 241
almacenes de datos frente a, 861–862
arquitectura de tres esquemas y, 31–32
CREATE VIEW, comando, 241–242
definición, 12
DROP VIEW, comando, 241
equivalencia de vista, 537

especificación de, 240–241
especificar privilegios con, 686
implementación y actualización, 241–243
materialización de vista, 241
método de integración, 355–359
múltiples, 12
OQL, 642
Oracle 8, 672
serialización de vista, 537
vistas materializadas, 715
VSAM (Método de acceso de almacenamiento virtual), 457

**W**

WAL, 574–575
WAN (red de área extendida), 754
Web. *Véase también* PHP, lenguaje de *scripting*
arquitectura, 43
comercio electrónico, 22, 693–694
datos estructurados, 784–788
datos no estructurados, 784–788
datos semiestructurados, 784–788
panorámica de la, 783–784
semántica, 110
WHERE, cláusula, 214, 217–218, 228–229
WITH CHECK OPTION, cláusula, 243
WORM, discos, 391
WRITE, operación, 523

**X**

XDE (entorno de desarrollo extendido), 381
XML (Lenguaje de marcado extensible), 803–822
almacenar documentos, 813
consulta, 819–820
control del acceso, 693
convertir gráficos en árboles, 816–818
datos web usando, 22
definición, 783
documentos, 805–807
documentos, extraer de bases de datos, 814–816, 818–819
espacio de nombres, 943
modelo de datos jerárquico, 46, 803–805
repaso/ejercicios, 821–822
Schema, 807–812
SQL con, 243
XPath, expresiones, XML, 819–820
XQuery, XML, 820–821